Digital Whisper Technologic Papers

House Every Weekend - GLIBC Heap Exploitation

מאת Idan Banani

הקדמה

לפניכם המאמר המקיף ביותר על heap exploitation שהתפרסם כאן. זהו מעין סיכום המרכז את המידע הנחוץ כדי ללמוד את הנושא מאפס ויכול לשמש גם את מי שהנושא מוכר לו בתור quick reference או כדי להרחיב את יריעותיו. הוא מבוסס ברובו על מסמך מקורס בשם HeapLab מאת Max Kamper (חוקר ומפתח exploits, היוצר של Rop Emporium). כמשתמע משם הקורס – הוא כולל "מעבדת" תרגול המכילה אתגרים, קלים עד קשים, עם פתרונות מודרכים לדוגמה, וסביבת ניסויים (testbed) המאפשרת לעצב ולבדוק מתקפות עבור גרסאות build שונות של libc לבחירתכם. בדומה ל-Rop Emporium, הקורס מכיל תרגילים שמאפשרים למקד ולבודד את הלמידה של הנושא מנושאים משלימים (הנדסה לאחור וכו'). סשן לדוגמה החופף ברובו לפרק הראשון בקורס ניתן לצפייה כאן וכאן. למעוניינים – לינק לחלק הראשון בסדרה (מתוך 3, ה-3 יצא כנראה רק ב-2022) של הקורס עצמו - קישור (מכיל referral code של יוצר הקורס על מנת לתמוך בו עקב שיטת התגמול בפלטפורמה הלימודית). Heap exploitation לא נוראי ומפחיד כמו שאולי חשבתם, בפרט הודות לכלים כמו pwndbg (מבית היוצר של ספריית pwntools) ושאר כלים ויזואליים להמחשת מצב הזיכרון. מה שיפה בו, בין השאר, זה שהוא משאיר הרבה מקום ליצירתיות ודמיון של מפתח ה-exploit בכל הקשור להערמה על מנגנון ההקצאה. המתקפות המורכבות והמתוחכמות יותר (אם כי לפעמים נרצה את הדרך/וקטור התקיפה הכי פשוט ומהיר לניצול שעושה את העבודה) משלבות כמה אלמנטים שונים בצורת שרשרת (chain exploit), כאשר חלקם נוגעים למקצה הזיכרון במקרה שלנו. כראוי למחקר חולשות Low-level, נדרש כאן המון ריכוז, עבודה בצורה מסודרת ומתועדת היטב. מימוש מוצלח exploit מורכב יכול להימשך גם כמה שבועות ובמקרה של "שחקנים" בקטגוריית APT כנראה גם חודשים.

מאמר זה עוסק ב-Glibc ו-Malloc בסביבת לינוקס (ובפרט בהפצת Ubuntu), הוא נחשב למוקשח יחסית מפני מתקפות heap, ועל כן עשוי להוות בסיס מחשבתי טוב לפני התעסקות עם מימושים אחרים של מנגון הקצאות הזיכרון הנמצאים בשימוש דוגמת Jemalloc musl ו-Scudo.

לקינוח, בחלק השני במאמר נדגים פתרון צעד-אחר-צעד של אתגר מורכב המתמודד עם הגרסה האחרונה של glibc (2.34, האתגר היחיד שמסתובב ברשת שמקושר לגרסה זו) מתחרות corCTF שהתרחשה בסוף אוגוסט. דוגמאות ל-CTF-ים "ריאליסטיים" נוספים עם תרגילים ברמה קשה המתארת אתגרים מודרניים (defconCTF, plaidCTF , BlazeCTF, google-ctf, dragonctf, HitconCTF,C3CTF, BalsnCTF

רקע מומלץ: שימוש בכלי command line, יכולות כתיבת סקריפטים בפייתון, היכרות עם סביבות דיבאג דוגמת GDB, x86-64 assembly , Linux binary exploitation, C program memory layout , פיתוח ב-C, מערכות הפעלה, Reverse engineering , vTables (virtual functions table for exploiting file structures)

הבהרה: מפאת אילוצי זמן, רוב הטכניקות המפורסמות שמסוקרות לקראת סוף המאמר לא כוללות writeup שלם עם אנימציות לבינארי לדוגמה (של תוכנית אותה תוקפים, ושל ה-exploit כנגדה). מה גם שזה היה מכפיל/משלש את כמות העמודים במסמך. כאמצעי משלים – ניתן לפנות לקורס הנ"ל (למרות שהאורך הכולל של הסרטונים בכל חלק בסדרה הוא בין 6 ל-8 שעות, לפתור בעצמכם בצורה מסודרת את התרגילים וסיכום של כל הפרטים הקטנים יכול לקחת הרבה יותר מזה. מה גם שהמסמך עליו מבוסס המאמר לא כולל את כל הפרטים הקטנים ואת כל מגוון הרעיונות שמוצגים בקורס ומאפשרים לנו להשיג את המטרה. אך בשורה התחתונה – נדרשת יצירתיות, עבודה בצורה מסודרת ויכולת איתור חולשות בקוד של glibc) או למקורות המצורפים בסוף המסמך או לחפש CTF write-ups מתחרויות עבר. בפרופיל הגיטהאב שלי יהיה ניתן למצוא בהמשך לכל הפחות cheat-sheets ומיני סיכומים בנושא.

דגשים והמלצות לגבי עבודה עם מסמך זה:

בכל עת מומלץ לנסות לאתר בקוד של glibc את הקטעים הרלוונטיים למה שאתם קוראים. כמו כן ניתן ואף מומלץ תוך כדי או לאחר שאתם קוראים על טכניקה לחפש תרגילים קשורים (בין אם ללא פתרון או כאלו שכוללים פתרון מודרך) כדי לוודא שההבנה שלכם תואמת למציאות. את התרגילים מתוך הקורס המוזכר אין באפשרותי לשתף, לכן צירפתי בסוף המאמר כמה אתרים רלוונטיים לתרגול. תמיד אפשר לחפש אתגרים מתחרויות CTF בנושא מסויים ע"י חיפוש כמו: "subject/variable_name glibc_version heap exploit ctf" בטלגרם יש ערוץ נחמד שאוסף אוטומטית פתרונות חדשים מ-ctftime ומצרף להם את הנושאים שלהם. ניתן לחפש גם שם לפי מילות מפתח.

חשוב לומר שגם אם אתם נתקעים בפתרון של תרגיל מסויים בתחילת הדרך, ובפרט בתחום של heap-exploits, זה טבעי לחלוטין ואין סיבה להרגיש רע לגבי זה. המאמץ תמיד ישאר, הוא פשוט יחלוף מהר יותר ככל שתצברו ניסיון. מתודולוגיה לדוגמה עבור למידה של אקספלוטציה אפשר למצוא כאן: חלק 1, חלק 2, חלק 3

מסיבה כלשהי, מרבית המקורות ברשת על היבטים מורכבים של heap exploitation כמו ניצול חולשות במנגנונים אחרים של glibc כדי להתגבר על הגנות heap מודרניות בדרכים "יצירתיות" הם בשפות סינית/יפנית/קוריאנית/מנדרינית וכו' (הנפוצות במדינות אסיה), לא ברור האם זה כי אתגרי CTF קשים הם חלק ממנהגי התרבות שלהם (בתור משחק) או כי מדובר במדינות מיליטנטיות. אל תחששו להשתמש בכלי תרגום כדי ללמוד גם ממקורות אלו.

היבטים/אתגרים טכניים הקשורים לפיתוח תוכנה

סביר להניח שבמוקדם או במאוחר תרצו או תאלצו לערוך ולקמפל קוד שלכם או של מישהו אחר שאתם מעוניינים לבחון (בין אם כזה שמכיל כבר אקספלויט של המתקפה או כזה שדורש מכם לבנות אותו) – תידרשו לעשות זאת בקונפיגורציה המתאימה להגנות (Hardening) ולשיטת הייצור הרצויות. האופציה הפשוטה ביותר עבור קובץ בודד היא דרך שורת הפקודות/טרמינל עם אפשרויות/דגלים מתאימים למשל עם gcc. כדי להפוך את התהליך לאוטומטי ומסודר יותר, רצוי להשתמש בסקריפט דוגמת קובץ Makefile עבור gcc ו-clang או קובץ CMakeLists.txt עבור CMake (ייתכן וחלקכם משתמשים בעבודה ב-GNU Autotools עבור פרויקטים גדולים), או לעשות זאת דרך IDE בו יוגדרו הדגלים דרך GUI/ידנית (CLion למשל שהינו חינמי עבור סטודנטים). לצרכי למידה - רצוי לקמפקל עם debug symbols בעזרת הוספת הדגל g– לדגלי הקימפול (כך שתוכלו בעת דיבאג לראות לא רק את האסמבלי, אלא גם את שורות קוד המקור בהן אתם נמצאים). כמו כן, יתכן שבשלב כלשהו תרצו לקמפקל בעצמכם את glibc אחרי שתבצעו בו שינויים בקוד המקור כמו למשל לבטל מנגנון בשם tcache (סוג של patching), ובפרט עם debugging symbols או לחפש גרסאות מוכנות שקומפלו כך (כמו אלו שמסופקות בקורס הנ"ל) כדי לקבל את היכולת הנ"ל גם עבור הקוד של libc.

רשימה של דגלי ה-"CFLAGS" ש-gcc תומכת בהם ניתן למצוא כאן, וכאן. רשימה של דגלי ה-"LDFLAGS" שה-(ld) GNU linker תומך בהם ניתן למצוא כאן. בפועל – בד"כ נשתמש רק בחלק קטן מהם. דוגמאות ל-makefiles העושים שימוש ב-clang ניתן לראות באתגר השב"כ 2021.

עניין נוסף הוא שלב ה-linkage. במקרה של בינארי מסוג dynamically linked, יתכן ותתקשרו מול צד השרת שבו רצה תוכנית מקובץ בינארי (elf למשל) שמקושר לגרסה מסויימת של libc המצא אצלו (שקיבלתם אותה או שיכול להיות שתצטרכו להשיג אותה ע"י איתור שלה ברשת בעזרת כלים כמו libc database או בשיטות צוות-אדום). במקרה שכזה נצטרך לבצע patching לבינארי בעזרת כלים כמו pwninit (לא תמיד עובד) או patchelf (כי אין לנו אפשרות לבצע קומפילציה בעצמנו ולתת לה דגלים מתאימים של rpath ו--dynamic linker). במקרה של שרת המריץ הפצה שונה של לינוקס, אם נרצה להריץ את הבינארי בהפצה שונה ממנו אצלנו, נצטרך להשיג גם את קובץ ה-ld.so (loader) מ-package המכיל את אותו libc (למשל מתוך binutils)

על מנת להריץ את הבינארי בתצורת שרת/backend, בדומה לנעשה ב-CTFs ומערכות אמיתיות המשרתות clients, ניתן לעשות זאת בעזרת כלים פשוטים כמו ynet/xinetd/socat או בעזרת כלי DevOps כמו docker containers. בין היתרונות של שימוש ב-docker לצרכים אלו: הרצת הבינארי במחשבים אחרים (פתרון בעיית התלויות שמוכרת כ-"אבל אצלי במחשב זה עבד" ע"י הרצת השרת והבינארי בסביבה מבודדת משאר הספריות המתוקנות במערכת ההפעלה הודות לשימוש ב-image "קל-משקל" של הפצה נקייה/מותאמת אישית מהרשת) וכמו כן זה מאפשר לנו התקנה מהירה של סביבה המגיעה עם גרסאות מסויימות של כלים/ספריות המותקנים בבת אחת, אוטומטית. דוגמה מעניינת לתשתית CTF מורכבת (מלבד הפיצ'ר של הדגלים שמחוללים אקראית עבור כל קבוצה) העושה שימוש ב-docker, socat ו-Makefiles היא מתוך תחרות של חיל האוויר האמריקאי Hack A Sat 2 Qualifiers (אם כי פורסם רק חלק מהקוד של התשתית שלהם)

וכעת נתחיל עם החלק התאורטי:

GLIBC – GNU C library

מספקת את ספריות הליבה עבור מערכת GNU ומערכות GNU/Linux וכמו כן עבור מערכות הפעלה רבות אחרות שמשתמשות בלינוקס כגרעין. ספריות אלו מספקות API-ים הכוללים תשתיות בסיסיות כמו open, read, write, malloc, printf, getaddrinfo, dlopen, pthread_create, crypt, login, exit.

פרויקט ה-GLIBC הינו בקוד פתוח ומתוחזק על ידי קהילת מפתחים כבר יותר מ-3 עשורים.

Malloc: (לא הפונקציה malloc())

זהו השם שניתן למקצה הזיכרון (memory allocator) של GLIBC. ניצול חולשות במנגנון זה מהווה מסורת האקרים כבר יותר מ-20 שנים ועדיין נחשב לנושא אקטיבי.

זהו אוסף של פונקציות ומטא-דאטה שמשתמשים כדי לספק לתהליך רץ זיכרון דינאמי (בזמן ריצה לעומת זיכרון סטטי המוקצה בזמן קומפילציה).

המטא-דאטה מורכב מ-arenas (מבנים שכל אחד מהם משמש לניהול heap אחד או יותר ועשוי להיות משותף בין כמה threads), heaps (אין שום קשר למבנה נתונים בעל שם זהה מאלגוריתם heapsort, השם נבחר כך משום שאזור זה מכיל חתיכות זיכרון בגדלים משתנים) – שטח גדול בזיכרון המכיל בלוקים רציפים הניתנים לחלוקה ל-chunks , ו-chunks – המבנים שבתוכם נשמר המידע מהמשתמש.

הפונקציות של Malloc משתמשות ב-arenas וב-heaps (המוצבעים על ידם) על מנת לבצע תנועות (בסגנון "עובר ושב" כמו בנק) של חתיכות זיכרון אל מול/עבור תהליך.

Malloc Internals – המימוש הפנימי של המנגנון

Chunks

אבניי הבניין של הזיכרון ש-Malloc עוסק בו, לרוב הם מופיעים בצורה של חתיכות של זיכרון מה heap, אם כי הם יכולים להיווצר כישות נפרדת (אזור רציף משלהם במרחב הזיכרון הוירטואלי של התהליך הניתן לבחינה ע"י פקודות כמו proc info / vmmap מתוך pwndbg/GDB) ע"י קריאה ל-()mmap (system call) בין אם ישירות או בעקיפין (לדוגמה המשתמש קרא לפונקציית הספריה ()malloc שבתורה החליטה לשרת את הבקשה ע"י ()mmap). Chunks מורכבים משדה גודל ושדה עבור מידע של המשתמש כמוצג באיור 1. על מה שמאוחסן מסביב לשדות אלו נדבר בהמשך.

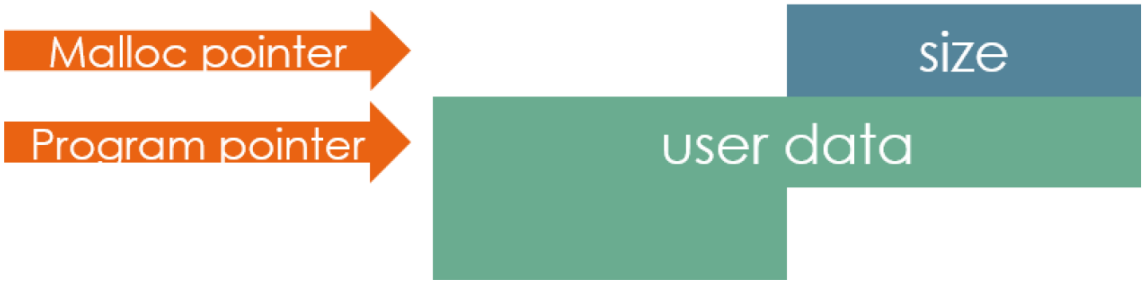


Figure 1: Chunk layout

איור 1 מבנה ה-Chunk. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

בעוד תוכניות (קוד) עובדות עם מצביעים לחלק של ה-user data, malloc מחשיב את ההתחלה של ה-chunk כ-8 בתים לפני מיקום שדה ה-size (היוצא מן הכלל הוא כאשר מנגנון שיוצג בהמשך בשם Tcache מופעל. ה-Tcache מחזיק מצביעים ל-user data כמו התוכנית).

שדה ה-size מציין את כמות הבתים של ה-user data (אלו שניתנים לכתיבה ע"י המשתמש) + כמות הבתים שתופס שדה ה-size בעצמו – 8 בתים, כך שעבור chunk עם 24 בתים של user data למשל,ה-size field יכיל את הערך 0x20, או 32 בדצימלי. **מעתה ואילך נתייחס לערך שדה זה כגודל ה-chunk.** גודל ה-chunk המינימלי שניתן לשימוש הוא 0x20, אם כי בתוך הקוד של malloc נעשה גם שימוש ב-fencepost chunks בגודל 0x10. נזכיר אותם בהמשך, כרגע הם לא רלוונטיים.

ערכי גודל chunk גדלים בקפיצות של 16 בתים, כך שהגודל הבא אחרי 0x20 הוא 0x30, ואז 0x40 וכו'. המשמעות היא כי ה-least significant nibble (4 הביטים התחתונים) של ה-size field לא משפיעים על גודל ה-chunk, במקום זאת הם מכילים דגלים (כל ביט) שמציינים את מצב ה-chunk. דגלים אלו, מהביט הנמוך לגבוה (מימין לשמאל): PREV_INUSE – כאשר דלוק (set) מציין שה-chunk הקודם נמצא בשימוש, כאשר מאופס (clear) מציין שה-chunk הקודם חופשי (ניתן להקצאה בדרך כלשהי). IS_MMAPPED – כאשר דלוק, מציין ש-chunk זה הוקצה ע"י mmap() ואחרת רגיל על ה-NON_MAIN_ARENA.heap – כאשר דלוק, מציין ש-chunk זה לא שייך ל-main arena. לדגלים אלו חשיבות רבה אך לא תמיד המנגנון מתייחס לכל הדגלים (יתרון לתוקף שמוגבל ביכולתו לבחור ערך שרירותי עבורם). הביט הרביעי לא בשימוש. ניתן לראות אותם באיור 2.

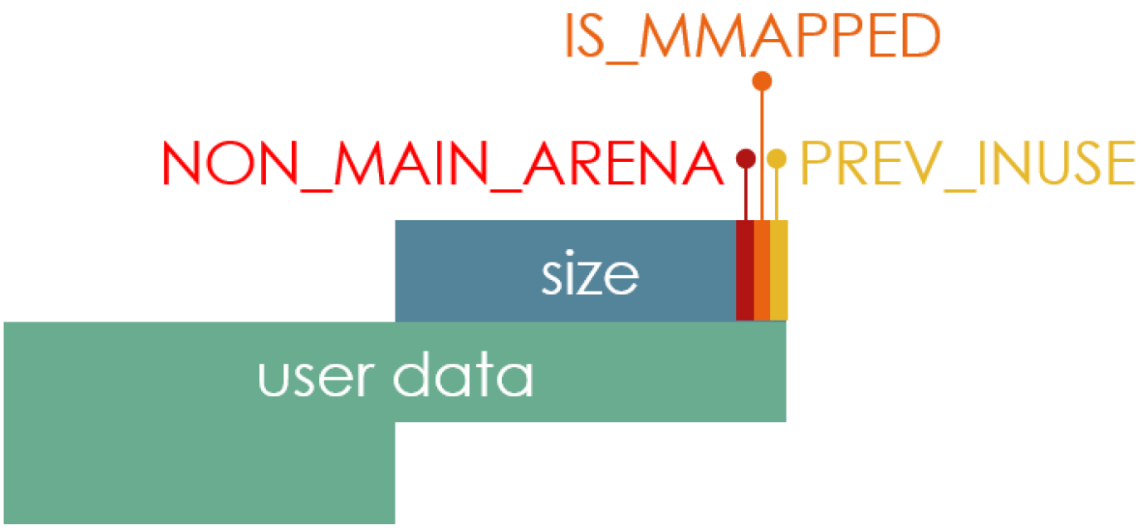


Figure 2: Size field flags

איור 2 דגלי שדה הגודל. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

שדה ה-user data הוא הזכרון הזמין עבור התהליך שביקש אותו. הכתובת של שדה זה היא זו שמוחזרת מפונקציות הקצאת הזיכרון של malloc (malloc(), calloc() , realloc() וכו'). Chunks יכולים להיות באחד משני מצבים שאינם יכולים להתקיים בו זמנית (mutually exclusive): מוקצה או משוחרר.

כאשר chunk משוחרר, עד 5 quadwords (1qw=4*2B=8Bytes) מתוך ה-user data שלו מיועדים מחדש כ-metadata של malloc ואף עשויים להפוך לחלק מה-chunk הבא (נדגים בהמשך). פירוט על ה-metadata שבשימוש בכל bin (bins הם המקומות בזיכרון בהם נשמרים מצביעים לרשימות מקושרות של chunks משוחררים לצרכי מחזור) נמצא תחת החלק של Arenas במאמר זה.

ה-qwadword הראשון של ה-user data ייועד מחדש כ forward pointer (fd) כאשר ה-chunk משוחרר, כל ה-bins משתמשים ב-forward pointer ברשימה המקושרת שלהם. ה-qw השני ייועד מחדש כ-backward pointer בתוך chunks משוחררים שקושרו אל תוך רשימה מקושרת דו-כיוונית כלשהי כמו זו של ה-unsortedbin או ה-smallbins. הqwadwords השלישי והרביעי ייועדו מחדש כמצביעים בשם fd_nextsize ו-bk_nextsize אך ורק במקרה שה-chunk המשוחרר יקושר ל-largebins (רק הוא עושה בהם שימוש) המיקום של metadata זה מוצג באיור 3:

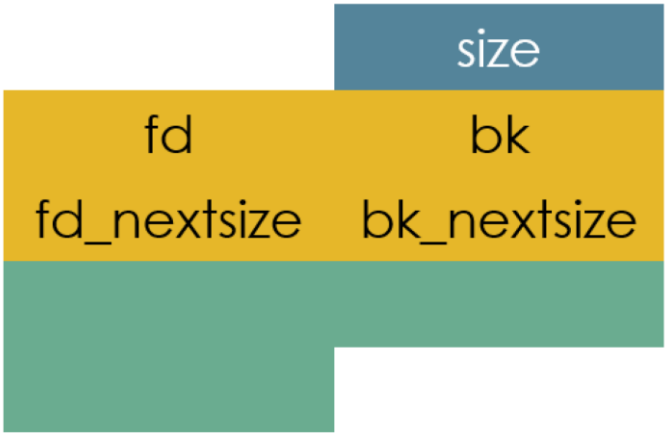


Figure 3: Inline malloc metadata

איור 3 שדות ניהול של chunk. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

ב-bins שתומכים באיחוד (consolidation) , ה-quadword האחרון של ה-user data ב-chunk המשוחרר מיועדים מחדש כשדה בשם prev_size שמציין את הגודל של ה-chunk המשוחרר בדומה לערך ה-size field אך בניגוד אליו – ללא הדגלים (מאופסים. נעשה בד"כ ע"י bitwise AND masking). Malloc מחשיב את שדה ה-prev_size **כחלק מה-chunk הבא** והנוכחות שלו מלווה באיפוס דגל ה-PREV_INUSE של ה-chunk הבא (כי ה-chunk שלפניו משוחרר כעת), כמודגם באיור 4:

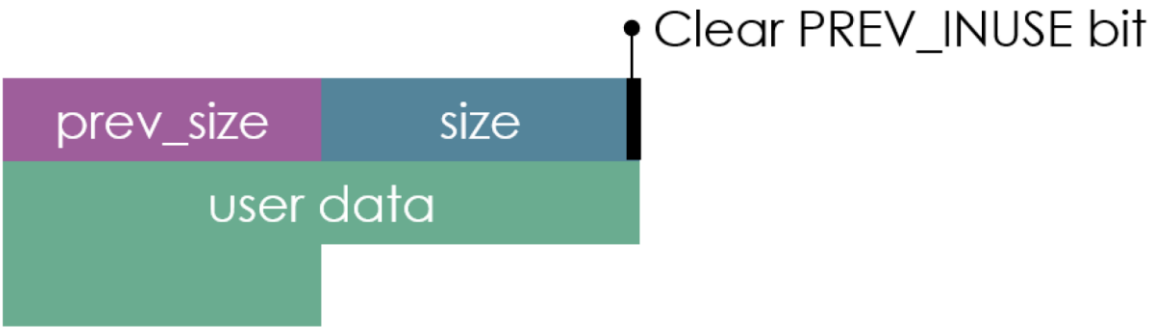


Figure 4: A prev_size field

איור 4 שדה גודל ה-chunk הקודם. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

בגרסאות של GLIBC >= 2.29, ה-qw השני מתוך ה-5 שהוזכרו לעיל של chunks משוחררים שקושרו אל תוך Tcachebin מיועדים מחדש כשדה "key" המשמש לזיהוי של תרחישי double free (הגנה מפני). איור 5 מתאר chunk משוחרר המקושר ל-tcachebin :

Figure 5: Tcache metadata

איור 5 tcache chunk metadata. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

Unlinking

במהלך פעולות הקצאה ושחרור, chunks עשויים להינתק מה-free list שבה היו, ה-chunk שמוסר מהרשימה בד"כ נקרא chunk "victim" (הקורבן שנבחר) בקוד המקור של Malloc. בהמשך נרחיב במידע נוסף על ה-free lists בחלק של Arenas. ישנן כמה דרכים בהן עשוי להתבצע Unlinking:

Fastbin & Tcache Unlink:

ה-fastbins וה-tcache משתמשים ברשימה מקושרת חד כיוונית במימוש LIFO (Last in is First out), התרת chunks מרשימות אלו כוללת העתקה של של ה-fd של ה-victim chunk אל תוך שדה ראש הרשימה (ה-head החדש). מידע נוסף על ה-Fastbin מופיע תחת Arenas בהמשך.

Partial Unlink:

מתרחש כאשר chunk מוקצה מתוך unsortedbin או smallbin. מתבצעת עקיבה אחרי מצביע ה-bk של ה-victim chunk (נקרא לתוכן זה ככתובת ה-destination chunk) והכתובת של ראש ה-bin (כתובת שנמצאת ב-arena של אותו bin ולא ב-heap) מועתקת אל תוך ה-fd של ה-destination chunk [וזאת כי במקרה שלהם הקורבן נבחר בשיטת FIFO, כלומר נלקח מהזנב]. הסיבה למילה Partial בשם זה היא שבניגוד לשתי הכתיבות שמתבצעות בעת הסרה סטנדרטית מרשימה דו כיוונית לצורך שימור המבנה, כאן מתבצעת רק אחת מהן [כי אין צורך במעבר לכך]. מידע נוסף על ה-unsortedbin ו-smallbins נמצא תחת Arenas.

Full Unlink:

מתרחש כאשר chunk מאוחד אל תוך free chunk אחר, וגם כאשר chunk מוקצה מתוך ה-largebins או ע"י חיפוש מסוג binmap (מוסבר בהמשך). מתבצעת עקיבה אחר ה-fd של ה-victim chunk, וה-bk של ה-victim מועתק לתוך ה-bk של ה-destination. לאחר מכן (באופן סימטרי) מתבצעת עקיבה אחר ה-bk של ה-victim, וה-fd של ה-victim מועתק לתוך ה-fd של ה-destination.

Heaps

הינם בלוקים רציפים של זיכרון. מכילים את ה-chunks ש-Malloc מקצה לתהליך. הם מנוהלים באופן שונה כתלות בהאם הם שייכים ל-main arena או לא. (מידע נוסף תחת Arenas)

Heaps יכולים להיווצר, להתרחב, להתכווץ או להימחק. ה-Heap של ה-main arena נוצר במהלך הבקשה הראשונה של זיכרון דינאמי. Heaps של Arenas אחרים נוצרים ע"י קריאה לפונקציה ()new_heap.

Heaps של ה-Main arena גדלים וקטנים ע"י syscall ()brk, אשר מבקשת זיכרון נוסף מה-Kernel או מחזירה לו זיכרון. Heaps של Arenas אחרות (Non-main-arena) נוצרים עם גודל קבוע והפונקציות ()grow_heap ו-()shrink_heap ממפות יותר או פחות זיכרון כניתן-לכתיבה (writeable).

Heaps השייכים ל-Non-main-arena גם עשויים (בנוסף לכך) להרס ע"י המאקרו ()delete_heap במהלך קריאות ל-()heap_trim.

Arenas

Malloc מנהל את ה-Heap של תהליך בעזרת מבנים מסוג malloc_state, הידועים כ-arenas. אותם arenas מכילים בעיקר "bins" המשמשים למחזור של free chunks בתוך זכרון ה-heap. Arena יחיד יכול לנהל מספר heaps בו זמנית.

Arenas חדשים נוצרים ע"י קריאה לפונ' ()_int_new_arena ומאותחלים ע"י ()malloc_init_state.

המספר המקסימלי של Arenas המנוהלות במקביל מבוסס על מספר הליבות (cpu cores) הזמין לתהליך.

Arena Layout

איור המתאר את מתווה ה-Arena (מוגדר בתוך struct malloc_state)

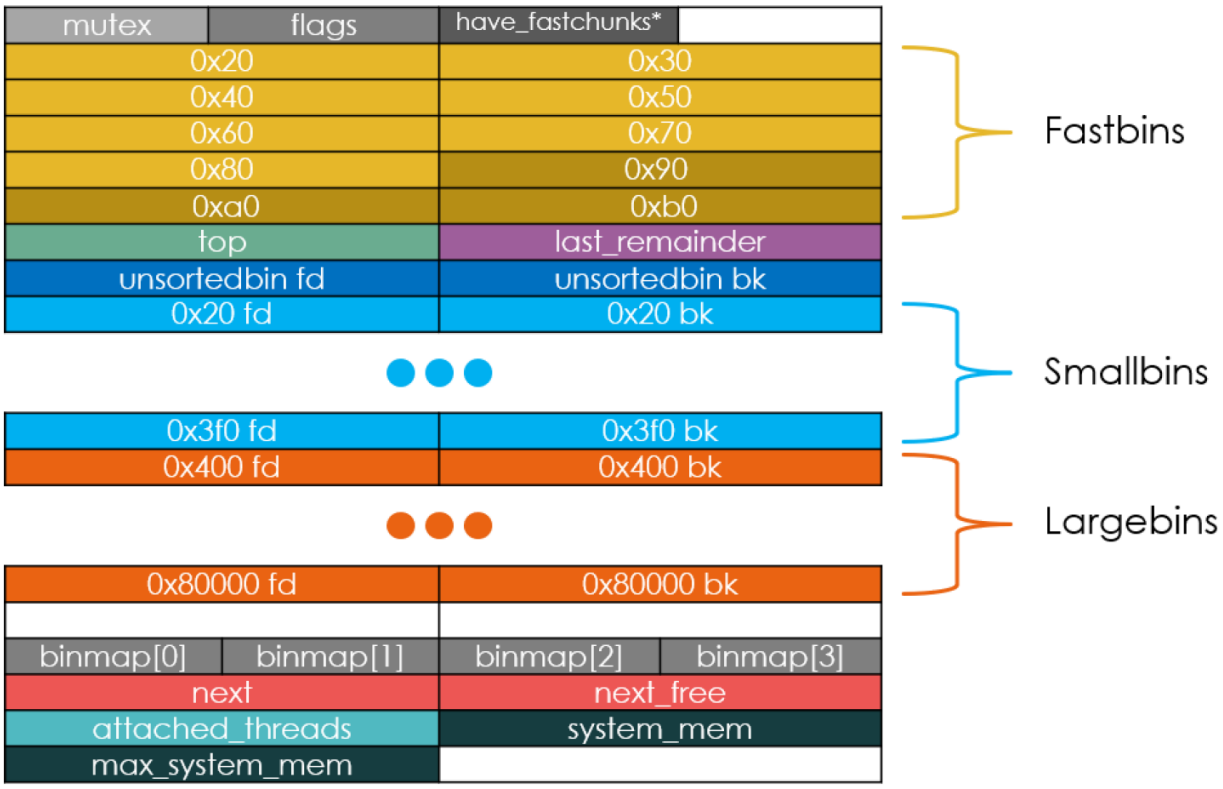


Figure 6: Arena layout

איור 6 תצורת ה-Arena. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

mutex: מבצע סיריאליזציה לגישה ל-arena. Malloc נועל את ה-mutex של ה-arena לפני בקשת זיכרון heap ממנו.

flags: מחזיקים מידע כמו האם זיכרון ה-heap של ה-arena רציף או לא. (יתכנו מצבים בהם ניתן להערים על Malloc לחשוב שיש חורים ב-heap. מתקשר ל-top chunk)

have_fastchunks: מפורש כ-bool כמציין האם ה-fastbins ריקים או לא. מודלק כאשר chunk מקושר לתוך fastbin ומאופס ע"י malloc_consolidate() (יתבהר בהמשך)

הערה – שדה זה וה-padding dword שאחריו (בלבן) מופיעים רק בגרסאות 2.27 =< . בגרסאות ישנות יותר (2.26 ומטה) הם חלק משדה ה-flags.

Fastbins

קוד המקור של Malloc מתאר את ה-fastbins כ-bins מיוחדים שמחזיקים. הם מהווים אוסף של רשימות מקושרות חד-כיווניות, ללא מעגלים (acyclic) שמחזיקים free-chunks בגדלים ספציפיים. ישנם 10 fastbins לכל arena*, כל אחד מהם אחראי "להחזיק" free-chunks בגדלים 0x20 עד 0xb0. לדוגמה: 0x20 fastbin

יחזיק רק free-chunks בגודל 0x20. אם כי רק 7 מתוך ה-fastbins האלו זמינים תחת תנאי ברירת-מחדל. הפונקציה ()mallopt יכולה לשנות מספר זה ע"י שינוי ערך המשתנה global_max_fast.

מצביע לראש (במקרה זה מצביע ל-chunk הראשון ברשימה המקושרת) של כל fastbin שוכן בתוך ה-arena שלו, אם כי הקישורים בין chunks עוקבים מאוחסנים inline (במקום, בתוך איברי הרשימה עצמם). ברגע שה-chunk משוחרר (ומיועד ל-fastbin), הquadword הראשון של ה-user data שלו מיועד מחדש כ-forward pointer (fd) ומעתה ואילך הינו חלק מהרשימה (מקושר אל תוך ה-fastbin). fd בעל ערך null מציין שזהו ה-chunk היחיד והאחרון ב-fastbin.

Figure 7: A fastbin linked list

איור 7 רשימה מקושרת של fastbin. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

Fastbins עובדים בשיטת LIFO, שחרור chunk אל תוך fastbin מקשר אותו כראש של אותו fastbin. כמו כן, בקשה של chunks בגודל שתואם ל-fastbin שאינו ריק, יגרמו להקצאה (חוזרת, מחזור) של ה-chunk בראש אותו fastbin.

Free chunks מקושרים ישירות אל תוך ה-fastbin התואם להם במידה וה-tcache המתאים לאותו גודל מלא. בעת בקשת chunk בגודל הנופל בטווח של גדלי ה-fastbin יבוצע חיפוש ב-fastbin רק לאחר שבוצע חיפוש ב-tcache ולפני שמבוצע חיפוש בכל שאר סוגי ה-bins.

*ה-bin בגודל 0xb0 נוצר בטעות, זאת עקב חוסר השיוויון בין איך שמתייחסים לקבוע MAX_FAST_SIZE כגודל בקשה לעומת איך שמתייחסים למשתנה global_max_fast כגודל chunk.

Top (top chunk)

בעבר היה נקרא גם "wilderness". מתוך malloc.c : top chunk הוא ה-chunk הזמין שהינו ה-topmost (בעל הכתובת הגבוהה ביותר) כלומר זה שגובל בסוף הזכרון הזמין. לאחר ש-Arena חדש מאותחל, top chunk תמיד יהיה קיים ותמיד יש רק אחד כזה לכל Arena. בקשות מקבלות זיכרון מה-top chunk רק כאשר הן לא יכולות להענות ע"י bin כלשהו אחר באותו Arena.

כאשר ה-top chunk קטן מדי כדי לשרת בקשה מתחת ל-mmap treshold (בקשות מעל לסף מקבלות זיכרון מ-()mmap), malloc ינסה להגדיל את ה heap שבו ה-top chunk שוכן בעזרת הפונקציה ()sysmalloc ולאחר מכן ירחיב את ה-top chunk. במידה ונכשל, heap חדש יוקצה ויהפוך ל-top chunk של אותו arena, וכל הזכרון שנותר ב-top chunk הישן משוחרר. כדי להשיג זאת, malloc ממקם שני chunks "fencepost" בגודל 0x10 כל אחד בסוף ה-heap הישן (איפה שמתחיל החור) כדי להבטיח שניסיון לאיחוד-קדמי

(forward consolidation) לא יגרום ל-Out-of-bounds read. הפונקציות המשמשות לניהול ה-heaps מופיעות תחת הכותרת Heaps.

Malloc מנהל מעקב אחר גודל הזכרון הנותר ב-top chunk בעזרת ה-size field שלו, ביט ה-PREV_IN_USE של אותו שדה תמיד דלוק. top chunk תמיד מכיל מספיק זיכרון כדי להקצות chunk בגודל מינימלי ותמיד יסתיים בגבול של סוף דף זכרון. כלומר הוא תורם ל-"alignment של ה-heap" בגודל של דף, במקרה שלנו מדובר ב-4KB. כלומר סך כל גודל ה-heap לא כולל ה-qw ה-1 (אמנם המצביע לheap מצביע לשם) יהיה ערך שבו שלושת הבתים התחתונים מאופסים (תזכורת: ערכי ה-page size הזמינים במערכת תלויים ב-ISA, סוג המעבד ומוד הפעולה (מיעון). מע"ה בוחרת אחד או יותר מבין הגדלים הנתמכים בארכיטקטורה. עד כאן לבינתיים לגבי זיכרון וירטואלי ו-paging)

Last_remainder

שדה זה מחזיק את כתובת ה-chunk שנותר מפעולת ה-remaindering הקודמת (פעולה בה הבקשה מקבלת מענה מתוך חלק של free chunk וחלק השארית נשאר פנוי לשימוש). הוא מאוכלס כאשר מגיעות בקשות בגודל הנופל בטווח של ה-smallbin שמסופקות מתוך התוצר הראשי של פעולת ה-remaindering (לא השארית) על chunk שהגיע מה-unsortedbin.

כאשר גודל הבקשה מחוץ לטווח של ה-smallbin, שדה זה לא יאוכלס לאחר פעולת remaindering על ה-largebin וה-unsortedbin או מחיפושי binmap.

כדי לבצע remaindering מתוך unsortedbin, ה-last remainder chunk חייב להיות הראש של אותו unsortedbin. עוד על כך בחלק הבא:

Unsortedbin

הינו רשימה מקושרת דו-כיוונית מעגלית שמחזיקה free chunks מכל גודל. המצביעים לראש והזנב שלה שוכנים בתוך הArena המשוייכת כמתאור באיור 6 (בצורה של "chunk" השלמה דמיוני שאינו מכיל user data, אלא רק מצביעים ההופכים את הרשימה למעגלית ונותנים גישה אליה) ואילו שדות ה-fd וה-bk המקשרים בין chunks עוקבים נשמרים inline בתוך על גבי ה heap (בתוך ה-chunk's metadata).

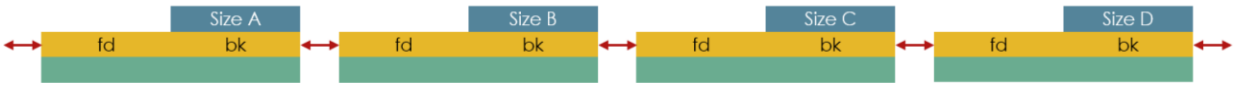


Figure 8: Unsortedbin doubly linked list

איור 8 רשימה מקושרת דו-יכוונית של Unsortedbin. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

Free chunks מקושרים ישירות אל תוך ראש הרשימה המקושרת הנ"ל כאשר ה-tcachebin התואם להם מלא או כאשר הם מחוץ לטווח של גדלי ה-tcache (0x420 ומעלה תחת תנאי ברירת מחדל). בגרסאות בהן GLIBC מקומפל ללא ה-tcache (GLIBC <= 2.25) במצב ברירת מחדל. נרחיב תחת Tcache לגבי היוצאים מן הכלל) free chunks מקושרים ישירות אל תוך ראש הרשימה כאשר הם מחוץ לטווח גדלי ה-fastbin (כלומר החל מ-0x90 ומעלה תחת תנאי ברירת מחדל).

חיפוש ב-unsortedbin מתבצע לאחר שבוצע חיפוש ב-tcache, ב-fastbins וב-smallbins כאשר הגודל נופל בתוך טווחים אלו אך לפני ה-largebins. חיפוש ב-unsortedbin מתחיל מסוף ה-bin ומתקדם לכיוון החזית (כזכור, הקורבן נבחר בשיטת FIFO), אם chunk מתאים בדיוק לגודל הבקשה לאחר נרמול (עיגול כלפי מעלה לכפולה של 0x10 לאחר שמוסיפים 8B לגודל הבקשה) – הוא יוקצה והחיפוש יפסיק, אחרת הוא ממויין אל תוך ה-smallbin או ה-larbin המתאים.

אם ה-chunk שנבדק במהלך סריקת ה-unsortedbin לא מתאים בדיוק אבל הוא ה-last remainder וגדול מספיק כדי "לבצוע" אותו שוב (remaindering) אז כך יקרה. Chunks שהם תוצר של פעולת remaindering שכזו מקושרים בחזרה אל ראש ה-unsortedbin (וכמובן שהנתח שנגזר בהתאם למידות של הבקשה מועבר למשתמש ומוצא מהרשימה. מדובר בחלק עם הכתובת הנמוכה מבין השניים).

יוצא שבעצם Malloc נותן הזדמנות יחידה לכל chunk ב-unsortedbin להיות מוקצה ברגע שהוא נסרק לפני שהוא ממויין. כך שבעצם הוא משמש כסוג של תור שמוכנסים אליו chunks במהלך free וגם בעת malloc_consolidate() ונלקחים (לשימוש או הלאה לאחד ה-bins) בתוך malloc(). דגל ה-NON_MAIN_ARENA תמיד מאופס עבור unsorted chunks ועל כן לא נקלח בחשבון בעת השוואת size fields (כל הקלה שכזו לטובת התוקף יכולה להקל עליו לעבור בביטחה mitigations של השחתת זכרון)

Smallbins

אוסף של רשימות מקושרות דו-כיווניות מעגליות שכל אחת מהן מחזיקה free chunks בגודל מסויים. ישנם 62 smallbins בכל arena, שכל אחד מהם אחראי על chunks בגדלים 0x20 עד 0x3f0 כאשר חלק מהם חופפים לגדלי ה-fastbin (בעצם טווח הגדלים של ה-fasbin מוכל בתוך טווח זה). לדוגמה: ה-smallbin המתאים לערך 0x20 מכיל רק free chunks בגודל 0x20, וה-smallbin 0x300 מכיל רק free chunks בגודל 0x300.

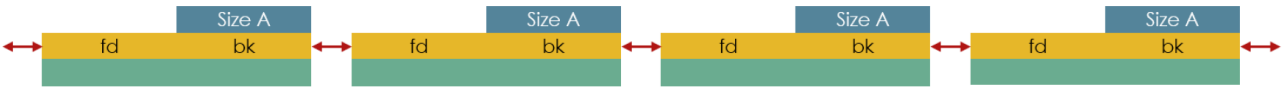


Figure 9: Smallbin doubly linked list

איור 9 רשימה מקושרת דו-כיוונית של Smallbin. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

המבנים עובדים בשיטת FIFO, מיון chunk לתוך smallbin (כזכור, מיון הוא מתוך ה-unsortedbin) מקשר אותו אל הראש של אותו smallbin. באותו אופן, בקשה של chunks בגודל שתואם ל-smallbin שאינו ריק תגרור הקצאה של chunk מהזנב של אותו smallbin.

חיפושים ב-Smallbin מבוצעים כאשר גודל הבקשה נופל בטווח גדלי ה-smallbin בנק' זמן שהינה לאחר חיפוש ב-Tcache, ואחרי חיפוש ב-fastbin (אם גודל הבקשה נופל בטווח זה), אך לפני שמבוצע חיפוש בשאר ה-bins

Largebins

אוסף של רשימות מקושרות דו-כיווניות מעגליות שכל אחת מהן מחזיקה free chunks בטווח מסויים של גדלים (כאשר האחרון מביניהם לא חסום מלמעלה). יש 63 largebins בכל arena. מדובר על גדלים של 0x400 ומעלה. לדוגמה: ה-largebin 0x400 מחזיק free chunks בגדלים בטווח 0x400-0x430, בעוד שלמשל largebin 0x2000 מחזיק free chunks בגדלים 0x21f0 – 0x2000.

הראש של כל largebin (מצביע) שוכן בתוך ה-arena שלו, בעוד הקישורים בין chunks עוקבים בתוך אותו bin שמורים inline (בתוך ה-chunk's metadata). Free chunks מקושרים אל תוך ה-largebin המתאים להם אך ורק מתוך ה-unsortedbin של אותו arena כאשר מתרחש מיון של אותו chunk (unsortedbin scan).

רשימות של ה-Largebins מתוחזקות בסדר יורד מבחינת הגודל, כלומר ה-chunk הגדול ביותר ב-bin מסויים נגיש דרך ה-fd של אותו bin (ה-chunk הדמיוני בתוך ה-arena שלא מכיל user data), וה-chunk הקטן ביותר ב-bin נגיש בעזרת ה-bk של ה-bin. כאשר chunk מקושר אל תוך ה-largebin, ה-quadword ה-1 של ה-user data שלו מיועד מחדש כ- forward pointer (fd) וה-qw ה-2 מיועד מחדש כ- backword pointer (bk).

Figure 10: Largebin doubly linked list with skip list

איור 10 רשימה מקושרת דו-כיוונית עם רשימת דילוגים של Largebin. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

ה-chunk הראשון מבין שאר ה-chunks עם אותו גודל (מדוייק) שמקושר אל ה-largebin יעשה שימוש ב-quadwords ה-3 וה-4 של ה-user data (ייועדו מחדש) כמצביעי skip list (רשימת דילוגים) בשם fd_nextsize ו-bk_nextsize בהתאמה. המצביעים הללו מעצבים רשימה מקושרת דו-כיוונית מעגלית נוספת המחזיקה את ה-chunk הראשון מכל גודל הקיים באותו bin. היה וה-chunk הראשון מגודל מסויים קושר

(לראשונה) לתוך largebin מסויים, ה-chunks הבאים באותו גודל מתווספים אחרי אותו chunk ראשון בגודל זה על מנת למנוע ניתוב-חוזר (rerouting) של ה-skip list.

חיפוש בתוך ה-Largebins מבוצע במהלך טיפול בבקשות ל-chunks בגודל 0x400 ומעלה, רק לאחר סריקה של ה-unsortedbin, אך לפני חיפוש ב-binmap (יוסבר בהמשך). במהלך חיפוש ב-largebin, Malloc מוודא שה-bin המתאים מחזיק chunk מספיק גדול כדי לתמוך בבקשה; אם כך, ה-bin נסרק מהסוף להתחלה בחיפוש אחר chunk עם התאמת גודל מדוייקת או גדול מגודל הבקשה (המנורמל).Malloc יקצה (יבחר) chunk שנמצא גם ב-skip list רק במידה וזהו ה-chunk האחרון מאותו גודל, אחרת הוא יקצה את ה-chunk מאותו גודל שנמצא אחרי (בכיוון הסריקה) אותו "skip chunk" כדי להימנע מלהצטרך לבצע reroute על ה-skip list לעיתים תכופות (כמה שפחות פעולות תחזוקה מטעמי ביצועים. גם ה-Tcache שנסביר עליו בהמשך הוכנס משיקולי ביצועים).

כל הקצאה מסוג "התאמה לא מדוייקת" מה-largebin מנוצלת במלואה (exhausted) או מפוצלת (remaindered), אך בכל מקרה – שדה ה-last_remainder לא יעודכן (ככל הנראה משיקולי ביצועים)

Binmap

זהו וקטור שמייצג בצורה לא הדוקה (loosely) אילו מבין ה-smallbins וה-largebins (של ה-arena בה הוא שמור) מאוכלסים (לא ריקים). נמצא בשימוש ע"י malloc כדי למצוא במהירות את ה-bin המאוכלס הגדול ביותר הבא כאשר לא התאפשר לתת מענה לבקשה מתוך ה-bin המתאים לה.

חיפושים ב-binmap מתרחשים לאחר חיפוש לא מוצלח ב-unsortedbin או ב-largebin, כתלות בגודל הבקשה. Malloc מוצא את ה-bin המאוכלס הגדול ביותר הבא ומנצל או מחלק את ה-chunk האחרון באותו bin. במקרה של פעולת remaindering, במידה וגודל הבקשה בטווח של ה-smallbin, חלק השארית "מפורסם" כ-last_remainder.

Bin יסומן כמאוכלס כאשר chunk ממויין לתוכו במהלך סריקה של ה-unsortedbin. Bin יסומן כריק כאשר חיפוש binmap מוצא bin ריק שסומן כמאוכלס (סוג של תיקון שגיאות זיכרון)

next

רשימה מקושרת חד-כיוונית מעגלית של כל ה-Arenas ששייכים לתהליך זה

next_free

רשימה מקושרת חד כיוונית לא מעגלית של free arenas (arenas שלא מצומדים ל-threads). ראש רשימה זו הוא ה-free_list symbol

attached_threads

מספר ה-threads שעושים שימוש במקביל ב-arena זה.

system_mem

סך כל הזיכרון הניתן לכתיבה שממופה כרגע ע"י arena זה.

max_system_mem

הכמות המקסימלית של זיכרון ניתן לכתיבה ש-arena זה מיפה אי פעם. בשימוש ע"י calloc() כדי לקבוע האם זיכרון heap ממופה "טרי" צריך להיות מאופס (בתוך calloc()) או לא.

Remaindering

זהו המונח בו malloc עושה שימוש עבור פעולת פיצול של free chunk לשני chunks קטנים יותר, ולאחר מכן הקצאה של ה-chunk המתאים עבור הבקשה. ה-chunk הנותר מקושר אל ה-unsortedbin השייך ל-arena של אותו chunk.

לדוגמה, במהלך בקשה ל-chunk בגודל 0x100, אם ה-arena של ה-thread יכול להציע רק chunk בגודל 0x300, malloc יבצע unlink של ה-chunk בגודל 0x300 מה-free list בה הוא נמצא, יגזור ממנו chunk בגודל 0x100 ואת ה-chunk הנותר (remainder) בגודל 0x200 יקשר אל תוך ראש ה-unsortedbin ויקצה עבור התוכנית את ה-chunk עם הגודל 0x100. (שימו לב שהמיקום בזיכרון של ה-chunk השלם לאחר פעולת הפיצול לא משתנה, אלא רק לאן כל אחד מהחלקים משויך או לא משויך (אין מעקב אחרי allocated chunks ! ,המצביעים נשמרים במקום הניתן לכתיבה בזיכרון בצורה של מצביעים הנשמרים למשתנים היושבים לדוגמה על ה-stack, ה-heap או ה-bss ע"י כותב התוכנית, ואם הוא מאבד גישה אליהם, תהיה לנו זליגת זיכרון.)

תהליך זה עשוי להתרחש באחת מתוך שלוש מקומות בתרשים-זרימה המתאר את האלגוריתם של malloc() (מופיע בהמשך): 1.במהלך הקצאות מתוך ה-largebins 2.במהלך חיפוש binmap 3.מתוך last_remainder במהלך unsortedbin scan.

Exhausting

במקרה שבו thread מבקש chunk בגודל 0x80, וב-arena שלו יש רק chunk בגודל 0x90, malloc ימצה ויקצה באופן מלא את כל ה-chunk הזה במקום לפצל אותו. זאת משום שאין בו מספיק זיכרון כך שיוותר לאחר פיצול (לקיחת 0x80 בתים ממנו) שארית של chunk בגודל מינימלי (0x20)

Unlinking

במהלך פעולות הקצאה ושחרור, chunks עשויים להיות מוצאים מה-free list שבה הם שוכנים, ה-chunk המוצא החוצה נקרא chunk "victim" בתוך קוד המקור של malloc (קורבן נבחר, בדומה לשימוש במילה victim באלגוריתם page replacement של מע"ה או כאשר יש צורך להכניס נתון לתוך cache line מלא – "evicting from cache" דרך אגב - לעיתים ישנו גם מבנה חומרתי/תוכנתי ששומר את אותם קורבנות דוגמת בלוקים, דפים וכו' כדי לתת להם הזדמנות שניה לשימוש בהם ובכך לנסות לחסוך בפעולות IO מיותרות). מידע נוסף על ה-free lists נמצא תחת הכותרת Arenas.

ישנן כמה דרכים בהן מתבצע Unlinking:

Fastbin & Tcache Unlink

ה-Fastbins וה-Tcache bins משתמשים ברשימות מקושרות חד-כיווניות בשיטת LIFO. ניתוק של chunks מרשימות אלו מלווה בהעתקה אחת בלבד של שדה ה-fd של ה-victim chunk אל תוך ראש הרשימה. מידע נוסף על ה-fastbins נמצא תחת הכותרת Arenas, וכמו כן מידע נוסף על ה-Tcache מופיע תחת הכותרת Tcache.

Partial Unlink

מתרחש כאשר chunk מוקצה מתוך unsortedbin או smallbin. ה-bk של ה-victim chunk (שהרי נלקח מסוף הרשימה) נקרא (נעקב) והכתובת של ראש ה-bin מועתקת לתוך ה-fd של ה destination chunk. בנוסף לכך, ה-bk של ה-vitcim מועתק לתוך שדה ה-bk של ראש ה-bin (תזכורת: זהו לא ראש הרשימה אלא chunk שלא מכיל user data ומקשר בין ראש וזנב הרשימה אבל. זה אותו struct מסוג בשם malloc_chunk). מידע נוסף על ה-unsortedbin וה-smallbin נמצא תחת Arenas.

Full Unlink

מתרחש כאשר chunk מאוחד אל תוך free chunk אחר. בנוסף לכך, מתרחש גם כאשר chunk מוקצה מתוך ה-largebins או ע"י חיפוש binmap. ה-fd של ה-victim chunk נעקב וה-bk של ה-victim מועתק (הופך להיות) אל ה-bk של ה-destination. לאחר מכן ה-bk של ה-victim נעקב וה-fd של ה-vicim הופך להיות ה-fd של ה-destination (כלומר- במקרה של איחוד, קדמי או אחורי, כתובת ה-chunk החדש תמיד תהיה זו של ה-chunk שהיה עם כתובת נמוכה יותר)

<u>Malloc Parameters</u>:

מבנה המחזיק משתנים המכתיבים כיצד malloc פועל, מוגדר בתוך המבנה malloc_par :

```
 struct malloc_par
{
  /* Tunable parameters */
```

```
  unsigned long trim_threshold;
  INTERNAL_SIZE_T top_pad;
  INTERNAL_SIZE_T mmap_threshold;
  INTERNAL_SIZE_T arena_test;
  INTERNAL_SIZE_T arena_max;
  /* Memory map support */
  int n_mmaps;
  int n_mmaps_max;
  int max_n_mmaps;
  /* the mmap threshold is dynamic, until the user sets
     it manually, at which point we need to disable any
     dynamic behavior. */
  int no_dyn_threshold;
  /* Statistics */
  INTERNAL_SIZE_T mmapped_mem;
  INTERNAL_SIZE_T max_mmapped_mem;
  /* First address handed out by MORECORE/sbrk.  */
  char *sbrk_base;
#if USE_TCACHE
  /* Maximum number of buckets to use.  */
  size_t tcache_bins;
  size_t tcache_max_bytes;
  /* Maximum number of chunks in each bucket.  */
  size_t tcache_count;
  /* Maximum number of chunks to remove from the unsorted list, which
     aren't used to prefill the cache.  */
  size_t tcache_unsorted_limit;
#endif
};
```

:Tcache (Thread cache)

בגרסאות GLIBC >= 2.26 , לכל thread מוקצה מבנה משלו בשם tcache. Tcache מתנהג כמו arena, אבל שלא כמו arenas רגילים – tcaches לא משותפים בין threads (ומכאן שנחסך הצורך בנעילת מנעול של ה-heap לצרכי סינכרון בין threads בעת בקשות להקצאה ושחרור זיכרון מתוך ה-tcache. ובכך מושג שיפור כללי בביצועים). הם נוצרים ע"י הקצאת מקום על ה-heap (בתחילתו) השייך ל-arena של אותו thread ומשוחררים כאשר ה-thread מסתיים (exits). המטרה של ה-tcache היא להפיג את התחרות בין ה-threads על המשאבים של malloc ע"י כך שניתן לכל thread אוסף משלו של chunks שאינם משותפים עם threads אחרים שמתשמשים באותו arena.

ה-tcache מוגדר בתוך tcache_perthread_struct בדומה לאיור 11, אשר מחזיק את 64 ה-tcachbins כשאחריהם מערך מונים העוקב אחרי מספר ה-free chunks בכל tcachebin.

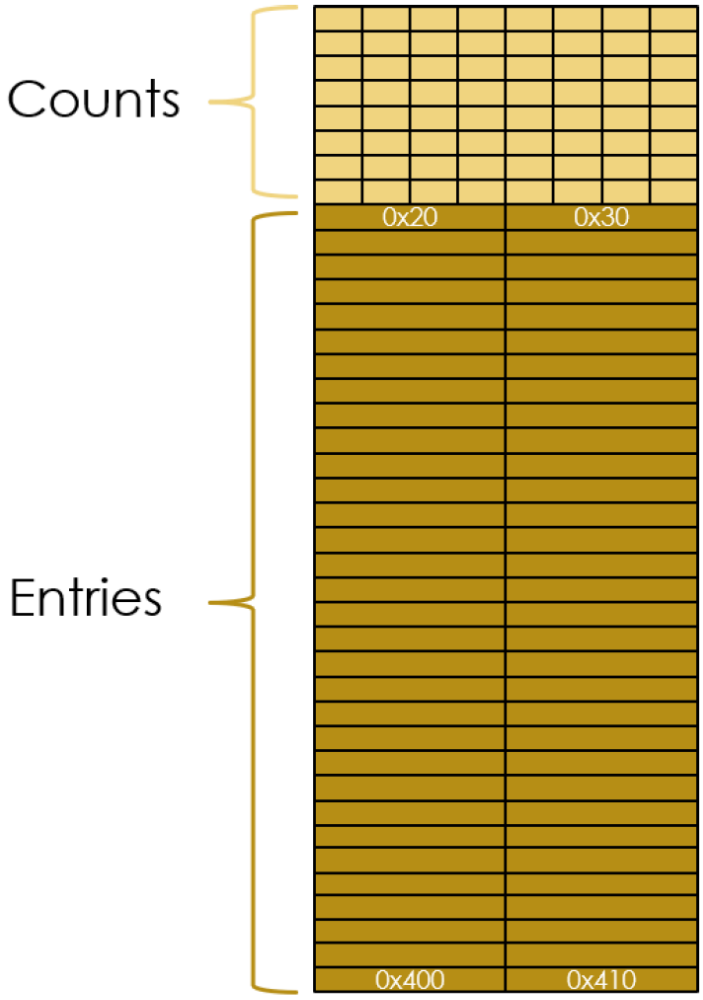


Figure 11: A tcache_perthread struct

איור 11 מבנה ה-tcache הנמצא בתחילת ה-heap של thread מסויים. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

שימו לב שבאיור זה המונים מיוצגים על ידי מערך של words (word = 2 בתים), זהו המצב רק עבור גרסאות GLIBC >= 2.30, בגרסאות ישנות יותר זה היה מערך של chars (בית אחד). תחת תנאי ברירת מחדל tcache מחזיק chunks בגדלים בטווח 0x20 – 0x410 כולל. אותם tcachebins מתנהגים בצורה דומה ל-fastbins, כשכל אחד מהם מתפקד כראש רשימה מקושרת חד-כיוונית לא מעגלית של free-chunks בגודל מסויים. הכניסה/רשומה הראשונה במערך המונים עוקבת אחר מספר ה-free chunks המקושרים אל ה- 0x20 tcachebin, הכניסה השניה עושה אותו דבר עבור ה-tcachebin 0x30 וכו'.

תחת תנאי ברירת-מחדל, יש מגבלה (מכסה) על כמות ה-free chunks ש-tcachebin יכול להחזיק. ערך זה שמור בתוך המבנה malloc_par תחת השדה (member) tcache_count. כאשר מונה של tcachebin מגיע לסף זה, free chunks של המתאימים לאותו bin (באותו גודל) יקבלו יחס כאילו ה-tcache איננו קיים.

לדוגמה: אם ה-tcachbin 0x20 מלא (מחזיק 7 free chunks) אז ה-chunk הבא בגודל 0x20 שישוחרר יקושר לתוך ה-fastbin 0x20. Malloc עושה שימוש במונים הנ"ל כדי לקבוע האם bin מסויים מלא.

הקצאות מה-tcache של thread מקבלות עדיפות על פני ה-arena שלו, פעולה זו מבוצעת מתוך הפונקציה __libc_malloc() ולא מתבצעת כניסה לתוך _int_malloc() . Chunks שמשוחררים וגודלם בטווח של ה-tcache מקושרים אל תוך ה-tcache של אותו thread אלא אם tcachebin היעד מלא, ובמקרה שכזה ה-arena של אותו thread ישרת את הבקשה. שימו לב שכניסות ב-tcache משתמשות במצביעים ל-user data ולא ל-metadata של ה-chunk.

Tcache Dumping/Stashing

בגרסאות של GLIBC שמקומפלות עם תמיכה ב-tcache, chunks בטווח גדלי ה-tcache מוטלים אל ה-tcache כאשר thread מקבל הקצאה מה-arena שלו. כאשר chunk מוקצה מתוך ה-fastbins או ה-smallbins, malloc מטיל את כל ה-free chunks שנותרו באותו bin אל תוך ה-tcachebin המתאים להם עד שהוא מלא (מכאן השימוש במילה stashing, שפירושה החבאה/הסתרה כלפי חוץ) כפי שניתן לראות באיור 12

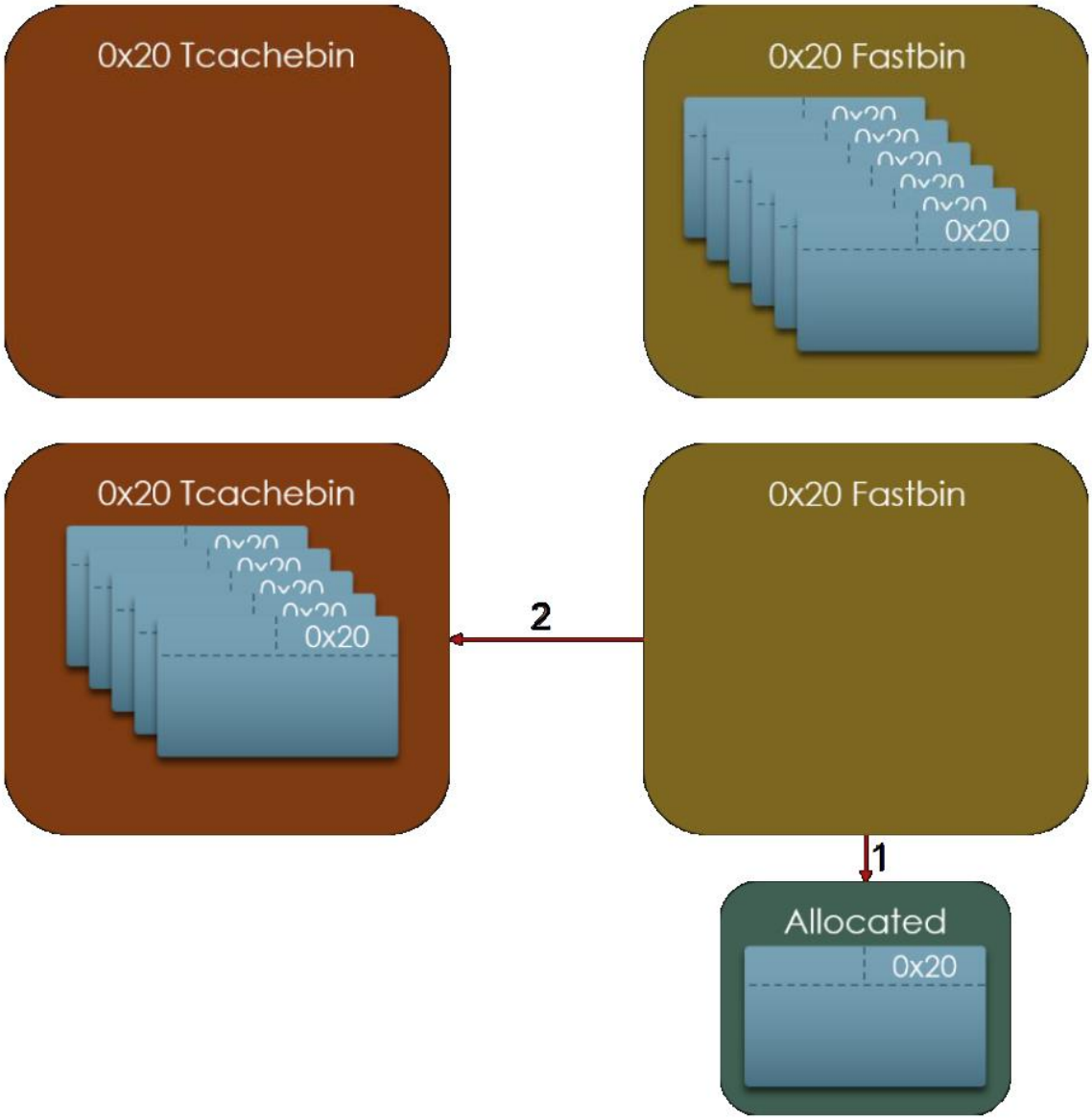


Figure 12: Tcache dumping from a fastbin

איור 12 ריקון ה-chunk הנותרים של fastbin אל תוך ה-Tcache לאחר הקצאה ממנו. מתוך HeapLAB , Heap Exploitation Bible, Max Kamper באישור היוצר

כאשר מתרחשת סריקה של ה-unsortedbin (במטרה למצוא free chunk מתאים להקצאה), malloc מטיל כל chunk שהוא מוצא עם גודל מתאים בדיוק לגודל הבקשה אל ה-tcache המתאים לו. אם ה-tcachebin המיועד מלא ו-malloc מוצא chunk שמתאים בדיוק בגודל בתוך ה-unsortedbin, אז chunk זה יוקצה. אם הסריקה הנ"ל מסתיימת ואחד או יותר מה-chunks הוטלו לתוך ה-tcachebin, אז יוקצה chunk מתוך אותו tcachebin.

הפונקציות של Malloc האחראיות להקצאה ושחרור

void* malloc(size_t bytes)

פונקציית הקצאת הזיכרון הדינמי של GLIBC – מקבלת גודל בקשה בבתים כארגומנט יחיד ומחזירה מצביע לאזור לא מאותחל של ה-user data של chunk בגודל מתאים מזכרון ה-heap. סימבול ה-malloc (imported symbols) הינו alias (שם נוסף) ל ()__libc_malloc , וזו מהווה פונ' מעטפת מסביב ל-()_int_malloc שבה נמצא רוב קוד ההקצאה.

void* realloc(void* oldmem, size_t bytes)

מספקת מספיק זיכרון דינמי כדי להחזיק bytes בתים של מידע. Oldmem הוא מצביע שמסופק במקור ע"י אחת מפונקציות ההקצאה. במהלך פעולה זו עשוי להיות מוקצה chunk חדש, העתקה של המידע מה-oldmem chunk, שחרור של oldmem והחזרת chunk חדש מוקצה. ()Realloc משתמשת בכמה אופטימיזציות כדי לוודא שזה נעשה ביעילות, למשל ע"י מיזוג קדימה עם free chunk כדי להימנע מפעולת ההעתקה. כאשר bytes == 0 אנו מקבלים בעקיפין פעולת ()free ! .

Void free (void* mem)

פונקציית מחזור הזיכרון הדינמי של GLIBC – מקבלת לשטח זיכרון שבמקור מסופק ע"י אחת מפונקציות ההקאצה של malloc וממחזרת אותו. הסימבול free הוא alias ל ()__libc_free , אשר בתורה מהווה מעטפת מסביב ל ()_int_free – היכן שנמצא מרבית הקוד האחראי על מחזור הזיכרון.

Malloc Hooks

GLIBC מספקת hooks (אמצעי להרחבת ההתנהגות של תוכנית בזמן ריצה, כלומר אמצעי שנועד לשרת את המפתח אך אנחנו נשתמש בו ל"צרכינו האישיים") עבור חלק מפונקציונליות הליבה של malloc. שימושים טיפוסיים ב-hooks אלו כוללים ניטור סטטיסטיקות זיכרון דינאמי או מימוש של מקצה זיכרון שונה לגמריי. עקב כך שהם נשארים ניתן לכתיבה במהלך מחזור חיי התוכנית, הם מהווים מטרה מעשית (viable) עבור heap exploits המנסים להשיג יכולת הרצת קוד. GLIBC מספקת את ההוקים הבאים בהקשר של malloc:

- __after_morecore_hook
- __free_hook
- __malloc_hook
- __malloc_initialize_hook
- __memalign_hook
- __realloc_hook

חלק מהוקים אלו מאוכלסים ע"י ערכי אתחול המאופסים (ל-NULL) לאחר הקריאה הראשונה לפונקציה. לדוגמה: __malloc_hook מאוכלס ע"י הכתובת של הפונקציה ()mallok_hook_ini במהלך האתחול של GLIBC אשר מאפסת את __malloc_hook וקוראת ל-()ptmalloc_init

```
static void* malloc_hook_ini (size_t sz, const void* caller) {
        __malloc_hook = NULL;
        ptmalloc_init();
        return __libc_malloc(sz);
}
```

כאשר hook מאופס, קריאות לפונקציית האב שלו הולכות ישירות אל אותה פונקציה (האב). כאשר hook מאוכלס, ריצת התוכנית ממשיכה (execution redirection) מהכתובת המוצבעת ע"י ה-Hook כאשר פונקציית האב נקראת (מזכיר קצת PLT/GOT hooking). לדוגמה, השורות הראשונות של __libc_malloc() נראות כך:

```
void* (*hook) (size_t, const void*) = atomic_forced_read __malloc_hook);
if __builtin_expect(hook != NULL, 0))
      return(*hook)(bytes, RETURN_ADDRESS(0));
```

(המשתנה hook הוא מצביע לפונקציה)

החל מגרסה 2.34 של GLIBC (העדכנית ביותר נכון לפרסום המאמר) הוסרו מרבית ה-hooks הנ"ל מה-API. בין השאר (אבל לא רק) בגלל סיבות אבטחה.

" The deprecated memory allocation **hooks __malloc_hook, __realloc_hook, __memalign_hook and __free_hook** are now removed from the API. (...) These hooks no longer have any effect on glibc functionality. (...)

The __morecore and __after_morecore_hook malloc hooks and the default implementation __default_morecore have been removed from the API. Existing applications will continue to link against these symbols but the interfaces no longer have any effect on malloc. "

Mitigations (מנגנוני הגנה בקוד של GLIBC malloc)

ראשית, נבהיר כי כאשר הפצה של לינוקס, הקבצים הבינארים של GLIBC שלה חשובים. ייתכן ונראה שתי גרסאות של אותה ההפצה עם אותו מספר גרסה של GLIBC אך עם הגנות (mitigations) שונות. הסיבה לכך יכולה להיות בגלל שימוש ב-release branch בניגוד ל-master branch, וקימפול של הקבצים הבינארים של GLIBC שלהם פעם אחת לפני שבוצע backporting (patching) של גרסה ישנה עם תכונות מגרסה עדכנית יותר) ומאוחר יותר לאחר שבוצע backporting. הפצת Fedora 27 היא דוגמה מצויינת לכך. לכן, ייתכן שתיתקלו בגרסה של libc עם mitigations שהוצגו לראשונה בגרסה הממוספרת גבוה יותר.

סקירה היסטורית **חלקית** של הגנות בקוד מפני אקספלוטציית heap שהוכנסו ל-GLIBC לאורך השנים:

[מקור: HeapLAB - GLIBC Heap Exploitation Bible by Max Kamper]

Commit date	Published in GLIBC version	Author	Description	Diff
19/08/2003	2.3.3	Ulrich Drepper	Ensure chunks don't wrap around memory on free().	diff
21/08/2004	2.3.4	Ulrich Drepper	Safe unlinking checks.	diff
09/09/2004	2.3.4	Ulrich Drepper	Check that the chunk being freed is not the top chunk. Check the next chunk on free is not beyond the bounds of the heap. Check that the next chunk has its prev_inuse bit set before free.	diff
19/11/2004	2.3.4	Ulrich Drepper	Check next chunk's size sanity on free().	diff
20/11/2004	2.3.4	Ulrich Drepper	Check chunk about to be returned from fastbin is the correct size. Check that the chunk about to be returned from the unsorted bin has a sane size.	diff
22/12/2004	2.3.4	Ulrich Drepper	Ensure a chunk is aligned on free().	diff
13/10/2005	2.4	Ulrich Drepper	Check chunk is at least MINSIZE bytes on free().	diff
30/04/2007	2.6	Ulrich Drepper	Unsafe unlink checks for largebins.	diff
19/06/2009	2.11	Ulrich Drepper	Check if bck->fd != victim when allocating from a smallbin. Check if fwd->bk != bck before adding a chunk to the unsorted bin whilst remaindering an allocation from a large bin. Check if fwd->bk != bck before adding a chunk to the unsorted bin whilst remaindering an allocation from a binmap search. Check if fwd->bk != bck when freeing a chunk directly into the unsorted bin.	diff
03/04/2010	2.12	Ulrich Drepper	When freeing a chunk directly into a fastbin, check that the chunk at the top of the fastbin is the correct size for that bin.	diff
17/03/2017	2.26	DJ Delorie	Size vs prev_size check in unlink macro.	diff
30/08/2017	2.27	Florian Weimer	Don't backtrace on abort anymore.	diff

30/11/2017	2.27	Arjun Shankar	Fix integer overflow when allocating from the tcache.	diff
12/01/2018	2.27	Istvan Kurucsai	Fastbin size check in malloc_consolidate.	diff
14/04/2018	2.28	DJ Delorie	Check if bck->fd != victim when removing a chunk from the unsorted bin during unsorted bin iteration.	diff
16/08/2018	2.29	Pochang Chen	Check top chunk size field sanity in use_top.	diff
17/08/2018	2.29	Moritz Eckert	Proper size vs prev_size check before unlink() in backward consolidation via free. Same check in malloc_consolidate().	diff
17/08/2018	2.29	Istvan Kurucsai	When iterating unsorted bin check: size sanity of next chunk on heap to removed chunk, next chunk on heap prev_size matches size of chunk being removed, check bck->fd != victim and victim->fd != unsorted_chunks (av) for chunk being removed, check prev_inuse is not set on next chunk on heap to chunk being removed.	diff
20/11/2018	2.29	DJ Delorie	Tcache double-free check.	diff
26/11/2018	2.29	Florian Weimer	Validate tc_idx before checking for tcache double-frees.	diff
14/03/2019	2.30	Adam Maris	Check for largebin list corruption when sorting into a largebin.	diff
18/04/2019	2.30	Adhemerval Zanella	Request sizes cannot exceed PTRDIFF_MAX (0x7fffffffffffffff)	diff

תרשים זרימה של Malloc:

להלן תרשים זרימה מפושט של האלגוריתם המתאר את מסלולי הקוד ש-malloc() יכול לקחת. בלוקים בצבע בצהוב רלוונטיים רק לגרסאות של GLIBC המהודרות עם תמיכה ב-tcache, בלוקים סגולים מכילים סדרה של פעולות המוצגות בתרשים זרימה משלהן. בלוקים ירוקים מציינים הקצאה מוצלחת וחזרה מ-malloc(). נק' הכניסה היא הבלוק העליון השמאלי.

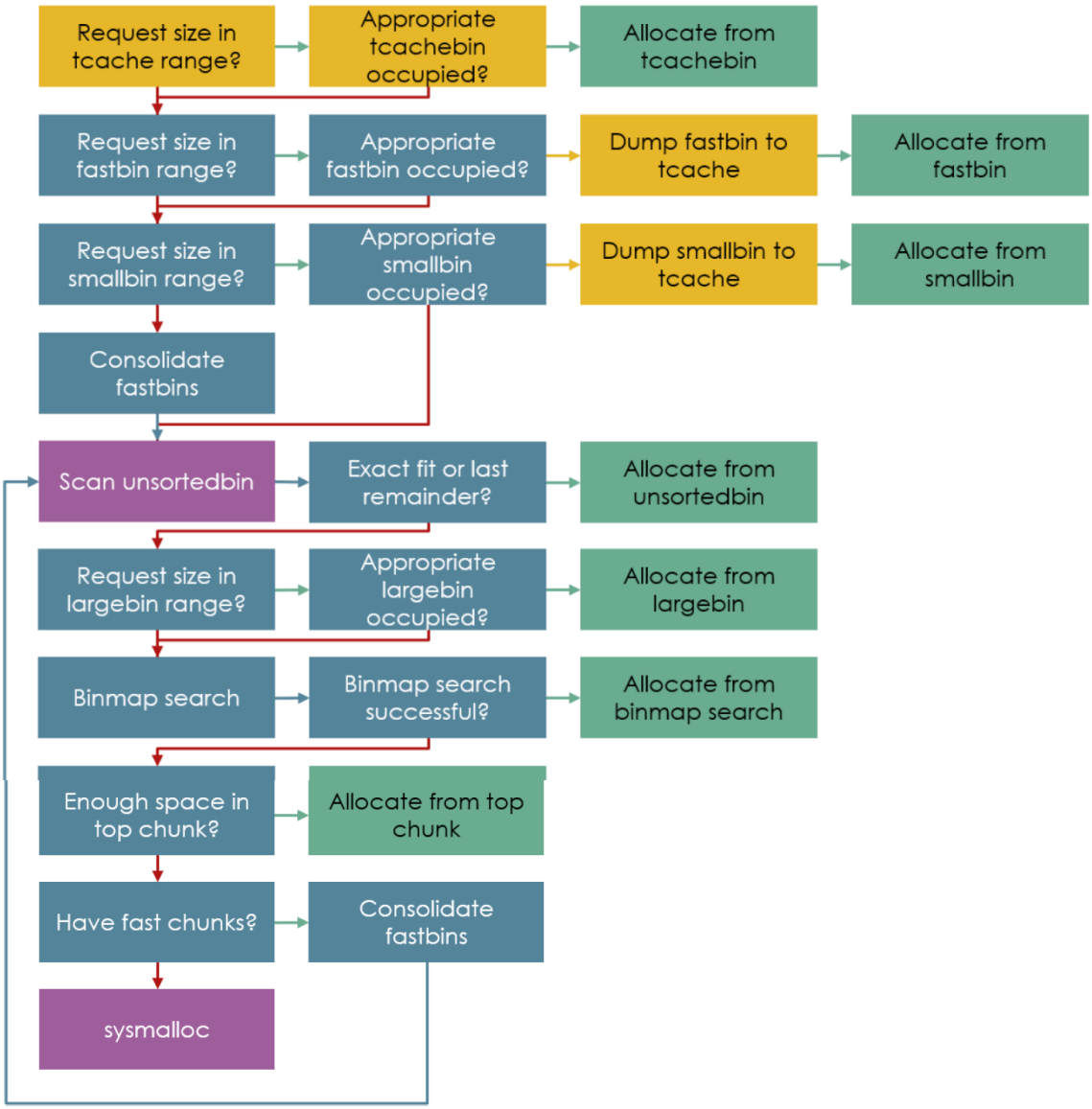


Figure 13: Malloc flowchart

איור 13 תרשים זרימה של malloc(). מתוך HeapLAB Heap Exploitation Bible, Max Kamper באישור היוצר

תרשים זרימה של Unsortedbin Scan:

להלן תרשים זרימה המתאר את מסלולי הקוד שסריקת ה-Unsortedbin יכול לקחת. בלוקים בצבע בצהוב רלוונטיים רק לגרסאות של GLIBC המהודרות עם תמיכה ב-tcache, בלוקים ירוקים מציינים הקצאה מוצלחת וחזרה מ-malloc(). אם ריצת התוכנית מגיעה לבלוק אדום – malloc() ממשיך מה-unsortedbin scan כמתואר בתרשים של malloc(). נק' הכניסה היא הבלוק העליון השמאלי.

שימו לב שה-unsortedbin מטפל ב-tcache dumping בצורה שונה לעומת ה-fastbins וה-smallbins. Chunks עם התאמה מדוייקת בגודלם מאוחסנים מיידית ב-tcache. אלא אם tcachebin היעד מלא או שה-victim chunk נמצא מחוץ לטווח גדלי ה-tcache, יוקצה chunk מ-tcachebin היעד בעת שה-unsortedbin ריק.

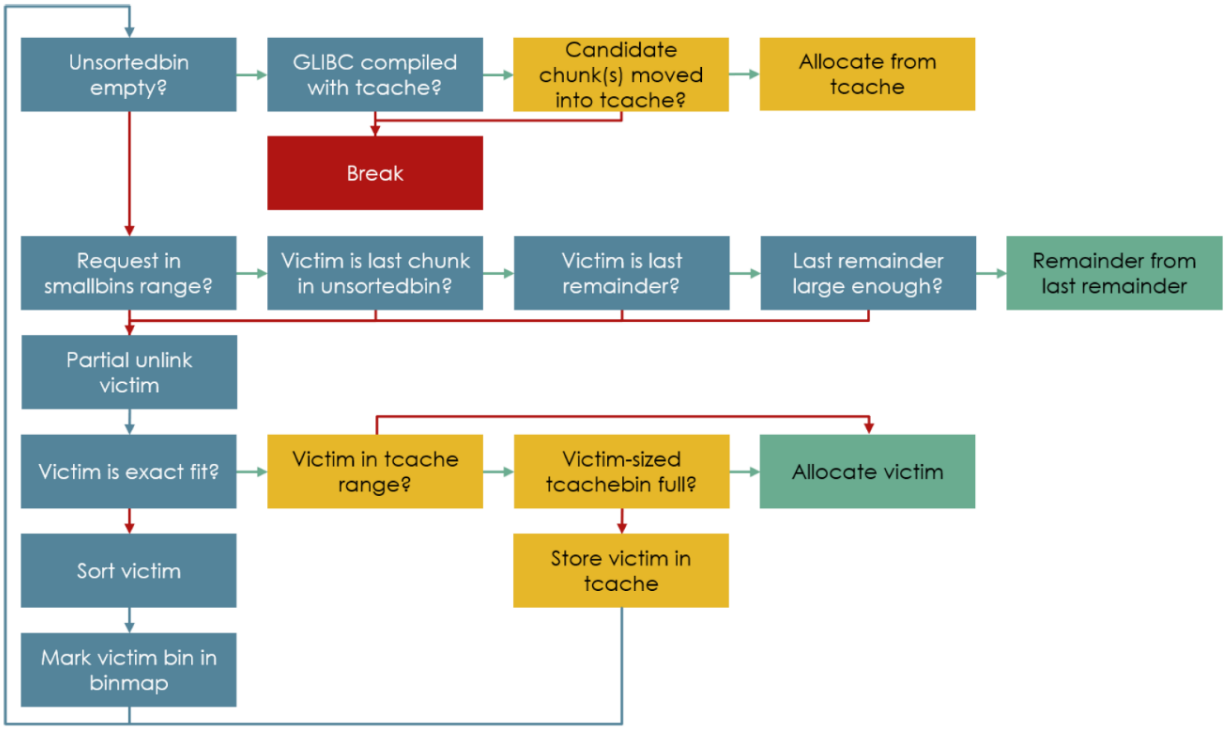


Figure 14: Unsortedbin flowchart

איור 14 תרשים זרימה של הקצאה מן ה-Unsortedbin. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

תרשים זרימה של Sysmalloc:

להלן תרשים זרימה מפושט של האלגוריתם המתאר את מסלולי הקוד ש-()sysmalloc יכול לקחת. בלוקים ירוקים מציינים הקצאה מוצלחת וחזרה מ-()sysmalloc. במידה וריצת התוכנית מגיעה לבלוק אדום, מספר השגיאה ENOMEM נשמר לתוך המשתנה הגלובלי errno ו-()sysmalloc מחזירה 0. נק' הכניסה היא הבלוק העליון השמאלי.

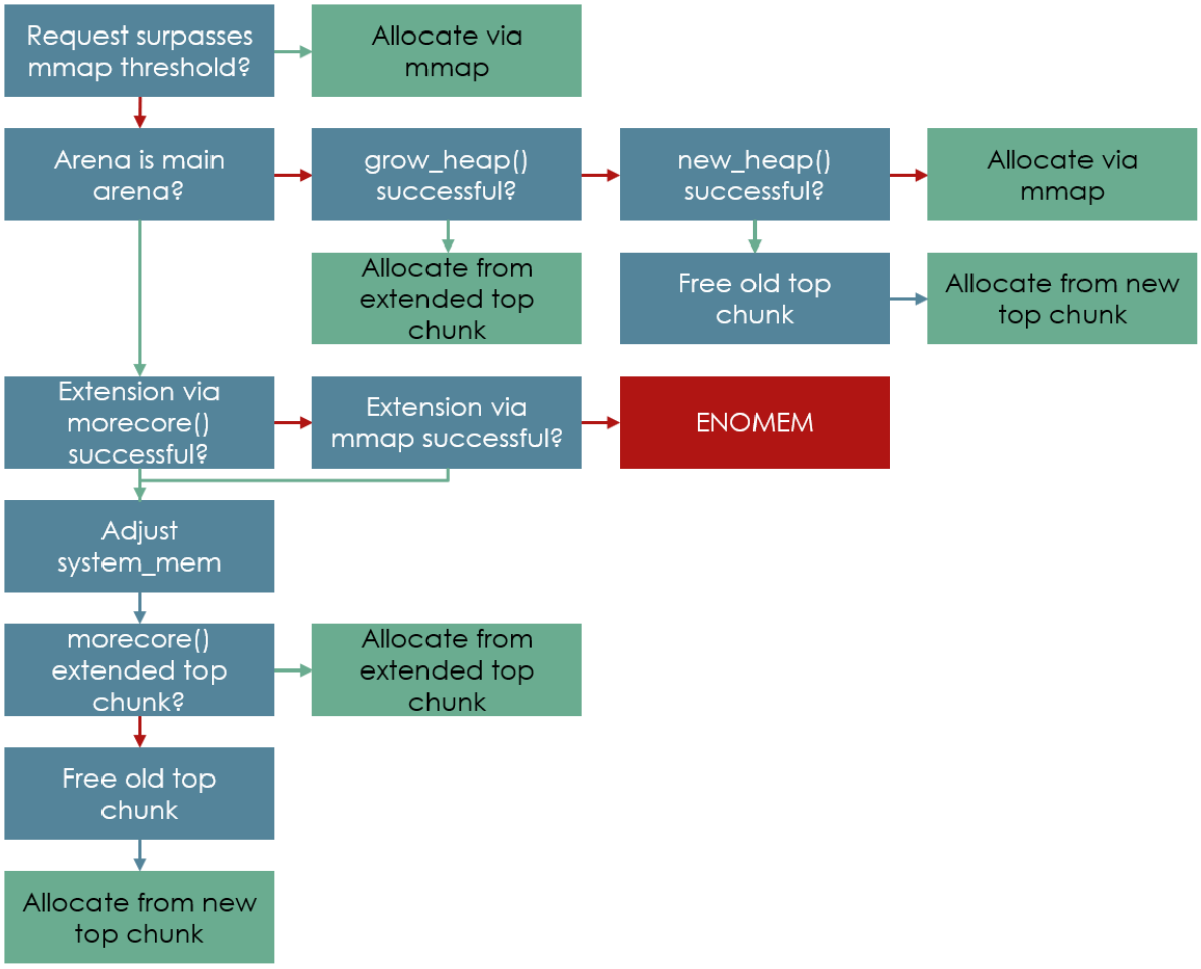


Figure 15: Sysmalloc flowchart

איור 15 תרשים זרימה של ()Sysmalloc. מתוך HeapLAB Heap Exploitation Bible, Max Kamper, באישור היוצר

תרשים זרימה של Free

להלן תרשים זרימה מפושט של האלגוריתם המתאר את מסלולי הקוד ש-()free יכול לקחת. בלוקים ירוקים מציינים "מסלול חזרה" (מסתיים ב-return). בלוקים בצבע בצהוב רלוונטיים רק לגרסאות של GLIBC המהודרות עם תמיכה ב-tcache.

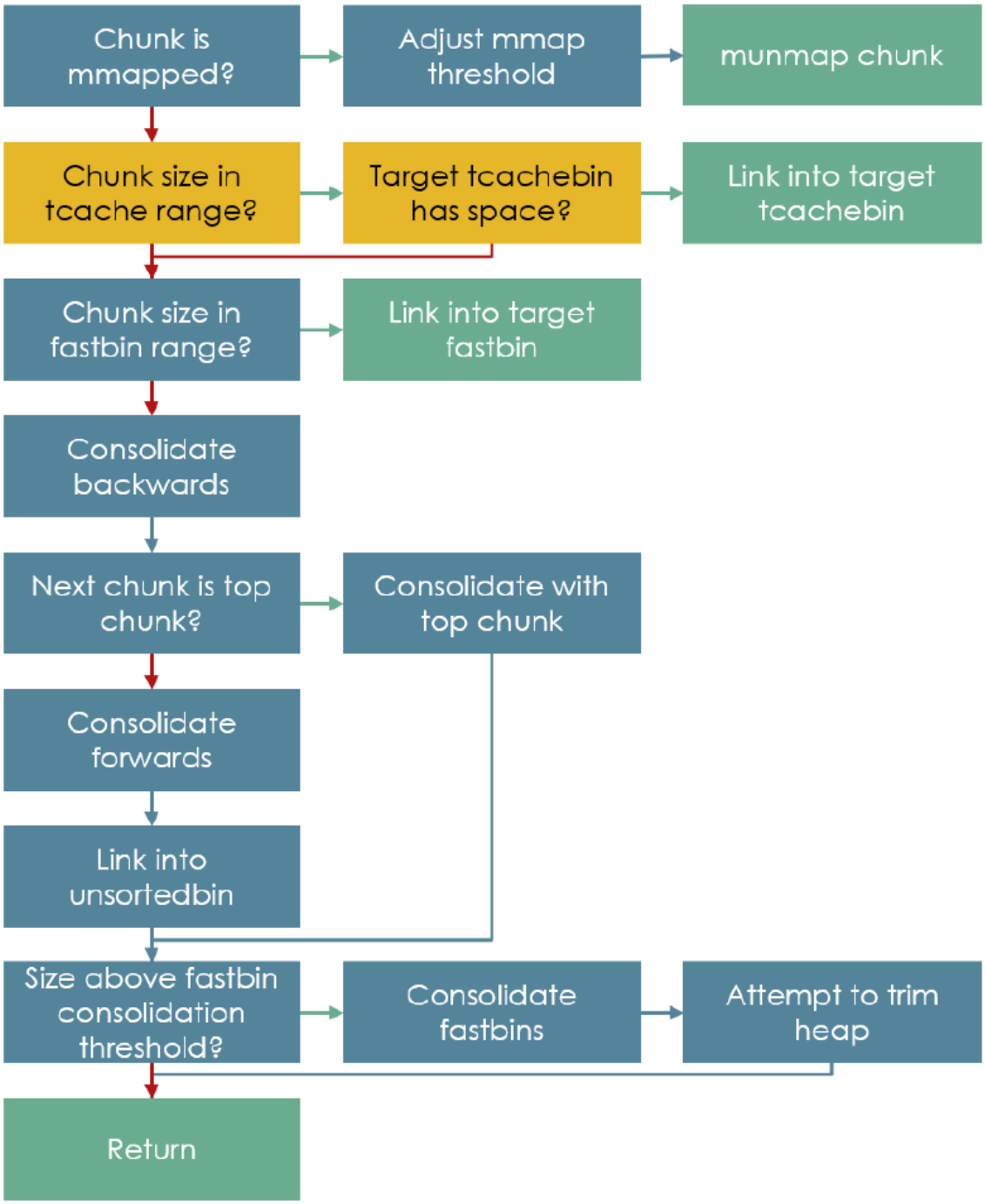


Figure 16: Free flowchart

טכניקות ניצול חולשות/השמשה מפורסמות:

חלק זה מהווה הרחבה לקורס ההכשרה HeapLAB והתרגול המעשי בסביבת Ubuntu Linux. הוא איננו מיועד להיות הסבר ממצה על כל טכניקת heap exploitation (למעשה, כל שלב בטכניקה הינו קריטי להצלחתה ונשען על הבנה של המימוש בתוך GLIBC)

House of Force

סקירה כללית

דריסה של שדה הגודל של ה-top chunk עם ערך גדול מאוד, לאחר מכן בקשה של מספיק זיכרון כדי לגשר על הפער (מרחק בזיכרון) בין ה-top chunk וה-target data (הכתובת אליה אנו מעוניינים לכתוב). הקצאות שנעשות בצורה זו יכולות להקיף מסביב את מרחב הזיכרון הוירטואלי! (to wrap around), ובכך לאפשר לטכניקה זו לטרגט זיכרון בכתובת נמוכה יותר מזו של ה-heap.

פרטים

בגרסאות GLIBC לפני 2.29, שדה הגודל של ה-top chunk לא נתון לשום בדיקות תקינות (integrity) במהלך הקצאות!. אם גודל ה-top chunk נדרס ע"י overflow למשל ומוחלף בערך גדול מאוד, הקצאות עוקבות מתוך ה-top chunk (לא ע"י mmap()) יכולות להקיף את הזיכרון שבשימוש (להגיע עד סוף מרחב הכתובות הוירטואלי ואז להמשיך מתחילתו). הקצאות גדולות מאוד מ-top chunk שהושחת יכולות לגרום להקפה שכזו בגרסאות GLIBC < 2.30. [בעצם הצלחנו להשיג write primitive לכל כתובת שנרצה (arbitrary) כל עוד המרחק אליה בעת הקפת מרחב הכתובות לא גדול מדי ביחס לערך ה-size field של ה-top chunk]

לדוגמה, top chunk שמתחיל בכתובת 0x405000 ויעד לכתיבה שנמצא בכתובת 0x404000 בתוך אזור ה-data. של התוכנית שאני חייבים לכתוב אליו. נדרוס את שדה הגודל של ה-top chunk בעזרת באג, נחליף את ערכו ל-0xffffffffffffffff1 , לאחר מכן נחשב את כמות הבתים הדרושים כדי להזיז (קדימה) את ה- top chunk לכתובת שנמצאת בדיוק לפני המטרה. לכן בסה"כ 0x405000 - 0xffffffffffffffff בתים כדי להגיע לסוף מרחב הכתובות הוירטואליות (VA), בתוספת של 0x404000 בתים ולבסוף החסרה של 0x20 בתים כדי לעצור בדיוק 16 בתים לפני כתובת המטרה.

למה מחסירים 0x20?דרך אחת להסתכל על זה היא שקודם כל צריך לחזור אחורה 0x10 בתים כדי להגיע לתחילת ה-chunk שמכיל את ה-target's data, כי לשם אנו רוצים שיזוז ה-top chunk, כדי שבהקצאה הבאה לאחר מכן נוכל לכתוב ליעד. הסיבה שאנו מעוניינים בשתי הקצאות היא שכתיבת נתונים לאורך מסלול ההקפה כמעט בטוח תנסה לכתוב לאזורים ללא הרשאות כתיבה ונקבל segfault, שנית – צריך

להפחית עוד 0x8 בתים מהתוצאה הראשונה והשניה של החישוב בגלל ה-size field שמתווסף. (כלל אצבע: אם רוצים chunk בגודל מסויים, נבקש 8 בתים פחות מגודל זה. כמובן שיש עוד ערכים קטנים יותר שיגרמו להקצאה של אותה כמות בתים).

יש כמה דרכים אפשריות להסתכל על חישובי גודל בקשה ורצוי לצייר דוגמה קטנה של למשל 3 chunk-ים להמחשה ואם לא מצליחים- לפעול בשיטת ניסוי וטעייה. האמת היא שניתן להשתמש בכל ערך בין 0x17 ל-0x26 כדי שזה יעבוד – malloc מעגל כלפי מעלה את גודל הבקשה לגודל הבלוק הקרוב ביותר. בסביבת production ("על אמת") ייתכן ולא נצליח לבקש את הכמות המדוייקת של בתים שיתיישרו בצורה מושלמת כי ה-target data עשוי להיות לא מיושר ל-16 בתים או שיכולת הכתיבה שלנו גם לא תהיה מיושרת. על כן אפשר לנסות לקרב את ה-top chunk כמה שיותר ליעד שלנו ומשם לבצע הקצאות נוספות.

לאחר שבקשה זו טופלה מתוך ה-top chunk, ה-chunk הבא שנקצה יוגש גם הוא ממנו ויחפוף בדיוק את נקודת היעד שרצינו לכתוב אליה.

שימושים נוספים

במקרה בו היעד נמצא באותו ה-heap שבו נמצא ה-top chunk, הזלגה (leaking) של כתובת מה-heap לא נחוצה (אילוץ אחד פחות עבור תוקף), ההקצאה יכולה להקיף מסביב את מרחב הכתובות הוירטואלי בחזרה אל אותו ה-heap לכתובת יחסית ל-top chunk (אם מציירים את זה, קל לשים לב שהערך שיש לבקש תלוי רק בהפרש בין כתובת ה-chunk הנוכחי לכתובת היעד! החישוב יהיה מהצורה: 0xffffffff – delta(x,target) – 0x20 () במערכת עם מרחב כתובות של 64 ביט. והרי שהמרחק delta(x,target) לא מושפע מ-ASLR, שניהם זזים באותה מידה ביחד עם כתובת הבסיס של ה-heap. כך שבהפרש ביניהם ההזזות יתקזזו)

ה-malloc hook הוא מטרה מעשית עבור טכניקה זו משום שהיכולת להעביר גודל בקשה גדול כרצוננו (שרירותי) ל-()malloc היא תנאי מקדים למתקפה מסוג House of force. דריסה של ה-malloc hook עם הכתובת של ()system (hijacking) ולאחר מכן העברת הכתובת של המחרוזת "/bin/sh" אל malloc כארגומנט המתחפש לגודל בקשה – שקולה להרצת ("/bin/sh")system [<u>שאלה</u>: האם חייבים לכתוב בעצמנו לכתובת כלשהי בזיכרון את המחרוזת "/bin/sh" ולשלוח כתובת זו? <u>תשובה</u>: לא, מחרוזת זו תמיד נמצאת בתוך ה-GLIBC's shared object הנטען למרחב הזיכרון הוירטואלי של התוכנית במהלך הריצה. הסיבה לכך היא שפונקציות (מיובאות) כמו ()system במילא עושות פעולה שקולה ל- ("/bin/sh")system בעזרת קריאת המערכת ()execve המשתמש להרצת הפקודה ש-()system קיבלה]

מגבלות

בגרסה 2.29 של GLIBC הוצגה לראשונה בדיקת שפיות על שדה הגודל של ה-top chunk המוודאת שה-top chunk לא יותר מערך המשתנה system_mem של ה-Arena שלו .ערך משתנה זה מייצג את כמות הזיכרון שה-arena קיבל (checked out) מהקרנל ו"חייב לו בחזרה". הוא קטן ברגע שה-Arena "מחזיר חובות" בעת שחרור זכרון מ-heaps השייכים לאותו arena)

בגרסה 2.29 של GLIBC הוצגה לראשונה בדיקה על גודל ההקצאה המקסימלי, אשר מגביל את גודל הפער הניתן לגישור על ידי שמתקפת ה-House of force.

Fastbin Dup

סקירה כללית:

שימוש בבאג מסוג שחרור-כפול כדי לכפות על malloc להחזיר למשתמש את אותו ה-chunk פעמיים,בלי לשחרר אותו בין לבין. טכניקה זו מבוצעת בדרך כלל ע"י השחתת מטא-דאטה שנוגע ל-fastbin כדי לקשר chunk מזוייף אל ה-fastbin. chunk מזוייף זה יכול להיות מוקצה ולאחר מכן, בהתאם לתוכנית, ישמש לקריאה מ/ כתיבה אל כתובת זיכרון שרירותית.

פרטים:

הבדיקה שלא מבוצע שחרור כפול ל-fastbin רק מוודאת שה-chunk שמנסים לשחרר אל תוך ה-fastbit שונה מה-chunk הראשון (בראש הרשימה, ניתן לבדיקה ע"י השוואת שדה ה-key) של אותו bin, אם chunk אחר מאותו גודל משוחרר בין השחרור הכפול,אז נעבור את הבדיקה.

לדוגמה, נבקש chunks בשם A ו-B, שניהם יהיו מאותו גודל המתאים ל-fastbin כאשר ישוחררו, אחר כך נשחרר את chunk A. אם chunk A ישוחרר שוב מיידית – הבדיקה תיכשל כי הוא כבר נמצא בראש רשימת היעד. במקום זאת, נשחרר את chunk B ואז נשחרר את chunk A שוב. בדרך זו chunk B יהיה בראש הרשימה שבו A משוחרר בפעם השניה. כעת, נבקש 3 chunk-ים מאותו הגודל של A ו-B, malloc יחזיר את ה-chunks בסדר A B A.

דבר זה עשוי להניב הזדמנות לקרוא או לכתוב אל chunk שהוקצה למטרה אחרת. לחילופין, ניתן להשתמש בו כדי לחבל ב-fastbin metadata, בפרט במצביע הקדמי (fd) של ה-chunk המשוחרר פעמיים. זה עשוי לאפשר ל-chunk מזוייף להיות מקושר אל תוך ה-fastbin ולאחר מכן להיות מוקצה. לאחר ההקצאה של ה-chunk המזוייף הוא יוכל לשמש לקריאה או כתיבה מכתובת זיכרון שרירותית. Chunks מזוייפים שמוקצים בצורה זו חייבים לעבור את הבדיקה על שדה הגודל המוודאת שערך השדה שלהם תואם לגודל של אותו fastbin שממנו הם מוקצים.

יש להימנע מפני קומבינציות של דגלים לא תואמים בשדה הגודל המזוייף, דגל NON_MAIN_ARENA דלוק עם CHUNK_IS_MMAPED מאופס יכולים לגרום ל-segmentation fault כש-malloc תנסה לאתר את ה-arena שלא קיים. (ערך של 0xF לדגלים לא יגרום לשגיאה)

שימושים נוספים

ה-Malloc hook הוא מטרה טובה עבור טכניקה זו. אך אם נדרוס את ערך ה-fd עם __malloc_hook-16, נגלה שניכשל בבדיקת ה-size field הנ"ל. למזלנו - שלושת הבתים העליונים (MSB) של מצביע ה-vtable של _IO_wide_data_0 (שתמיד יהיה נוכח בזיכרון, 35 בתים לפני ה-malloc hook) יכולים לשמש, יחד עם חלק מ-quadword הריפוד (מכיל אפסים, תמיד) העוקב כדי לעצב שדה גודל של 0x7f (בעצם נבצע גישה לזיכרון לכתובת לא מיושרת כך שהבית עם הערך 0x7f של ה-vtable ptr יתלכד עם שדה הגודל מנק' המבט של malloc). זה מתאפשר כי הקצאות לא נתונות לבדיקות יישור (alignment) או להשחתת דגלים. שאלה: איך זה יכול להיות שהערך של ה-MSB של ה-vtable ptr (0x7f) לא מושפע מ-ASLR? תשובה: הכתובת שלו כן מושפעת מ-ASLR, אך במרבית מערכות לינוקס x86 – ספריות תמיד ממופות לכתובות המתחילות ב-0x00007f. כמובן שהטריק הזה נשען על כך שביכולתנו לבחור לבקש chunks בגודל 0x70.

מגבלות

בדיקת שדה הגודל עבור fastbin chunks במהלך הקצאה מגבילה את המועמדים ל-chunks מזוייפים

Unsafe Unlink

סקירה כללית:

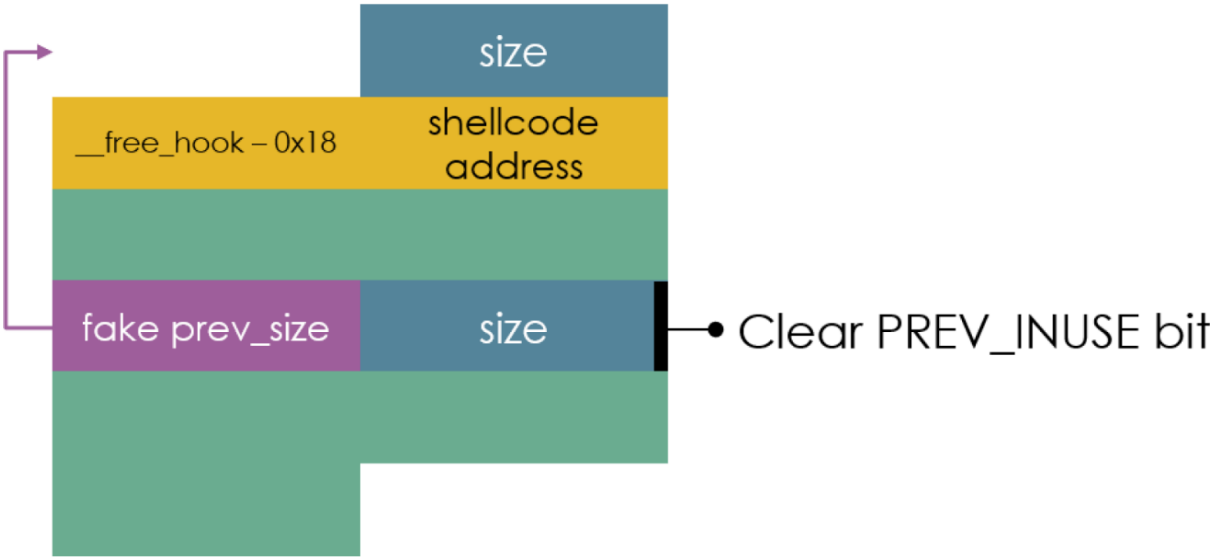
הכרחת המאקרו של ה-unlink לעבד מצביעי fd/bk בשליטת המעצב שתוביל לכתיבה "משוקפת"

פרטים:

במהלך איחוד Chunk, ה-chunk שכבר מקושר לתוך ה-free list מוצא מרשימה זו ע"י מאקרו ה-unlink. תהליך ההתרה הוא מבצע כתיבות ("reflected write") תוך שימוש במצביעי ה-fd וה-bk של ה-chunk. ה-bk של ה-victim מועתק על גבי ה-bk של ה-chunk המוצע ע"י ה-fd של ה-victim וה-fd של ה-victim נכתב על גבי ה-fd של ה-chunk המוצבע ע"י ה-bk של ה-victim. במידה ו-chunk בו המצביעים fd ו-bk ניתנים לעיצוב כרצוננו (בשליטתנו) יותר מתוך ה-free list – נוכל לבצע מניפולציה על הכתיבה.

דרך אחת להשיג זאת היא ע"י גלישה אל תוך שדה הגודל, שבה נאפס את ביט ה-PREV_INUSE. כש-chunk שכזה ישוחרר, malloc ינסה לאחד אותו אחורנית. בעזרת היכולת לעצב את שדה ה-prev_size (ה-qwords שלפני ה-size_field) נוכל לכוון את ניסיון האיחוד אל chunk מוקצה שבו נמצאים מצביעי ה-fd וה-bk המזוייפים

לדוגמה: נבקש שני chunks – A ו-B, המידע שנכתוב ל-A יגלוש לתוך שדה הגודל של B, ו-B מחוץ לטווח של גדלי ה-fastbin. נכין fd ו-bk מזוייפים בתוך A, ה-fd יצביע אל 0x18 – __free_hook וה-bk יצביע אל shellcode שהוכן ונשמר במקום אחר בזיכרון. נכיון שדה prev_size עבור B שיגרום לניסיון לאיחוד אחורי לפעול על ה-fd וה-bk המזוייפים.נמנף את הגלישה כדי לאפס את ביט ה-PREV_INUSE של B.



איור 17 תוכנית מתקפת ה-Unsafe-Unlink שלנו. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

כאשר chunk B משוחרר – ביט ה-prev_size המאופס שלו גורם ל-malloc לקרוא את שדה ה-prev_size של B ולהתיר את ה-chunk שנמצא מספר זה של בתים מאחוריו (חישוב מתמטי במקום לקרוא מצביעים!). כאשר מאקרו ה-unlink פועל על ה-fd וה-bk המזוייפים, הוא כותב את כתובת ה-shellcode אל תוך ה-free hook ובנוסף לכך כותב את הכתובת של 0x18 – __free_hook אל ה-3rd quadword של ה-shellcode. זה לא כזה נורא כי ה-shellcode יכול להשתמש בפקודת jump כדי לדלג מעל הבתים שהושחתו עם ערך ה-fd. קריאה ל-()free תריץ את ה-shellcode

שימושים נוספים

בדוגמה הקודמת, ניתן גם להשתמש בשדה prev_size עם ערך 0 ולעצב את מצביעי ה-fd וה-bk המזוייפים בתוך chunk B. אותה הטכניקה ניתנת ליישום עבור איחוד קידמי, אך דורשת שליטה נוקשה יותר על ה-heap.

מגבלות

טכניקה זו תפעל רק בגרסאות GLIBC <= 2.3.3. הפיכת תהליך ההתרה לביטוחתי יותר ("safe unlink") הוכנסה בגרסה 2.3.4 בשנת 2004. גרסאות כה ישנות של GLIBC דיי נדירות. טכניקה זו נוצלה לראשונה נגד פלטפורמות ללא NX/DEP (Non-executable stack, heap and data section) כמתואר בדוגמה הנ"ל. בשנת 2003 AMD הכניסו לשימוש תמיכה ב-NX בחומרה למעבדי הדסקטופ לצרכנים, וב-2004 אינטל עשתה כמו כן. מערכות ללא הגנה זו לא נפוצות (ניתן למצוא כאלו אולי בשוק ה- Embedded systems & Internet-of-things).

Safe Unlink

סקירה כללית

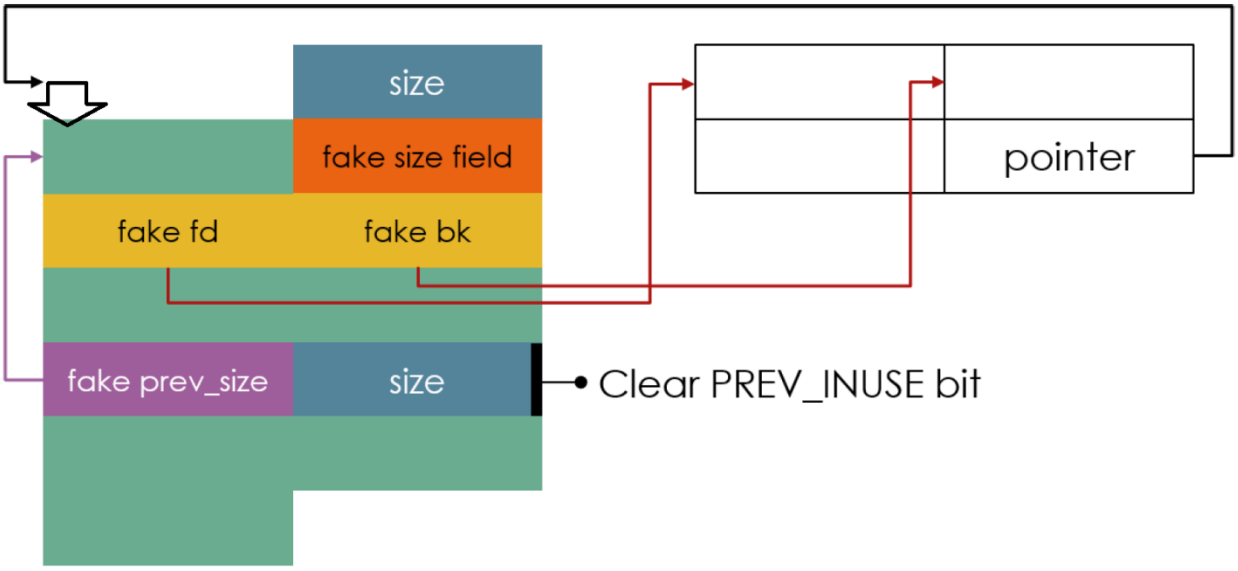
המקבילה המודרנית של טכניקת Unsafe Unlink. מאלצת את מאקרו ה-unlink לעבד את מצביעי ה-fd וה-bk שבשליטת מעצבם, שתוביל ל"reflected write". עמידה בתנאי הבדיקות החדשות מושגת ע"י כיוון הכתיבה הזו אל מצביע אל chunk שנמצא בשימוש (מוקצה). בהתאם לקוד התוכנית – נוכל לדרוס מצביע זה שוב, שבתורו ישמש לכתיבה אל או קריאה מכתובת שרירותית.

פרטים

טכניקה זו דומה לקודמת, אבל לוקחת בחשבון את הבדיקות שהוכנסו בגרסה 2.3.4. בדיקת ההתרה הבטוחה מוודאת שה-chunk שאנו מנסים להתיר מרשימה הוא חלק מרשימה דו כיוונית לפני ביצוע ההתרה. הבדיקה תעבור בהצלחה אם ה-bk של ה-chunk שמוצבע ע"י ה-fd של ה-victim chunk מצביע בחזרה אל ה-victim, ובאותו אופן – שה-fd של ה-chunk שמוצע ע"י ה-bk של ה-victim מצביע בחזרה אל ה-victim.

נעצב chunk מזוייף המתחיל ב-quadword הראשון של user data של chunk לגיטימי מוקצה, נכתוב לערכי ה-fd וה-bk שלו את הכתובות הנמצאות 0x18 ו-0x10 בתים לפני מצביע (הנמצא בתוך user data של chunk דמיוני אחר שלא חייב להיות ב-heap - מופיע באיור הבא כ-'pointer') ל-user data של ל-chunk הלגיטימי בו הם יושבים,

נעצב את שדה ה-prev_size (בסגול) עבור ה-chunk העוקב שהינו 0x10 בתים פחות מאשר הגודל האמיתי של ה-chunk הקודם (כדי ש-malloc יחשוב שה-chunk שלפניו הוא ה-chunk המזוייף שבנינו, והרי שאז ה-chunk ש-malloc יפעיל עבורו את מאקרו ה-()unlink יהיה זה שמתחיל ב-user data של ה-chunk הלגיטימי). ננצל באג מסוג גלישה כדי לאפס את ביט ה-PREV_INUSE (כדי לאפשר את האיחוד האחורי), וכאשר chunk זה ישוחרר – malloc ינסה לאחד אותו אחרונית עם ה-chunk המזוייף



איור 18 Safe-Unlink עיצוב chunk מזוייף בתוך chunk לגיטימי. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

נשים לב שכך ה-bk של ה-chunk המוצבע ע"י ה-fd של ה-chunk המזוייף מצביע בחזרה ל-chunk המזוייף, ושה-fd של ה-chunk המוצבע ע"י ה-bk של ה-chunk המזוייף גם כן מצביע חזרה אל ה-chunk המזוייף. כך קיימנו את דרישות בדיקת ההתרה הבטוחה. תוצאת תהליך ההתרה היא שהמצביע ל-chunk המזוייף (מצביע ל-user data של chunk לגיטימי) נדרס עם הכתובת של עצמו פחות 0x18 (24 בתים)

אם מצביע זה משמש לכתיבה מידע, אפשר להשתמש בו כדי לדרוס את עצמו פעם שניה עם הכתובת של מידע רגיש (הזלגת מידע) ולאחר מכן להשתמש בו כדי לחבל במידע זה.

הבהרה- 'pointer' הוא המצביע שהמשתמש קיבל ל-user data של ה-chunk הראשון שהוקצה. אנו מניחים שאנו יודעים היכן המשתנה שמור בזיכרון (לדוגמה: ע"י שימוש בשם של מערך על ה-stack שמכיל מצביעים כאלו)

שימושים נוספים

ע"י עיצוב של prev_size גדול מאוד – הניסיון לאיחוד עשוי להקיף את מרחב הזיכרון הוירטואלי ולפעול על chunk מזוייף בתוך ה-chunk המשוחרר

מגבלות

גרסה 2.26 של GLIBC הציגה בדיקה של שדה הגודל (size) לעומת prev_size הדורשת ששדה גודל של ה-chunk המזוייף יעמוד בתנאי פשוט: שהערך הנמצא בכתובת &fake_chunk + size_field יהיה זהה לשדה הגודל שלו (זאת דרך "לא בטוחה" לבדוק שערך ה"fake prev_size" זהה ל-fake size_field), דוגמה לכך שהיא לא בטוחה: אם נכתוב לתוך שדה הגודל של ה-chunk המזוייף את הערך 8, הוא תמיד יעבור את הבדיקה (לא משנה מה יש ב-fake prev_size). בדיקה נוספת דומה הוצגה בגרסה 2.29, היא דורשת שערך שדה הגודל של ה-chunk המזוייף יתאים לערך ה-prev_size (שלו) שעיצבנו.

Unsortedbin Attack

סקירה כללית

מתקפה זו מניבה לנו primitive שנראה בלתי מזיק, אך כאשר היא מוצמדת לטכניקות אקספלוטציית heap אחרות – מקבלים אפקט ניכר. היא כותבת את כתובת ה-unsortedbin של ה-arena למיקום שרירותי בזיכרון.

פרטים

בעת ש-chunk מוקצה או ממויין מתוך unsortedbin, הוא נתון למה ש-malloc מתייחס אליו כ"התרה חלקית". זהו התהליך של הסרת chunk מתוך קצה הזנב של רשימה מקושרת דו-כיוונית מעגלית, וחלק אחד מתהליך זה כרוך בכתיבת הכתובת של ה-unsortedbin (chunk דמיוני השמור בתוך ה-arena ומכיל רק מצביע קדמי לראש ואחורי לזנב רשימת ה-unsortedbin) על גבי מצביע ה-fd של ה-chunk המוצבע ע"י שדה ה-bk של של ה-victim chunk.

במתקפה זו אנו מעצבים את מצביע ה-bk של chunk המקושר אל תוך ה-unsortedbin ע"י באג גלישה או write-after-free, לאחר מכן מקצים את ה-chunk מה-unsortedbin ובכך קיבלנו כתיבה של כתובת ה-unsortedbin אל 0x10 בתים אחרי הכתובת שהכנו מראש (שדה ה-fd) במצביע ה-bk.

שימושים נוספים

שימושים מודרניים במתקפה זו כוללים הזלגה של כתובת מ-libc, למשל כתיבה של כתובת ה-unsortedbin של ה-main arena אל תוך חוצץ פלט.

ניתן להשתמש בה כדי לנטרל את בדיקת השלמות של ה-vtable של libio בגרסאות GLIBC 2.24-2.26 ע"י סימון סימבול ה-dl_open_hook_ כיעד כתיבה. ניתן להשתמש בה גם כדי להשחית את המשתנה global_max_fast כדי למנף פרימיטיב מסוג House of prime. טכניקת ה-House of Orange מטרגטת (מעוניינת לכתוב אל) את סימבול ה-IO_list_all_ בעזרת unsortedbin attack כחלק מניסון לבצע אקספלוטציה של File stream (FSOP- File Stream oriented programming)

מגבלות

מגוון בדיקות שלמות על ה-unsortedbin הוכנסו בגירסה 2.29 של GLIBC, אחת מהן מוודאת שה-chunk המוצבע ע"י victim.bk->fd הוא אכן ה-victim chunk ובכך לשבש מתקפה זו.

(2016 - An-Jie Yang - Angelboy) House of Orange

סקירה כללית

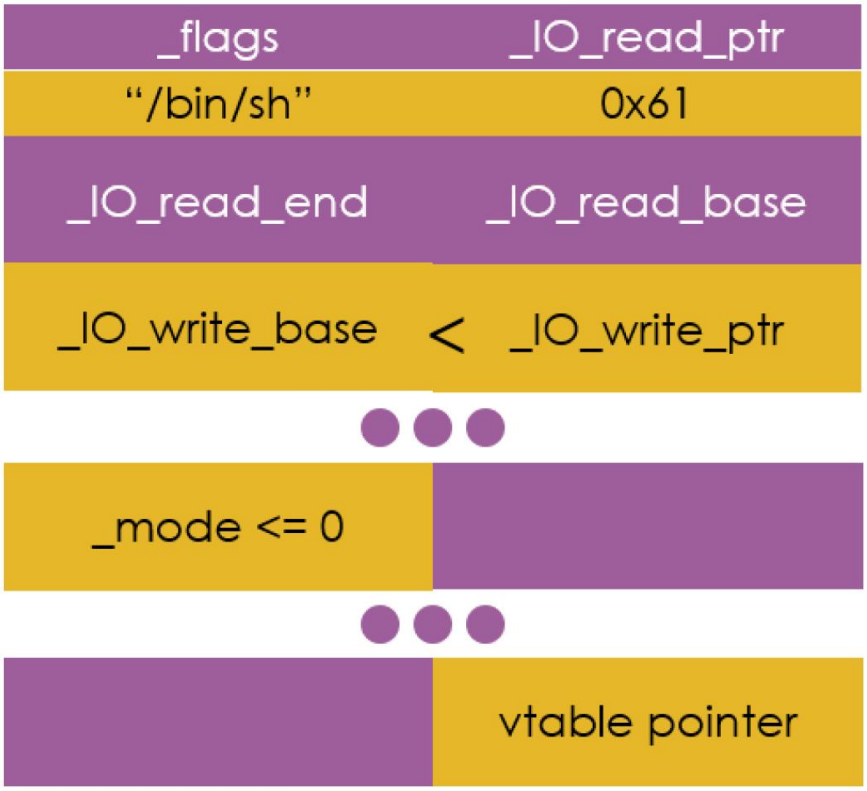
טכניקה זו ממנפת גלישה אל תוך ה- top chunk של ה-main arena על מנת להשיג shell (באנגלית אומרים לעיתים drop/pop a shell), היא עושה שימוש באקספלוטציה של file stream (FSOP) כדי להשיג יכולת הרצת קוד וכוללת ישום חדש/מקורי של מתקפת unsortedbin.

פרטים

ניתן לפרק מתקפה זו ל-3 שלבים: הרחבת ה-top chunk, מתקפה על ה-unsortedbin, ו-FSOP. השלב הראשון מנצל את הדרך שבה ה-main arena מבצע הרחבה של ה-top chunk. כאשר ה-top chunk של ה-main arena ממוצה עד תום, נשלחת בקשה לעוד זיכרון מהקרנל ע"י ה-syscall brk(). אם זיכרון זה אינו רציף ביחס ל-top chunk – הוא יסומן כ-top chunk החדש, וה-top chunk הישן ישוחרר. ה-House of Orange מנצל עובדה זו ע"י גלישה אל ה-top chunk כדי לכתוב על גבי שדה הגודל שלו ערך קטן המיושר לגודל דף (השומר על הישור), לאחר מכן נבקש מספר גדול מדי של בתים כך שה-top chunk (שהתכווץ כרגע בכאילו) לא יוכל לשרת את הבקשה. מה שיגרום ליצירת free chunk המקושר אל תוך ה-unsortedbin העובר במסלול של הגלישה.

השלב השני עוסק במינוף של באג הגלישה בפעם השניה, הפעם כדי לדרוס את מצביע ה-bk השייך ל-top chunk הישן שכעת נמצא ב-unsortedbin של ה—Main arena. דריסה זו היא על מנת לכוון מתקפת unsortedbin על מצביע ה-_IO_list_all, שהולך להיעשות בו שימוש ע"י הפונקציה _IO_flush_all_lockp() כדי לבצע flush (כתיבה לתוך מבנה FILE של כל מה שעוד לא הספיק להיכתב מחוצץ הכתיבה) לכל ה-file streams הפתוחים.

במהלך הכתיבה על גבי שדה ה-bk של ה-top chunk הישן, מעוצב file stream מזוייף על גבי ה-heap בעזרת גלישה בצורה כזו ששדה ה-_flags של ה-FS המזוייף חופף למיקום ה-prev_size ב-top chunk הישן. שדה ה-_mode יאותחל לערך קטן או שווה לאפס וה-_IO_write_ptr יאותחל לערך הגדול יותר מערך שדה ה-_IO_write_base (אם ההפרש ביניהם חיובי, סימן שיש מידע שעדיין לא נכתב). אל שדה המצביע ל-vtable נכתוב כתובת של vtable מזוייפת (בכל מיקום שהוא שבו מעצב המבנה יכול ליצור אחת כזו) שבה כניסת ה-__overflow (מצביע לפונקציה, int __overflow (FILE *f, int ch)) תוחלף בכתובת של system().
המחרוזת "/bin/sh\0" תיכתב לתוך שדה ה-_flags של ה-FS המזוייף ולשדה הגודל של ה-top chunk הישן (החופף במיקום לשדה ה-_IO_read_ptr נכתוב את הערך 0x61.



איור 19 דוגמה ל-File stream מזוייף. מתוך HeapLAB Heap Exploitation Bible, Max Kamper, באישור היוצר

איור זה מציג דוגמה ל-File stream מזוייף, עם שדות שאנו חייבים לכתוב אליהם בצבע צהוב ושאר שדות המבנה (members) בסגול.

לאחר מכן, נבקש chunk בגודל שונה מ-0x60. ה-top chunk הישן ימויין לתוך ה-0x60 smallbin במהלך סריקת ה-unsortedbin אחר מועמד להקצאה, מה שיעורר את מתקפת ה-unsortedbin (הוא מוצא מן הרשימה) שבתורה תכתוב על גבי מצביע ה-_IO_list_all את כתובת ה-unsortedbin (כתובת מה-arena). הסריקה תמשיך וה-"chunk" החופף למצביע _IO_list_all יכשל בבדיקת השפיות של שדה הגודל, ובכך תיקרא הפונקציה abort().הפונקציה abort() תבצע flush לכל ה-file streams (הפתוחים) בעזרת _IO_flush_all_lockp() אשר קוראת את ערך המצביע _IO_list_all כדי למצוא את ה-fs הראשון. ה-fs המזוייף החופף ל-main arena (אנחנו בעצם בונים אותו על גבי ה-main arena) לא נשטף, ומצביע ה-_chain שלו (החופף למצביע ה-bk של ה-0x60 smallbin) נעקב (מתקדמים הלאה ברשימה לשאר ה-FS שאולי זקוקים ל-flush) וממנו מגיעים ל-fs מזוייף אחר על גבי ה-heap (כזכור, ה-main arena נמצא בתוך ה-data section של glibc ולא בזכרון heap). שדות ה-_mode, _IO_write_base & _IO_write_ptr של ה-fs המזוייף (השני) מבטיחים שכניסת ה-__overflow ב-vtable של ה-fs המזוייף (השני) תיקרא עם הכתובת של ה-fs המזוייף כארגומנט ה-1 שלה. כך נקבל קריאה ל-system("bin/sh")

מגבלות

כאשר מנסים לקבוע האם ה-file stream המזוייף החופף ל-main arena צריך "להישטף", ערך תא ה-fd של ה-smallbin 0xb0 הופך ל-fp->_wide_data->_IO_write_ptr, וה-bk של ה-smallbin 0xa0 הופך ל--fp->_wide_data->_IO_write_base. כאשר bins אלו ריקים, תמיד נקבל ש-fp->_IO_write_base יהיה קטן מ-fp->_IO_write_ptr, מה שיגרור קריאה לפונקציה __overflow() של ה-fs אם fp->_mode>0. כאשר זה קורה, fp->_vtable יחפוף ל-bk של ה-smallbin 0xd0 ובכך גם הכניסה __overflow() ב-vtable תחפו, ל-bk של ה-smallbin 0xd0 ב-main arena ריק. במידה ושדה ה-__overflow() של ה- vtable נקרא – נקבל segfault.

אם שדה ה-fp->_mode של ה-file stream בתוך ה-main arena (אשר חופף ל-fd של ה-smallbin 0xc0) גדול מ-0 אז כניסת ה-__overflow() ב- vtable תיקרא והתוכנית תחטוף segfault במהלך ניסיון להרצת נתונים המסומנים כ-Non executable. על כן – טכניקה זו תעבוד רק כאשר ה-dword (16 בתים) הנמוך של ה-fd של ה-smallbin 0xc0 יכול להיות מפורש כמספר שלילי. הביט הקובע מצב זה נתון להשפעת ASLR ולכן טכניקה זו עובדת בקירוב רק 50% מהזמן, כתלות בכתובת הטעינה של libc בזיכרון.

House of Spirit

סקירה כללית

העברה של מצביע שרירותי אל פונקציית free(), קישור chunk מזוייף אל תוך bin הניתן להקצאה אח"כ.

פרטים

זוהי הטכניקה היחידה שלא נשענת על אחד מהבאגים השגרתיים של ה-heap, במקום זאת, היא מנצלת את התרחיש שבו מתאפשר למעצב המתקפה להשחית מצביע שמועבר לאחר מכן ל-free() כארגומנט.

ע"י העברת מצביע ל-chunk מזוייף אל free(), ה-chunk המזוייף יכול להיות מוקצה ובכך נוכל לכתוב על גבי מידע רגיש. ה-chunk המזוייף חייב להיות עם שדה גודל מתאים ובמקרה של fast chunk הוא חייב של-chunk העוקב (הבא) יהיה שדה גודל שיקיים את תנאיי בדיקות השפיות על הגודל (האם נמצא בטווח ערכים), כלומר שהמעצב חייב לשלוט בלפחות שני quadwords שביניהם נמצא ה-target data (היכן שאנו מעוניים לכתוב)

במקרה של small chunk, חייבים להיות לנו שני שדות גודל "מזדנבים" (בסוף שני ה-chunks העוקבים) כדי להבטיח שלא יתבצע ניסיון לאיחוד קדמי. fencepost chunks יעשו את העבודה. בשל כך, המעצב נדרש לשלוט בלפחות שלושה quardwords מסביב ל-target data במקרה זה.

שימושים נוספים

כאשר טכניקה משולבת עם הזלגת כתובת של ה-heap, אפשר להשתמש בה כדי לגרום להפעלת double free – מה שיכול לתת לנו פרימיטיב חזק יותר.

מגבלות

אם דגל הרצף (contiguity) של arena דלוק, small chunk מזוייפים חייבים להימצא בכתובת נמוכה יותר מאשר זו של ה-heap של ה-thread שלהם. אילוץ זה לא תקף ל-fast chunks מזוייפים. Chunks מזוייפים חייבים לעבור בדיקת יישור המוודאת לא רק שהם מיושרים לפי 16 בתים, אלא גם שהביט LSB ה-4 בשדה הגודל שלהם מאופס.

Chunks מזוייפים חייבים להימנע מכך שהביטים NON_MAIN_ARENA ו-IS_MMAPED שלהם יהיו דלוקים, במקרה של הראשון – free() תחפש arena שאיננו קיים וכנראה יתקבל segfault במהלך פעולה זו, ובמקרה של השני – יבוצע unmap() על ה-chunk המזוייף במקום free.

House of Lore

סקירה כללית

קישור chunk מזוייף אל ה-unsortedbin, ה-smallbins או ה-largebins ע"י שינוי metadata של malloc.

פרטים

קישור של chunk מזוייף אל תוך ה-unsortedbin שקול ללכוון למתקפת unsortedbin על chunk מזוייף ע"י דריסת תוכן ה-bk של ה-unsortedbin עם כתובת ה-chunk המזוייף. ל-chunk המזוייף חייב להיות bk שמצביע לכתובת הניתנת לכתיבה. ה-chunk המזוייף יכול להיות מוקצה ישירות מה-unsortedbin, אם כי שדה הגודל שלו חייב להתאים לגודל הבקשה וחייב להיות שונה מזה של ה-chunk עם ה-bk שהושחת.

קישור של chunk מזוייף אל תוך ה-smallbin מחייב דריסה של ערך ה-bk של chunk המקושר אל תוך ה-smallbin (נקרא לו small chunk) עם כתובת ה-chunk המזוייף וכי נבטיח ש-victim == victim->bk->fd כדי לעבור את אותה הבדיקה ע"י כתיבה של כתובת של ה-victim (ה-small chunk) אל תוך ה-fd של ה-chunk המזוייף לפני שה-small chunk מוקצה. מרגע שה-small chunk מוקצה, ה-chunk המזוייף חייב גם הוא לעבור את אותה הבדיקה, אפשר להשיג זאת ע"י כך שנגרום ל-fd ול-bk שלו להצביע על עצמו. בתרחישים בהם לא ניתן לערוך את ה-chunk המזוייף לאחר שה-small chunk הוקצה, ניתן להשתמש ב-chunk מזוייף שני, אם כי רק quadword אחד נחוץ כדי להחזיק fd מזוייף. אם נגרום ל-fd של chunk מזוייף זה (השני) להצביע על ה-chunk המזוייף העיקרי (הראשון), ול-bk של ה-chunk המזוייף העיקרי להצביע על ה-chunk המזוייף השני – נקיים את תנאי הבדיקה. הגודל של ה-chunk המזוייף לא רלוונטי משום שהוא לא נבדק בתהליך זה.

הדרך הקלה ביותר כדי לקשר chunk מזוייף אל תוך largebin כרוכה בדריסת ה-fd של skip chunk (כזה שמוכל ברשימת הדילוגים) עם הכתובת של chunk מזוייף והכנת ה-fd וה-bk של ה-chunk המזוייף כך שיקיימו את בדיקות ה-safe unlinking. ה-chunk המזוייף חייב להיות עם אותו שדה גודל כמו של ה- skip chunk, וה-skip chunk חייב שיהיה בנוסף אליו עוד chunk בגודל זהה או קטן ממנו הנמצא באותו largebin יחד איתו, וזאת כי malloc **לא יבדוק** את "אמינות" ה-chunk המוצבע ע"י ה-fd של ה-skip chunk במידה וה-skip chunk נמצא ב-bin האחרון. ניתן לעצב מראש את ה-fd וה-bk של ה-chunk המזוייף כך שיקיימו את בדיקות ה-safe unlinking אם שניהם יצביעו על ה-chunk המזוייף

מגבלות

הכמות והמיקום המדוייק של זיכרון הנדרש שיהיה שבשליטנו כדי לבנות small chunks ו-large chunks מזוייפים יכול להפוך טכניקה זו לקשה למימוש כנגד אותם bins.

House of Einherjar

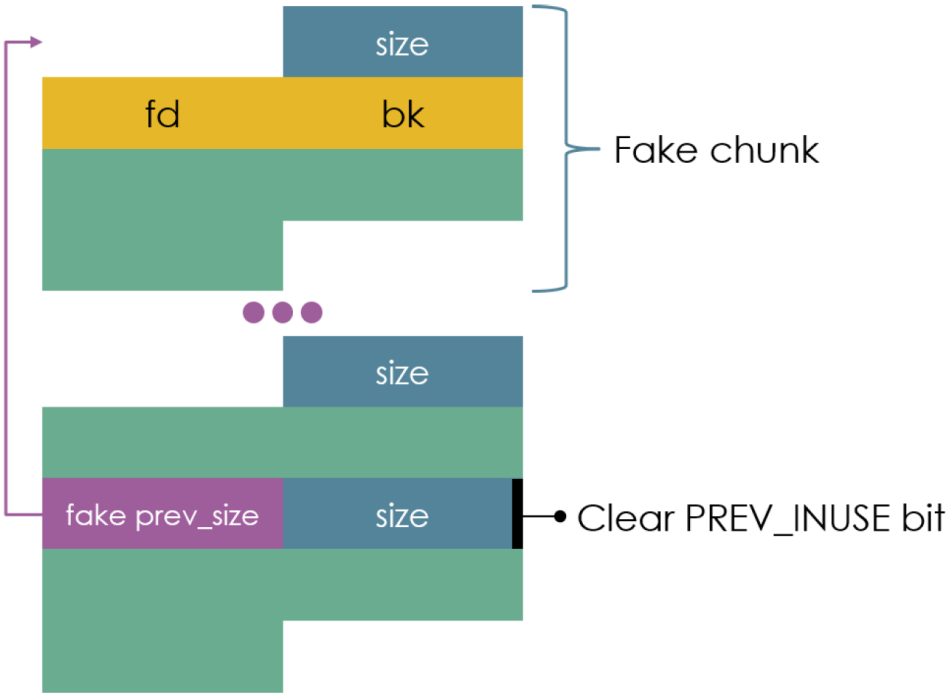
סקירה כללית

איפוס של ביט ה-PREV_INUSE של chunk ואיחוד אחורי שלו עם chunk מזוייף או עם free chunk קיים כדי ליצור חפיפה בין chunks

פרטים

טכניקה זו הוצגה במקור כטכניקה מסוג גלישה של null-byte יחיד, אך זהו לא היישום הכי ריאלי שלה. היא מניחה שהגלישה יכולה לאפס את ביט ה-PREV_INUSE של ה-victim chunk תוך כדי שליטה בשדה ה-prev_size שלו.

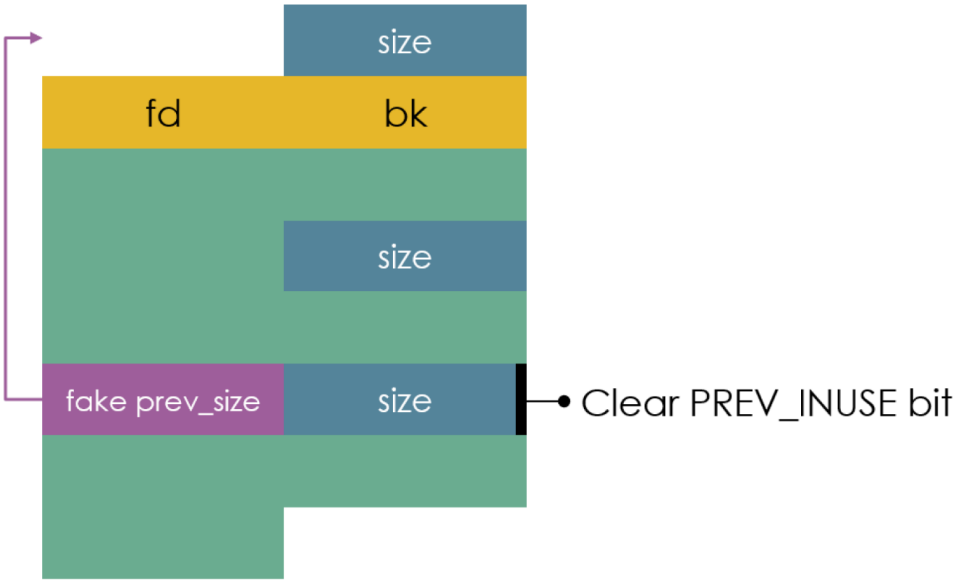
שדה ה-prev_size של ה-victim מאוכלס בצורה כזו שכאשר ה-victim משוחרר, הוא יאוחד אחורית עם chunk מזוייף הנמצא על ה-heap או במקום אחר. במקרה זה, הקצאות שרירותיות יכולות להיעשות מתוך ה-chunk המזוייף ואלו יכולות לשמש כדי לקרוא או לכתוב מידע רגיש



איור 20 House of Einherjar. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

שימושים נוספים

ניתן גם לבצע איחוד עם free chunks לגיטימיים הנמצאים ב-heap, ובכך ליצור חפיפה בין chunks כדי לבנות primitive חזק יותר



*איור **21** House of Einherjar משודרג. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר*

מגבלות

בגרסה 2.26 של GLIBC הוצגה בדיקה של size לעומת prev_size הדורשת מהמעצב להכין שדה גודל מתאים עבור ה-chunk המזוייף שלו.

House of Rabbit (Shimizu Yutaro – "Shift-Crops" from TokyoWesterns)

סקירה כללית

השגת primitive הדומה לזה של House of Force ע"י קישור של chunk מזוייף אל תוך ה-largebin הכי גדול והגדרת שדה הגודל שלו לערך גדול מאוד. השם של טכניקה זו נבחר כך משום שאנו "מקפיצים" את ה-chunk בין bins שונים לצורך השגת המטרה.

פרטים

מינוף של באג ב-heap כדי לקשר chunk מזוייף אל תוך fastbin, ה-chunk המזוייף כולל שני שדות גודל: אחד שייך ל-chunk המזוייף עצמו והאחר שייך לזה של ה-chunk העוקב. יחד עם זאת, שדה הגודל של ה-chunk העוקב ממוקם 0x10 בתים לפני שדה הגודל של ה-chunk המזוייף.

ה-chunk המזוייף מקיף את מרחב הזיכרון הוירטואלי עם שדה גודל 0xfffffffffffffff1 (אחרי הפעלת ה-mask לאיפוס הדגלים, זה שווה ערך לגודל של 16-) והגודל של ה-chunk העוקב יאותחל ל-0x11 (fencepost chunk, זוהי דריסת-הרגל/השטח הקטן ביותר של זיכרון ש-chunk מזוייף) בעוד שהוא מקיים את בדיקות ה-fastbin על ה-next size (גודל ה-chunk העוקב) ובכך להימנע מניסינות לאיחוד.

איור 22 House of Rabbit מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

אחת וה-chunk המזוייף מקושר אל תוך ה-fastbin, הוא יאוחד אל תוך ה-unsortedbin במסגרת malloc_consolidate(). אנחנו לא יכולים (לא רוצים) להפעיל את malloc_consolidate() דרך malloc() כי אז ה-chunk המזוייף ימויין – מה שיגרום ל-abort() להיקרא כאשר תיכשל בדיקת השפיות על הגודל.

במקום זאת, אנו נגרום ל-chunk המזוייף להיות ממויין ע"י שחרור של chunk שגודלו מעבר ל-FASTBIN_CONSOLIDATION_THRESHOLD (0x100000 כברירת מחדל), ניתן להשיג זאת ע"י שחרור של normal chunk (non-large) שגובל ב-top chunk משום ש-()_int_free מתחשבת בכל השטח המאוחד כגודל ה-chunk שנקראה עבורו.

נשנה את גודל ה-chunk המזוייף כדי שהוא יוכל להתמיין אל תוך ה-largebin הגדול ביותר (בקוד מופיע כ-bin[126]), malloc מחפשת רק ב-bin זה עבור בקשות גדולות מאוד. על מנת להתאים ל- bin זה, ה-chunk המזוייף חייב להיות בעל גודל של לפחות 0x80001. נמיין את ה-chunk המזוייף אל תוך bin[126] ע"י בקשת chunk גדול יותר. אם משתנה ה-system_mem של ה-arena קטן מ-0x80000 (ואכן כך יהיה עבור מצבי ברירת מחדל כאשר ה-heap לא הורחב), יידרש מאיתנו להגדיל בצורה מלאכותית את system_mem ע"י בקשת large chunk תחילה, שחרורו ולבסוף לבקש אותו שוב.

כעת, לאחר שה-chunk המזוייף קושר אל תוך ה-largebin הגדול ביותר, זה בטוח לשנות את גודלו בחזרה ל-0xffffffffffffffff1. שימו לב שערך כזה גדול של גודל עלול שלא להתאים כאשר ננסה (נרצה) לדרוס משתנים על המחסנית עקב כך שגודל ה-chunk המזוייף עלול להיות גדול מ-av->system_mem לאחר ההקצאה. דבר שכזה יכשיל את בדיקת השפיות על הגודל במהלך הקצאות עוקבות מה-unsortedbin.

איור 23 המסלול שעובר ה-Chunk המזוייף בין ה-bins השונים ב-House of Rabbit. מתוך HeapLAB Heap Exploitation Bible, Max Kamper , באישור היוצר

וכך השגנו פרימיטיב דומה לזה של House of Froce שבא לידי ביטוי בכך שמענה לבקשה גדולה יכול להיעשות מתוך ה-chunk המזוייף שמתפרש (spans) על פני המרווח בין ה-chunk המזוייף לזיכרון המטרה.

שימושים נוספים

דרך חלופית במקום שימוש ב-large chunk מזוייף המקיף את מרחב הזיכרון הוירטואלי במהלך הקישור הראשוני אל תוך ה-fsatbin היא להשתמש ב-fast chunk מזוייף עם fencepost chunks בסיומו. זה דורש שזיכרון נוסף יהיה בשליטתנו אך עוקף את בדיקות השפיות על הגודל שבתוך malloc_consolidate(), נכון לגירסה GLIBC 2.27.

מגבלות

הבדיקה של size לעומת prev_size הוצגה לראשונה ב-2.26 ומשמעותה שהמעצב חייב לאכלס ידנית את שדה ה-prev_size של ה-fencepost chunk המזוייף.

(Chris Evans, Project Zero @Google 2014) Poision Null Byte

[מבוסס על הוריאציה shrink_free_hole_alloc_overlap_consolidate_backward.c]

סקירה כללית

מינוף באג גלישה של null-byte יחיד לצורך יצירת chunks חופפים מבלי צורך בעיצוב שדה prev_size מזוייף.

פרטים

טכניקה זו מתמקדת בגלישה של null-byte יחיד בתרחיש מציאותי של סיום מחרוזת (כתיבת '\0') במקום לא נכון. בתרחישים מציאותיים שכאלו, לא סביר שנוכל לספק שדה prev_size מזוייף מאחר שסביר מאוד שה-quadword שלפני שדה הגודל של ה-chunk העוקב מחזיק מחרוזת שקשה להפוך אותה לערך הגיוני של שדה prev_size מבלי להשתמש ב-null bytes (הרי ברגע שנעשה זאת, בהתאם להתנהגות פונקציות libc שקוראות מחרוזות, מבחינתן שם הן מפסיקות לקרוא. זהו גם אחד מהאתגרים בכתיבת shellcodes).

הגלישה חייבת להיות מכוונת אל free chunk עם שדה גודל של 0x110 או גדול יותר, כאשר זה קורה הבית הנמוך ביותר של שדה הגודל (לשים לב ל-little endian) מאופס, כך שמנקודת המבט של malloc נאבד משם גודל של 0x10 בתים או יותר. כאשר ה-victim chunk מוקצה שוב, שדה ה-prev_size העוקב לא יעודכן. ניתן לנצל זאת כך:

נבקש 4 chunk-ים, נקרא להם A עד D: chukn A נועד כדי לבצע גלישה אל תוך chunk B שהינו בגודל 0x110 או גדול מכך. chunk C יהיה בגודל נורמלי (שלא מתאים לגודל של fastbin). chunk D בשימוש כדי להימנע מאיחוד עם ה-top chunk, זה לא הכרחי אך עוזר להפוך את תרגול הטכניקה לברור יותר. נשחרר את chunk B אל תוך ה-unsortedbin של ה-arena שלו ונבצע את הגלישה אל תוך chunk B שדיברנו עליה שתאפס את הLSByte שלו.

כעת נבקש שני chunks שיוקצו בשטח ש-B השאיר אחריו לאחר שחרורו (הוא יפוצל), במטרה ליצור שני chunks שנקרא להם B1 ו-B2, B1 חייב להיות בגודל normal (גדול מזה של ה-fastbin). נשחרר את B1 ולאחר מכן את C. Chunk C יאוחד אחורית עם B1 כך שתוצר האיחוד יחפוף את B2 (יכיל אותו בתוכו) בגלל ששדה ה-prev_size שלו לא עודכן מאז ש-B שוחרר. לבסוף נבקש זיכרון שיוגש מה-chunk החופף ל-B2 שעדיין מוקצה (שם נמצא ה-target data)

מגבלות

בגרסאות GLIBC >= 2.26 חייבים לעמוד בתנאי הבדיקה של size לעומת prev_size במאקרו/פונקציית ה-unlink כאשר chunk B מוצא מן ה-unsortedbin בתהליך ה-remaindering.

בגרסאות GLIBC >= 2.29 הבדיקה של size לעומת prev_size לפני פונק' ה-unlink תיכשל כאשר chunk C משוחרר כי שדה הגודל של B1 לא נכון (הגלישה שינתה אותו) ואין לנו איך לשנות אותו.

House of Corrosion (Max Kamper – "CptGibbon", 2019)

סקירה כללית

טכניקה זו ממנפת באג Write-After-Free קטן כדי להשיג shell כנגד בינארים מסוג PIE (Position independent execution) שלא מזליגים כתובות בכלל (leak-less) אך דורשת שליטה רבה על ה-heap. החיסרון העיקרי שלה הוא הצורך בניחוש 4 ביטים (nibble) של כתובת הטעינה(?)

פרטים

טכניקה זו עושה שימוש בפרימיטיב של House of Prime (שלא הוזכר כאן) בשילוב של באג WAF כדי להשיג shell בעזרת File stream exploitation (FSOP). באג ה-WAF בדוגמה המוצגת (זו שתודגם בחלק ה-3 בסדרה של הקורס הנ"ל יחד עם תכנים נוספים כמו FSOP. ישוחרר לקראת 2022) הוא בגודל-משתנה, כלומר שהמעצב יכול לכתוב על גבי בית אחד או יותר מתוך ה-metadata של free כראות עיניו. ניתן להשיג פרימיטיב זה גם ע"י הפעלה של אותו באג מסוג WAF על שדות שונים של metadata של free או ע"י כתיבה לשדות שונים של האובייקט המשוחרר. טכניקה זו מתוארת כאן עבור גרסה 2.27 של GLIBC אך גישה אחרת המשתמשת באותם כלים תהיה אפקטיבית מול גרסה 2.29.

התוצאה של שימוש בבאג ה-WAF כדי לבצע מתקפת unsortedbin כנגד משתנה ה-global_max_fast היא שפרימיטיב ה-House of Prime שבו שחרור של large chunks כותב את הכתובת שלהם אל מיקום שרירותי בהיסט מסויים מ"מערך" ה-fastbins ה. במקרה שבו ה-heap שייך ל-main arena, ניתן להשתמש בפרימיטיב זה כדי לבצע שינויים בסגמנט הניתן לכתיבה של libc. מתקפת ה-unsortedbin מתאפשרת ע"י דריסה של 2 הבתים הנמוכים של שדה מצביע ה-bk של ה-unsortedbin.

Tcache Dup

סקירה כללית

מינוף של באג מסוג שחרור כפול כדי לגרום ל-malloc להחזיר את אותו ה-chunk פעמיים, מבלי לשחרר אותו בין לבין. טכניקה זו מושגת בד"כ ע"י השחתת ה-metadata של ה-tcache כדי לקשר chunk מזוייף אל תוך tcachebin. את ה-chunk המזוייף הזה נוכל להקצות, ולאחר מכן, בהתאם לפונקציונליות התוכנית, נוכל לקרוא מ/לכתוב אל מיקום שרירותי בזיכרון.

פרטים

טכניקה זו פועלה בסגנון דומה לזה של מתקפת ה-Fastbin Dup, ההבדל העיקרי טמון בכך שבגרסאות GLIBC לפני 2.29 אין הגנה בפני שחרור-כפול ב-tcache. טכניקת ה-Tcache dup נותנת פרימיטיב עוצמתי משום שאין בדיקת שלמות על שדה הגודל בעת הקצאות מתוך ה-tcachebin! (הרי הוא התווסף מתוך מטרה לשפר את הביצועיים של התוכנית), מה שמאפשר לנו ביתר קלות לחפוף tcache chunk מזוייף עם כל כתובת בזיכרון.

שימושים נוספים

בגרסה 2.29 של GLIBC התווספה בדיקה לתפיסת שחרור-כפול ב-tcache: כאשר chunk מקושר אל תוך ה-tcachebin, הכתובת של ה-tcache של ה-thread שרץ נכתבת אל תוך תא שבד"כ שמור למצביע ה-bk של chunk משוחרר, תא זה מתוייג מאותו רגע כשדה בשם "key" (מפתח מזהה של כתובת מבנה ה-tcache שנמצא בתחילת ה-Heap). בהמשך נראה שהחל מגרסה 2.34 במקום אותה כתובת נשמר ערך אקראי בשדה ה-key). כאשר chunks משוחררים – שדה המפתח שלהם נבדק. במידה והוא תואם לכתובת של ה-tcachebin, יבוצע חיפוש בלולאה בתוך ה-tcachebin הרלוונטי אחר ה-chunk שמשחררים. אם ה-chunk מתברר כ"נמצא כבר" (מתבצעת השוואת כתובות, ולא של מפתחות) בתוך ה-tcache אז הפונ' abort() נקראת.

ניתן לעקוף בדיקה זו ע"י הצפה (למלא עד הסוף) של ה-tcachebin המיועד כדי שלאחר מכן נוכל לשחרר victim chunk ישירות אל תוך fastbin בגודל זהה. לאחר מכן נוכל לרוקן את ה-tcachebin (נצטרך לבצע הקצאות ברצף מספר פעמים שהינו גודל ה-tcache, לדוגמה 7), ואז נשחרר את ה-victim פעם שנייה (יקושר ל-tcachebin). כעת, ה-victim chunk יוקצה מתוך ה-tcachebin, ובנקודת זמן זו (בהנחה למשל שהתוכנית מאפשרת לכתוב ל-user data מיד לאחר ההקצאה שלו) המעצב יכול לשנות את שדה ה-fd שלו מבחינת ה-fastbin (תזכורת: יש העתק של ה-chunk ב-fastbin!. והרי לאן נרצה לגרום לו להצביע? אל chunk מזוייף שנרצה לכתוב לתוכו, לדוגמה: כזה המכיל את אחד מה-hooks).

כאשר נקצה את ה-victim chunk מתוך ה-fastbin בו הוא נמצא, תתרחש פעולה מיוחדת – כל שאר ה-chunks שנותרו בתוך אותו ה-fastbin יועברו אל ה-tcache התואם (תהליך ה-dump), ובמקרה שלנו זה כולל את ה-chunk המזוייף. תהליך ה-tcache dumping לא כולל בדיקה לזיהוי שחרור-כפול (כלומר ניתן ליצור מצב בו chunk נמצא יותר מפעם אחת ב-tcachebin). שימו לב שה-fd של ה-chunk המזוייף חייב להיות null כדי שזה יצליח (כדי שתהליך ה-dump לא יגרור ניסיון לקרוא מכתובת שאין לנו גישה אליה ונקריס את התוכנית)

מכיוון שה-tcache עצמו שוכן בתוך ה-heap (בתחילתו), ניתן להשחית אותו לאחר שהשגנו זליגת מידע של כתובת מיקום ידוע על ה-heap (heap leak).

בקורס המוזכר בתחילת המאמר, היוצר שלו בחר להציג את ה-tcache בשלב מאוחר יחסית (חלק 2) וטוען שלהכנסת מנגנון זה ל-glibc היו השלכות שליליות מבחינת האבטחה שלה (כלומר- זה הופך את תהליך האקספלוטציה לקל יותר)

המעבר לגרסה GLIBC 2.32 והכנסת מנגנון Safe-Linking (2020, Eyal Itkin)

מנגנון האבטחה "Safe-Linking" (לא התיקון שהוכנס בשנת 2005 לגרסה 2.3.6 של glibc בשם "Safe-Unlinking" כדי להגן על ה-bins שעובדים עם רשימות דו-כיווניות) נועד להגן על רשימות מקושרות חד-כיווניות של malloc מפני שיבוש (tampering) מצד תוקף. הוא הוכנס ל-glibc (Linux) ולספרייה מקבילה לתחום ה-Embedded בשם uClib-NG. לפני שהוכנס המנגנון – השחתה של רשימות חד-כיווניות (כמו זו של ה-Fastbins וה-Tcache) אפשרה לתוקף להשיג פרימיטיב מסוג הקצאה שרירותית, כלומר הקצאה קטנה של כתובת שרירותית בזיכרון בשליטת התוקף (מקום אליו הוא יכול לכתוב מידע לפני ההקצאה).

על פי הפרסום המקורי מהבלוג Check Point research מאת יוצר המנגנון, אייל איטקין, שהוביל להכנסת הגנה זו ועליו מתבסס חלק זה: קישור-בטוח עושה שימוש באקראיות של מנגנון ה-ASLR, שנעשה בו שימוש רחב במרבית מערכות ההפעלה המודרניות, כדי "לחתום" על מצביעי הרשימה. כאשר משלבים אותה עם בדיקת הישור של ה-chunk, הטכניקה החדשה מגנה על המצביעים מפני נסיונות השתלטות (hijacking) על המצביעים.

פתרון זה נועד להגן בפני שלוש מתקפות הנמצאות בשימוש תדיר ב-Exploits באותה תקופה:

1. דריסה חלקית של מצביע: שינוי הבתים הנמוכים של מצביע (Little Endian)
2. דריסה מלאה של מצביע: השתלטות על מצביע לכתובת שרירותית.
3. Chunks לא מיושרים: הצבעה מתוך הרשימה (יצירת קישור אל) כתובת לא מיושרת (כמו שעשינו למשל ב-Fastbin dup attack)

מידול האיום

לתוקף יש את היכולות הבאות:

- גלישת חוצץ לינארית (רצופה) מבוקרת מסוג overflow / underflow על גבי חוצץ ב-heap
- כתיבה לכתובת שרירותית יחסית לחוצץ ב-heap

חשוב לשים לב שהתוקף הנ"ל לא יודע את מיקום ה-heap, וזאת כי כתובת הבסיס של ה-heap נבחרת אקראית יחד עם כתובת הבסיס של אזור ה-mmap (mmap_base) ע"י מנגנון ה-ASLR

הפתרון שהוצע נועד להעלות את רמת המאמץ שתידרש מתוקף כדי שיצליח לפתח exploit מבוסס heap. מרגע שההגנה הופעלה, התוקף חייב שתהיה לו יכולת נוספת מהצורה של זליגת כתובת heap / כתובת מצביע(תיכף נבין למה). תרחיש לדוגמה עבור הגנה שכזו הוא בינארי תלוי-מיקום (נטען ללא ASLR. האמת היא שיש הבדל בין PIE ו-ASLR מבחינת האזורים המושפעים בזיכרון) הסובל מבאג גלישת חוצץ ב-heap כאשר מבוצע ניתוח של קלט המשתמש. (אותו מקרה שהוצג בפרסום מחקר קודם לכן)

עד אותה עת, תוקף היה מסוגל לנצל מטרות שכאלו מבלי שיזדקק להזלגה של כתובת מ-heap, ועם שליטה מינמלית על ה-chunks המוקצים שלו ע"י הסתמכות על כתובות קבועות בלבד בבינארי. ניתן לחסום ניסיונות ניצול שכאלו ולמנף את מנגנון ה-ASLR כדי להשיג אקראיות במהלך ניתוב מחדש של הקצאות זיכרון אל כתובות בקבצים הבינארים שמהווים מטרות פגיעות. (*)

ההגנה

במכונות לינוקס, ה-heap מקבל כתובת טעינה אקראית דרך mmap_base ע"פ הלוגיקה הבאה (ככל הנראה מדובר ב-psuedocode, לא הצלחתי לאתר שורה שכזו בקוד המקור של GLIBC)

random_base = ((1 << rndbits) - 1) << PAGE_SHIFT) <---------- ASLR Formula

rndbits – כברירת מחדל הינו 8 במערכות לינוקס 32 ביט, ו-28 במערכות 64 ביט).

נסמן את הכתובת שבה יושב מצביע של רשימה מקושרת חד-כיוונית ב-L. נגדיר את ה-Mask הבא שהוא בעצם הזה ימינה של הכתובת L כמספר ביטי ההיסט (מיקום מילה בתוך דף) שלא מושפעים מ-ASLR:

```
Mask := (L >> PAGE_SHIFT)
```

על פי נוסחת ה-ASLR הנ"ל, פעולת ההזזה ממקמת את הביט האקראי הראשון (מימין) מתוך כתובת הזיכרון בדיוק במיקום הביט הנמוך ביותר של ה-mask (או במילים פשוטות – ביט מספר 0, ה-LSbit)

מה שמוביל אותנו לסכימת ההגנה הבאה: נסמן את המצביע (המקורי, שהולך להיות מקושר אל תוך) של הרשימה המקושרת החד כיוונית ב-P. הסכימה נראית כך (פסאדו-קוד שיתורגם בסוף לקוד C)

```
PROTECT(P) := Mask XOR (P) = (L >> PAGE_SHIFT) XOR (P)
*L = PROTECT(P)
```

כלומר: במקום שתישמר ברשימה (ב-fd) כתובת המצביע המקורית, נשמור את תוצאת ה-XOR בינה לבין המסכה האקראית שחושבה.

בפורמט קוד C כפי שמופיע בקוד המקור בגרסה 2.34:

```
/* Safe-Linking:
   Use randomness from ASLR (mmap_base) to protect single-linked lists
   of Fast-Bins and TCache.  That is, mask the "next" pointers of the
   lists' chunks, and also perform allocation alignment checks on them.
   This mechanism reduces the risk of pointer hijacking, as was done with
   Safe-Unlinking in the double-linked lists of Small-Bins.
   It assumes a minimum page size of 4096 bytes (12 bits).  Systems with
   larger pages provide less entropy, although the pointer mangling
   still works.  */
#define PROTECT_PTR(pos, ptr, type)  \
        ((type)((((size_t)pos) >> PAGE_SHIFT) ^ ((size_t)ptr)))

#define REVEAL_PTR(ptr)  PROTECT_PTR (&ptr, ptr)
```

כאשר pos הוא כמו L, PAGE_SHIFT הוא 12 עבור סביבת 64 ביט (כלומר גודל דף הוא 2^12Bytes= 4KB, מה שאומר גם ש-3 הבתים הגבוהים של התוצאה לא מושפעים מ-ASLR ורק חמשת הבתים הנמוכים כן), ו-ptr הוא המצביע המקורי מתוך הרשימה המקושרת החד כיוונית (P).

פעולת הגילוי ("ההופכית" להגנה) עושה שימוש בתכונה פעולת ה-XOR: A ^ B = C ⇔ A ^ C = B כאשר &ptr = L . הסיבה שצריך אותה היא כדי שה-bins יוכלו להתנהג כמו לפני התיקון כמובן (לגשת ל-chunk הבא ברשימה וכו'). בצורה זו, הביטים האקראיים של הכתובת L משפיעים על הביטים הנמוכים של המצביע המאובטח, כפי שניתן לראות בדוגמה שבאיור הבא.

```
P  := 0x0000BA9876543210
L  := 0x0000BA9876543180

P                  =   0x0000BA9876543210
  ⊕                  ⊕
    L >> 12        =   0x0000000BA9876543
                       ----------------------
P'  := P⊕(L >> 12)     = 0x0000BA93DFD35753
```

איור 24 Safe-Linking: Eliminating a 20 year-old malloc() exploit primitive (From Check Point research blog)

הסבר לאיור: המצביע הממוסך (המאובטח) מכוסה בביטים אקראיים (מופיעים באדום, השאר ידועים) שכבת הגנה זו מונעת מתוקף (או לכל הפחות אמורה) לעשות שינויים במצביע ללא ידע מקדים על הביטים האדומים האקראיים (ביטי ה-ASLR). בזמן שהתוקף לא יכול להשתלט על המצביע, גם אנחנו נהיה מוגבלים משום שאנו לא יכולים לבדוק האם התרחש שינוי של המצביע. וכאן נכנסת בדיקה נוספת:

כל ה-chunks המוקצים על ה-heap מיושרים להיסט קבוע ידוע שהינו בד"כ 8 בתים במערכות 32 ביט, ו-16 בתים במערכות 64. אם נוודא שכל מצביע, לאחר שנפעיל עליו את reveal (גילוי/חשיפה של P המקורי), יהיה מיושר בהתאם אז יתווספו לנו שתי שכבות הגנה חשובות:

- תוקפים חייבים לנחש נכון את ביטי היישור.
- תוקפים לא יכולים לגרום ל-chunks להצביע לכתובות זיכרון לא מיושרות.

במכונות 64 ביט, ההגנה הסטטיסטית גורמת לניסיון התקפה להיכשל 15 מתוך 16 פעמים (למה?). אם נחזור לאיור הקודם, נראה שהערך של המצביע המאובטח מסתיים ב-0x3 nibble, פירוש הדבר שתוקף חייב להשתמש בערך 0x3 בגלישה שלו, כי אחרת הוא ישחית את הערך ויכשיל את בדיקת היישור. (גם לא ברור מדוע)

אפילו כשלעצמה, בדיקת ישור זו מוענת שימוש ב-primitives ידועים לצרכי אקספלוטציה כמו זה שגורם לקישור ה-malloc_hook אל ה-fastbin לצורך השגת יכולה הרצת קוד.

בדיקה חוזרת של מודל האיום

בדיקת הישור מקטינה את משטח התקיפה ודורשת ש-chunk ב-tcache או ב-fastbin יהיה חייב להצביע על כתובת זיכרון מיושרת. זה חוסם באופן ישיר מגוון שיטות אקספלוטציה שהוזכרו קודם לכן.

בדיוק כמו במנגנון ה-Safe-Unlink (שפועל על הרשימות המקושרות הדו-כיווניות), הגנה זו נשענת על העבודה שהתוקף לא יודע איך נראים מצביעי heap לגיטימיים.

במקרה של רשימות מקושרות דו-כיווניות, תוקף המסוגל לזייף מבנה זיכרון ויודע כיצד נראה מצביע heap תקין, יוכל לזייף גם זוג מצביעי FD/BK תקינים שאמנם לא יתנו לו יכולת כתיבה לכתובת שרירותית, אך יאפשרו לו לקשר chunk הנמצא בכתובת בשליטתו.

במקרה של רשימות מקושרות חד-כיווניות,, לתוקף שאין ברשותו זליגת כתובת מצביע, לא יוכל לשלוט באופן מלא על מצביע שנדרס עקב שכבת ההגנה שמסתמכת על האקראיות הנרכשת מפעולת ה-ASLR שפותחה כאן. הקבוע PAGE_SHIFT ממקם את הביטים האקראיים החל מהביט הנמוך ביותר של המצביע ששומרים. כאשר משלבים אותה עם בדיקת היישור, הסתברותית, נמנע מן התוקף לשנות אפילו רק את הביט/בית התחתון (Little endian) של המצביע המאוחסן ב-data של הרשימה המקושרת החד-כיוונית

סיכום

באותו פרסום, נכתב כהערת סיכום: הגנת ה-Safe-Linking איננה פתרון קסם שיעצור את כל נסיונות ניצול החולשות כנגד מימושי heap מודרניים. יחד עם זאת, זהו צעד נוסף בכיוון הנכון. ע"י הכרחת התוקף להצטייד בחולשת הזלגת מצביע לפני שהוא יכול אפילו להתחיל את מתקפה מבוססת heap, המגנים מעלים בהדרגה את הרף שיצטרכו להתמודד איתו התוקפים.

הגנה זו מהווה אחד מההישגים המשמעותיים ביותר מבחינת אפקטיביות שמומשה ב-GLIBC לאחרונה. זמן מה לאחר מכן, באופן לא מפתיע, חוקרים גילו שניתן לעקוף הגנה זו מבלי להזדקק להזלגת כתובת מה-Heap!. נתאר ונדגים זאת בטכניקות הבאות במסמך. [לשבור את ההגנה במצב בו יש לנו heap leak נחשב לעניין פשוט יחסית]

חשוב לציין שלא כל חולשה הניתנת לניצול בקוד של glibc לצרכים זדוניים אשר מדווחת בהכרח תוביל לתיקון הממצאים מיידית/בכלל (אפילו אם מוצע לה patch תיקון). התהליך כרוך (כראוי לפרויקט כה מורכב וחשוב) בבירוקרטיה הנופלת לעיתים על מוצא החולשה במקרה של תיקון מספר רב של שורות קוד. ויכול להיות שרק מספר מצומצם של מתחזקים אחראי להכנסת המיטיגציות לקוד (שימו לב לטבלת המיטיגציות שבמסמך זה). מה שאומר שניתן למצוא חורי אבטחה בדיווחים/הצעות ישנות המופיעות באתר הפרויקט או במערכת למעקב אחרי באגים שלו (אם יודעים מה ואיך לחפש, אפשר גם לנסות להגיע לממצאים כאלו דרך גוגל או מנועי חיפוש אחרים. באופן היפותטי, יתכן שישנם גופים מסויימים שיעדיפו לבצע את החיפושים האלו ברשת בצורה אנונימית, ובמקרה של אלו שלא יכולים להרשות לעצמם את הסיכון שאחרים ידעו שהם או מישהו אנונימי אחר בעלי עניין בוקטור תקיפה כזה או אחר, אולי יעדיפו לאתר את החולשות בעצמם, ומן הסתם גם לא לוודח עליהן)

לסיום, זוהי הזדמנות טובה להגיד כמה מילים טובות על אייל איטקין. אייל נחשב לאחד מחוקרי אבטחת המידע הבולטים בארץ ומחוץ לה, בין השאר תרם למגזין זה מספר רב של מאמרים על מחקריו ומדריכים. אייל פרסם לפני מספר חודשים שהוא מחליף עיסוק ועובר להתמקד בעולם התכנות. נאחל לו בהצלחה בדרכו החדשה.

House of IO: Remastered (2020, Awarau & Eyal Itkin)

סקירה כללית

השחתת ה-metadata של מבנה ניהול ה-tcache (של ה-main arena) שמוצבע מתוך chunks ששוחררו וקישורו אל תוך ה-tcache

פרטים

נרצה למנף את התכנון של ה-Tcache בו נשמר metadata על גבי שדה ה-user-data של ה-chunks לתועלתנו. את המבנה הכללי וצורת המימוש ה-Tcache אנו מכירים כבר מהחלק בו הצגנו אותו. בנוסף לכך, נשים לב שמנגנון ה-Safe-Linking מפעיל את מאקרו ה-()PROTECT רק על מצביעי next/fd. כלומר,

הראש של כל אחת מהרשימות נשאר לא מוגן. כמו במקרה של ה-fastbins, גם הם נמצאים באזור ה-heap (המצביע למבנה שבתוכו נשמרים המצביעים לראשי רשימות ה-tcachebins: static __thread *tcache tcache_perthread_struct שמור במשתנה סטטי הנגיש רק לפונקציות שבקובץ בו הוא מוגדר, אך המבנה עצמו מוקצה בתחילת ה-heap). בכל מקרה – זה תלוי מימוש. לא נוכל להיות בטוחים בוודאות שמיסוך (bitwise xor) המצביע עם הכתובת בה הוא נשמר יהיה אפקטיבי (עקב כך שזה תלוי במימוש של שאר הקוד).

השילוב של תכן לא עמיד מצד Glibc והנחה לא נכונה של אייל כאשר עיצב את המנגנון, איפשרו את ניצול טכניקה זו. נתקוף ישירות את ראש רשימת ה-free list של אחד מה-tcachebins.

מבנה ה-tcache_perthread_struct מוקצה בעת שה-heap נוצר (למשל- בעת הבקשה הראשונה לזכרון מה-heap), הוא יאוחסן בתחילת ה-heap (בהיסט 0). תוקף עם יכולת buffer underflow לינארית על גבי חוצץ ב-heap או יכולת מסוג כתיבה-שרירותית לכתובת יחסית (כלומר יכול לכתוב בהיסט/אינדקס מערך שלילי ביחס לכתובת על ה-heap, יהיה מסוגל למנף זאת לצורך השחתת מבנה ה-tcache הנ"ל כולו. בפרט, הוא יוכל להשחית כל tcache_entry שיבחר (שם יושב המצביע לראש הרשימה של אותו bin).

שימושים נוספים

בנוסף לכך, יש עוד 3 מקרי קצה שעשויים להיות מנוצלים ע"י תוקף. ניתן לסווג אותם כתת-קבוצה של כתיבת לכתובת שרירותית יחסית:

1. UAF – המאפשר קריאת כתובת של מבנה ה-Tcache השמורה במצביע וכתיבתה לכניסה השומרת מצביע בהיסט של 8 בתים (אחורנית מאותה כתובת) - הכתובת שקוראים שמורה בתוך מיקום שדה ה-key (ה-quadword (המצביע) השני) ב-user data (*tcache_perthread_struct) ולאחר מכן כתיבה אל מבנה ניהול זה הנמצא ב-main_arena. [key נכתב למיקום הנ"ל בעת קישור free chunk ל-tcache בפונקציה tcache_put()]. נחשב את tc_idx הרלוונטי (המתאים לגודל ה-victim chunk שברצוננו לקשר אל ראש רשימה מסויימת) ואת ההיסט מהכתובת שחשפנו אל tcache->entries[tc_idx] ו-tcache->counts[tc_idx] . נשנה את ערך המונה המתאים לערך חיובי המספק אותנו.נכתוב לתוך tcache->entries[tc_idx] את כתובת ה-target data שלנו (תזכורת: ב-metadata של ה-tcache chunk נשמרת הכתובת של ה-**user data** של ה-chunk הבא, ולא של תחילת ה-chunk). נקצה chunk מתוך אותו tcachebin וכעת נוכל לכתוב אל ה-target data שלנו באמצעות ה-victim chunk.
2. שחרור מבנה הניהול של ה-Tcache: סט קריאות ל-free() בסדר משובש – לדוגמה: עבור באג בקוד של שחרור מבנה המכיל מצביעים לזיכרון דינמי לפני ש-free() נקרא על ה-members שלו שצריך גם כן לשחרר אותם. לאחר מכן נשחרר את המצביע למבנה ה-tcache המוזכר ב-1 (למשל דרך member שטרם ביצענו עליו free() שחופף למיקום שדה ה-key), כך נקשר את ה-

tcache_perthread_struct אל ה-tcachebin המתאים לגודל 0x290 (= 64*8+64*(16/2) 0x280bytes, 8 more bytes needed for size field => 0x290). בשלב זה נשלח בקשה להקצאה בגודל מתאים (כל ערך בין 0x277 ל-0x288 אמור לעבוד) ובכך נשיג יכולת כתיבה על גבי מבנה ה-tcache ובפרט לצורך דריסת ערך של מצביע לראש רשימה. יחד עם זאת, עקב מגבלות אפשריות על גודל הבקשות העוקבות שניתן להעביר ל-()malloc, זהו תרחיש הרבה פחות סביר לעומת buffer underflow או Use-After-Free.

3. Buffer underflow: השחתת מבנה ה-tcache_perthread_struct ע"י גלישה אחורנית מחוצץ הנמצא בסמוך אליו על ה-heap (למשל על ידי שימוש באינדקס שלילי בגישה למערך).

ראינו שלתוקף עם אותן היכולות שהגנת ה-Safe-Linking תוכננה להגן בפניהן, יהיה מסוגל לעקוף אותה ולתקוף ישירות את מבנה הניהול הראשי של ה-tcache ובכך להשיג פרימיטיב מסוג Malloc-Where (להקצות chunk בכתובת שרירותית). אם תוקף יכול לדרוס את המצביע של ראש tcachebin כלשהו, הוא עשוי להצליח לעקוף את מכשול האקראיות שהמנגנון הנ"ל משתמש בו כדי להגן על מצביעי ה-next/fd.

מגבלות

המתקפה עובדת כנגד ה-tcache, אך היא לא יודעת להתמודד עם הבדיקה של ה-fastbins

המתקפה מצריכה מהתוקף שיהיה לו לפחות אחד מהשלושה: פרימיטיב גלישה אחורנית, UAF בהיסט ספציפי מתחילתו של מבנה על ה-heap או פרימיטיב המאפשר לקשר את ה-tcache_perthread_struct אל תוך ה-tcachebin היחיד שמתאים לו. וזוהי נקודה קריטית: מבחינה סטטיסטית, באג Underflow הוא הרבה פחות נפוץ מאשר באג Overflow. בנוסף לכך, על מנת להפוך את גרסאות ה-()free של מתקפה זו ליעילות בעולם האמיתי, יש צורך ביכולות כמו UAF על מצביע, המוכל בתוך המבנה ששוחרר, ונמצא בתוכו בהיסט של 8 בתים, באג בסדר הקריאות ל-()free על מבנה וה-members שלו, או אמצעי אחר בעזרתו קריאה ל-()free על מצביע תגרום לשחרור ה-tcache_perthread_struct .

גרסה 2.34 של GLIBC הציגה לראשונה מנגנון לקביעת ערכים שונים ל-key העושה שימוש במשתנה סטטי *static uintptr_t tcache_key* שנועד לתפוס נסיונות לשחרור-כפול בתוך thread, המאותחל בצורה אקראית ומוגרל לערך חדש בכל הכנסה ל-tcache (בעזרת מחולל מספרים רנדומליים שאינו בהכרח בטוח קריפטוגרפית) שמטרתו להחליף את המפתח שעד כה היה משותף בין כל ה-free-chunks שבתוך tcachebin מסויים (זיכרו שהמטרה שלו הייתה לזהות מי כבר נמצא בתוך אותו tcachebin, והרי שלכל גודל יכול להתאים רק tcachebin אחד. לכן היה מספיק טוב לשמור בשדה ה-key של כל ה-free chunks שקושרו ל-tcache את אותו המפתח! כי זה רק נועד לסמן שהוא משוייך לשם.) אך כעת מישהו נזכר שזוהי חולשה/באג בתכן (design) הניתנת לניצול. המנגנון החדש תוכנן כך שיגריל מספר יחודי שלא אמור להופיע בזיכרון עבור כל chunk המשתחרר אל ה-tcache כדי להשיג עוד מטרה חוץ מהקודמת (לזהות שחרור כפול): למנוע הזלגה של כתובת מבנה ה-tcache (אבל כמו שאומרים – לא לעולם חוסן...).

(c4ebt, 2020) House of Rust

סקירה כללית

עקיפת הגנת ה-Safe-Linking של GLIBC v2.32 ללא הזלגת מידע והשגת יכולת הרצת קוד (dropping a shell) כנגד בינארי המקומפל עם PIE ולא מזליג שום כתובת.

סיווג

טכניקה זו ממנפת באג מסוג Use-After-Free כדי לבצע מספר מתקפות ידועות שהשילוב שלהן מאפשר להתגבר על בדיקת ה-Safe linking מבלי להזדקק להזלגת כתובות. החוליה החלשה אליה היא מכוונת כדי לבצע זאת ביעילות היא מנגנון (מכניזם) ה-Tcache stashing (החבאה). היא משתמשת ב-Heap Feng Shui, Tcache statshing Unlink + attack, Tcache stashing אחד, מתקפת Unlink, שתי מתקפות largebin ומסתיימת במתקפה מסוג FSOP על ה-File stream של stdout.

פרטים

כמו ב-House of IO, גם כן נשתמש במבנה ה-tcache_perthread_struct, אבל פשוט כי הוא החוליה השימושית החלשה ביותר במימוש של Safe-Linking, ועל כן ייתן לנו exploit "נקי יותר". הפרימיטיבים שטכניקת ה-House of Rust נותנת לנו מאפשרים לנו לתקוף ישירות את ה-File stream של stdout מבלי שנצטרך לגעת במבנה ניהול ה-tcache. מתקפה שכזו תדרוש מאיתנו מתקפה נוספת מסוג Tcache +stashing unlink ומתקפה על ה-largebin. **לכן העדיף את שיטת....** במובן זה, House of IO ו- House of Rust שונים לחלוטין, משום שהמטרות המרכזיות שלהם הן מנגנונים שונים (מבנה ניהול ה-tcache לעומת מנגנון ה-tcache stashing בהתאמה).

וריאציות ודרישות

הגרסה הנקייה של הטכניקה דורשת שליטה על ה-heap במקומות בהם לא מצבעים השמה של NULL למצביעים אחרי ששוחררו ובכך יוצרים מצבים של UAF. היא דורשה מספר הקצאות סביב 65, בגדלים של עד 0x1b00. רכיב ה-FSOP על ה-stdout בה דורש מהבינארי שלא יהיה line-bufferd או fully-buffered.

הגרסה בנוסח House of Corrosion- "House of Crust": הפרימיטיב המשמש לעקיפת מנגנון ה--Safe Linking ב-House of Rust יכול להיות ממונף לגרסה משוכללת המובילה למתקפה שמזכירה את House-of-Corrosion. גרסה זו כוללת את אותן דרישות של הגרסה הנקייה, פרט לכך שהיא זקוקה למספר גדול יותר של הקצאות וגדלי בקשות. היתרון שלה הוא שהיא לא מציבה אילוצים על סוג ה-buffering בבינארי, הודות לכך שהיא נשענת באופן מלא על כתיבות למיקום יחסי ולא מסמנת כמטרה את ה-stdout file stream.

שתי הוריאציות דורשות הפעלת bruteforce עם אנטרופיה של 1/16 לצורך ניחוש כתובת הבסיס של libc.

ניתן לחלק טכניקה זו ל-5 שלבים המתבצעים בסדר הבא:

i. Heap Feng Shui / Heap massaging – עיצוב מבנה ותוכן ה-heap כשלב מקדים
ii. Tcache Stashing Unlink+ (TSU+) יחד עם Largebin attack
iii. Tcache Stashing Unlink (TSU) יחד עם Largebin attack
iv. FSOP – stdout leak
v. השגת shell

<u>שלב 1</u> - Heap Feng Shui : המטרה היחידה של שלב זה היא להכין את ה-heap עבור השלבים הבאים. הקצאות הזיכרון הראשוניות שיבוצעו מה-heap נעשות כדי שבקשות הנעשות בשלב מאוחר יותר יקבלו זיכרון מתוך המיקום בו נמצאים chunks אלו ולא ממקומות אחרים.

<u>שלב 2</u> – TSU+ יחד עם Largebin attack:

שלב זה מתחיל בהקצאת 14 smallbin chunks בגודל 0x90 (זאת אחת מהדוגמאות המומלצות). 7 מתוכם ישוחררו בשלב מאוחר יותר ויקושריו ל-tcachebin 0x90, ו-7 הנותרים יקושרו אל ה-unsortedbin כאשר ישוחררו, לאחר מכן נגרום להם להתמיין מיידית אל תוך ה-smallbin 0x90. בנוסף לאלו, דרושות עוד 5 הקצאות לצורך Largebin attack (3 בגודל 0x20 לצרכי מניעת איחוד, ו-2 large chunks)

לפני שנתחיל להקצות 14 chunk-ים, נקצה שני large chunks כך שה-qword השני של ה-data ב-chunk השני יחפוף את שדה הגודל של ה-chunk ה-14 בגודל 0x90 (זיכרו שהגודל המינימלי עבור large chunk הוא 0x400. 7 = 0x90 / 0x400). נשחרר את שני ה-large chunks כך שיאוחדו עם ה-top chunk. ה-heap יחזור למצב כמעט כמו ההתחלתי (ישמרו בו שתי כתובות של ה-Unsortedbin). נחזור על צעד זה כך שה-qword ה-2 של ה-large chunk השני יחפוף לשדה ה-bk_nextsize של ה-large chunk הראשון (**????**). כעת נשחרר שוב את שני ה-large chunks. בכך השגנו יכולת לערוך את שדה הגודל של של chunk בגודל 0x90 ואת שדה ה-bk_nextsize של large chunk בעזרת באג UAF. נקרא ל-chunks אלו ה-WAF chunks (write after free). לאחר שמיקמנו את שני ה-WAF chunks כראוי, נקצה chunk בגודל 0x30 מתוך כוונה לשחרר אותו מאוחר יותר כדי שכתובתו תיכתב על המצביע של ראש ה-tcachebin 0x30 במבנה ה-tcache.

נשחרר את 14 ה-chunks הראשונים בסדר עולה אך באופן שזור כדי להימנע מאיחוד chunks, מה שיגרום ל-tcache להתמלא (7 chunks) ו-7 נוספים יגיעו ל-unsortedbin. נקצה large chunk כדי שה-small chunks שב-unsortedbin ימויינו אל ה-smallbin 0x90, נערוך את ה-WAF chunk הראשון כדי לשנות את שדה הגודל של ה-chunk ה-14 כך שה-large chunk הראשון ישורשר אליו (**גודל 0xb0 אם גודל ה-chunk המפריד הוא 0x20).** נשתמש בבאג ה-UAF כדי לשחרר את ה-chunk ה-14 בפעם השניה ובכך נקשר אותו אל ה-tcachebin 0xb0 וערך ה-key של ה-tcache ייכתב ל-qword השני של ה-data שלו. נשים לב ששחרור chunk זה אל תוך ה-tcache **משחית את שדות ה-fd וה-bk של ה-smallbin**. שדה ה-key מצביע לתחילת מבנה ניהול ה-tcache. נערוך את ה-chunk ה-14 כדי לשנות את בית ה-LSB של שדה ה-bk שלו ל-0x80, כך הוא יצביע **לכתובת גבוהה יותר במבנה ה-tcache** וליתר דיוק אל כתובת ראש רשימת ה-tcachebin שלו פחות 0x18. הנוכחות של מצביע ראש ה-tcachebin 0x30 חשובה כדי לקיים את האילוץ של כתובת הניתנת לכתיבה עבור מתקפת ה-TSU plus.

ברגע זה, ה-chunk ה-14 מויין אל תוך ה-smallbin שלו ושדה ה-bk שלו מבציע אל כתובת המטרה עבור מתקפת ה-TSU plus. הבעיה היא ששדה ה-fd שלו הושחת לחלוטין ויגרום לכשלון התוכנית במהלך ביצוע ה-TSU plus. זאת הסיבה לכך שמתקפת ה-TSU Plus נחשבה לכזו שתלויה בהזלגת מידע (עד לפיתוח שיטה זו)

שדה מצביע ה-fd שהושחת ניתן לתיקון בעזרת largebin attack. נכוון אותה אל ה-qword הראשון של ה-small chunk ה-14. על מנת לעשות זאת, נשחרר קודם את ה-large chunk הראשון ונמיין אותו מיד לאחר מכן ע"י בקשת chunk גדול יותר שנשחרר לאחר מכן. נדרוס את בית ה-LSB של שדה ה-bk_nextsize של ה-large chunk הראשון בעזרת ה-"WAF chunk" השני כך שהשדה יצביע על הכתובת של ה-chunk ה-14 פחות 0x10, במחיר של השחתת שדה ה-fd_nextsize שלו, אך זה לא משנה. לאחר מכן נשחרר את ה-large chunk השני, נבקש ונשחרר לאחר מכן large chunk גדול יותר כדי שהקודם ימויין אל תוך ה-largebin. בכך תיכתב כתובת ה-largebin "השני" אל שדה ה-fd של ה-chunk ה-14

נערוך את ה-large chunk השני: נשנה לו את בית ה-LSB בשדה ה-bk כך שיצביע אחורנית אל ה-small chunk ה-14. בכך סגרנו את שרשרת ה-smallbin, ונוכל כעת לבצע את מתקפת ה-TSU plus ביעילות ומבלי להקריס את התוכנית.

לבסוף, נעורר את מתקפת את ה-TSU plus. כדי לעשות זאת, נרוקן את ה-tcachebin 0x90 ע"י הקצאת 7 chunk-ים ממנו, לאחר מכן נקצה עוד chunk אחד אשר יוגש מתוך ה-smallbin ובכך יתחיל תהליך ה-stashing/dumping שיבצע את התקיפה. התוצאה: המצביע לראש רשימת ה-tcachbin 0x90 עבר להצביע על &heap+0x80 (הרי מבנה ה-tcache נמצא ממש בתחילת ה-heap) בנקודה זו – הבקשה הבאה ל-chunk בגודל 0x90 תוגש מתוך chunk החופף ל-tcache_perthread_struct .

שלב 3 – Tcache Stashing Unlink בתוספת Largebin attack: את ההכנות עבור שלב זה ביצענו בצורת הקצאות בתחילת ה-exploit בשלב ה-Heap Feng Shui על מנת להימנע ממיון לא רצוי של chunks או הקצאה שלהם ממקומות לא רצויים. מטרת שלב זה היא לכתוב כתובת של libc איפשהו בתוך מבנה ניהול ה-tcache במקום הקרוב ל-chunk האחרון שהוקצה בשלב 2. היתרונות של שימוש במתקפה מסוג TSU במקום +TSU בשלב זה היא שאין היא דורשת מאיתנו כתובת הניתנת לכתיבה במיקום השווה ל-&target+0x18.

נתחיל בהקצאה של 15 (במקום 14) chunks בגודל 0xa0 (ערך מומלץ ולא מחייב). באופן דומה לשלב 2 (+TSU), נקצה עוד 5 chunks (3 בגודל 0x20 לצורך חציצה + 2 large chunks). חשוב: הגדלים של שני ה-large chunks בשלב זה חייבים להיות ממופים ל-largebin שונה מזה שבו עשינו שימוש בשלב 1. אם ה-large chunks משלב 1 התאימו ל-largebin 0x400, אז נשתמש (למשל) ב-largebin 0x480 עבור שלב זה.

בדומה לשלב 3, נקצה large chunks ונשחרר אותם כדי לנצל באג WAF (נוכל לכתוב אליהם לאחר ששוחררו) כדי לערוך metadata קריטי של chunk המוכל בהם בשלב מאוחר יותר. נצטרך לערוך את אותו ה-metadata שהוזכר קודם לכן – שדה הגודל של ה-small chunk ה-15, ושדה ה-bk_nextsize של ה-large chunk הראשון.

כעת, כמו בשלב 2, נשחרר את 15 ה-small chunks בסדר עולה אך באופן שזור כדי להימנע מאיחוד chunks, מה שיגרום ל-tcache להתמלא (7 chunks) ו-8 נוספים יגיעו ל-unsortedbin. לאחר מכן, נקצה large chunk כדי שה-unsorted chunks ימויינו אל תוך ה-smallbin 0xa0. מרגע זה אופן הפועלה כמעט זהה לשלב 2, מלבד כך שה-LSB שנכתוב לשדה ה-key של ה-tcache chunk יגרום להצבעה אל "קצת יותר גבוה" במבנה ניהול ה-tcache (tcache_perthread_struct).

בכך השגנו כתיבה של כתובת libc על גבי מבנה ה-tcache בסמוך ל-chunk האחרון שהוקצה בשלב 2.

שלב 4 – הזלגת כתובת libc (שכתבנו בשלב 3 למבנה ניהול ה-tcache) בעזרת טכניקת FSOP על stdout:

נתחיל בעריכת ה-chunk שהקצנו על ה-tcache_perthread_struct כדי לדרוס את 2 בתי ה-LSB של כתובת ה-libc, שנכתבה בשלב 3, על מנת לגרום לה להצביע

כדי לבצע את טכניקת ה-FSOP הזו, נדרוס את שדה ה-`_IO_2_1_stdout_._flags` עם הערך 0xfbad1800. נאפס את שלושת ה-qwords העוקבים ששייכים לשדות _IO_read_ptr, _IO_read_end ו-_IO_read_base. ב-qword הבא השייך ל-_IO_write_base נאפס את בית ה-LSB. בפעם הבאה שתבוצע פעולה על ה-stdout file stream , נקבל זליגת מידע גדולה. שימו לב שאת כל השדות האלו צריך לדרוש בבת אחת, אחרת תיתכן התנהגות בלתי צפויה.

שלב 5 – השגת shell: לאחר שהזלגנו כתובת של libc, נרצה לבצע כתיבה לתוך מבנה ניהול ה-tcache (tcache_perthread_struct), נדרוס את אחד מראשי רשימות ה-tcachebins עם הכתובת של ה-__free_hook. נקצה chunk מאותו tcachebin שערכנו ונזין לתוך ה-data שלו את הכתובת של system().
כעת כל שנותר הוא לערוך chunk כלשהו כך שב-qword הראשון של ה-data שלו ייכתב "bin/sh\0" ולבסוף לשחרר אותו עם free ונקבל shell

פתרונות אתגרים לדוגמה מתחרויות CTF:

CorCTF Helpless:

נשים לב שב-bss. נמצא מערך בשם chunks בגודל 100 שמסוגל לאחסן מצביעים (qwadwords) ל-chunks, וכמו כן מערך בשם sizes שגם הוא בגודל 100 ואחראי לשמירת הגדלים של כל אחד מה-chunks שהמשתמש קיבל.

הפונצקיה create(): מבקשת מהמשתמש אינדקס ותפעל רק אם הוא בין 0 ל-99, לאחר מכן תבקש מאיתנו "אורך מפתח" שזה הוא בעצם גודל הבקשה (משתנה בשם size) שהיא תעביר ל-malloc, והוא יכול להיות עד גודל 0x2000 כולל. במידה וההקצאה מצליחה, המצביע ל-chunk נשמר במערך chunks באינדקס המבוקש (אמנם אין בדיקה שהאינדקס שביקשנו איננו תפוס כבר, אבל התועלת היחידה בכך היא האפשרות להקצות יותר מ-100 chunks במחיר של לאבד גישה לחלק מהם). לבסוף היא קוראת מהמשתמש עד size בתים אל תוך ה-data של ה-chunk שהוקצה דינמית.

הפונקציה delete(): מבקשת מהמשתמש אינדקס (תומכת בערך בין 0 ל-99), במידה ותא זה במערך chunks מצביע על chunk כלשהו, היא תשחרר אותו אך לא תאפס את המצביע ל-null! (באג UAF). כמו כן לא מתבצע איפוס של הערך המתאים במערך ה-sizes.

הפונקציה edit(): מבקשת מהמשתמש אינדקס (תומכת בערך בין 0 ל-99), בודקת את הערך של sizes[index], אם ערך זה שונה מאפס – היא קוראת מהמשתמש עד sizes[index] בתים אל ה-data של אותו chunk.

נסכם את היכולות שהבאגים הנ"ל בקוד מספקים לנו: עריכת מידע של chunk לאחר ששוחרר (WAF), ובאופן תיאורטי גם double free (אם נצליח לעבור את כל ההגנות מפניו דוגמת:

Frame #5 malloc_printerr (str=str@entry=0x7ffff7f7b978 "free(): double free detected in tcache 2") at malloc.c:5543

בשלבי debug, נרצה להריץ את הסקריפט עם הפקודה: python3 xpl.py GDB NOASLR

(או במקרה של vim,

לאחר שתי ההקצאות הראשונות: תכולת ה-heap תיראה כך:

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x55555555b000
Size: 0x291

Allocated chunk | PREV_INUSE
Addr: 0x55555555b290
Size: 0x941

Allocated chunk | PREV_INUSE
Addr: 0x55555555bbd0
Size: 0x511

Top chunk | PREV_INUSE
Addr: 0x55555555c0e0
Size: 0x1ff21
```

נשים לב מה קרה ל-large chunk הראשון לאחר שחרורו :

```
0x55555555b280  0x0000000000000000      0x0000000000000000      ................
0x55555555b290  0x0000000000000000      0x0000000000000941      ........A.......          <-- unsortedbin[all][0]
0x55555555b2a0  0x00007ffff7fb4cc0      0x00007ffff7fb4cc0      .L.......L......
0x55555555b2b0  0x0000000000000000      0x0000000000000000      ................
```

לאחר שנשחרר גם את ה-large chunk השני, הוא יאוחד אחורנית עם הראשון וקדמית עם ה-top chunk.

כך שנקבל את המצג הבא:

```
0x55555555b280  0x0000000000000000      0x0000000000000000      ................
0x55555555b290  0x0000000000000000      0x0000000000020d71      ........q.......          <-- Top chunk
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x55555555b000
Size: 0x291

Top chunk | PREV_INUSE
Addr: 0x55555555b290
Size: 0x20d71
```

נשים לב שלאחר ה-top chunk, במקום בו היה ה-large chunk הראשון, עדיין נמצאות הכתובות של ה-unsortedbin (fd ו-bk). בכל מקרה יש לנו עדיין אפשרות לערוך את שני ה-large chunks הודות לבאג ה-WAF.

```
pwndbg> dq 0x55555555b290 200
000055555555b290     0000000000000000 0000000000020d71
000055555555b2a0     00007ffff7fb4cc0 00007ffff7fb4cc0
000055555555b2b0     0000000000000000 0000000000000000
```

נשים לב מה קורה ל-heap לאחר ההקצאה הבאה:

```
0x55555555b270  0x0000000000000000      0x0000000000000000      ................
0x55555555b280  0x0000000000000000      0x0000000000000000      ................
0x55555555b290  0x0000000000000000      0x0000000000001f51      ........Q.......
0x55555555b2a0  0x00007ffff7fb4c0a      0x00007ffff7fb4cc0      .L.......L......
0x55555555b2b0  0x0000000000000000      0x0000000000000000      ................
0x55555555b2c0  0x0000000000000000      0x0000000000000000      ................
```

בית ה-LSB של שדה ה-fd של ה-large chunk שלאחר ה-tcache מקבל את קוד האסקיי של newline char (זה המידע הקצר ביותר שחייבים לכתוב)

ההקצאה שלאחר מכן תיצור את ה-large chunk הבא:

```
0x55555555d1d0  0x0000000000000000      0x0000000000000000      ................
0x55555555d1e0  0x0000000000000000      0x0000000000000511      ................
0x55555555d1f0  0x0000000066647361      0x0000000000000000      asdf............
0x55555555d200  0x0000000000000000      0x0000000000000000      ................
0x55555555d210  0x0000000000000000      0x0000000000000000      ................
```

```
0x55555555b280  0x0000000000000000      0x0000000000000000      ................
0x55555555b290  0x0000000000000000      0x0000000000000091      ................
0x55555555b2a0  0x00007ffff7fb4c0a      0x00007ffff7fb4cc0      .L.......L......
0x55555555b2b0  0x0000000000000000      0x0000000000000000      ................
```

לאחר שורות אלו, לא נוכל לראות את כל ה-heap בעזרת הפקודה vis

```
for i in range(14):
    alloc(i+1, 0x88)
```

```
0x55555555d1c0  0x000000000000000a  0x0000000000000000  ................
0x55555555d1d0  0x0000000000000000  0x0000000000000451  ........Q.......
0x55555555d1e0  0x0000000000001f0a  0x0000000000000510  ................
0x55555555d1f0  0x0000000066647361  0x0000000000000000  asdf............
0x55555555d200  0x0000000000000000  0x0000000000000000  ................
```

```
0x55555555bb90  0x0000000000000000  0x0000000000000000  ................
0x55555555bba0  0x0000000000000000  0x0000000000000021  ........!.......
0x55555555bbb0  0x000000000000000a  0x0000000000000000  ................
0x55555555bbc0  0x0000000000000000  0x0000000000000431  ........1.......
0x55555555bbd0  0x000000000000090a  0x0000000000000510  ................
0x55555555bbe0  0x0000000066647361  0x0000000000000000  asdf............
```

לאחר שנשחרר את chunk ה-first_small:

```
0x55555555b280  0x0000000000000000  0x0000000000000000  ................
0x55555555b290  0x0000000000000000  0x0000000000000091  ................   <-- unsortedbin[all][0]
0x55555555b2a0  0x000055555555bb10  0x00007ffff7fb4cc0  ..UUUU...L......
0x55555555b2b0  0x0000000000000000  0x0000000000000000  ................
0x55555555b2c0  0x0000000000000000  0x0000000000000000  ................
0x55555555b2d0  0x0000000000000000  0x0000000000000000  ................
0x55555555b2e0  0x0000000000000000  0x0000000000000000  ................
0x55555555b2f0  0x0000000000000000  0x0000000000000000  ................
0x55555555b300  0x0000000000000000  0x0000000000000000  ................
0x55555555b310  0x0000000000000000  0x0000000000000000  ................
0x55555555b320  0x0000000000000090  0x00000000000000a0  ................
0x55555555b330  0x0000000000000000  0x0000000000000091  ................
```

נספח א': מקורות לימוד ותרגול נוספים + כלים שהוזכרו

כלים

http://pwndbg.com/ + https://browserpwndbg.readthedocs.io/en/docs/

http://pwntools.com/ + https://docs.pwntools.com/en/stable/

https://github.com/io12/pwninit - Automate starting binary exploit challenges

https://github.com/shift-crops/sc_expwn - pwntools extension for advanced users

Binary Exploitation

https://bitvijays.github.io/LFC-BinaryExploitation.html - סיכום נחמד הכולל גם היבטי קימפול

A First Introduction to System Exploitation With Georgia Tech's "pwnable" challenges, Ben Herzog

https://guyinatuxedo.github.io/

אתרים עם תשתית תרגול להשתלטות על מכונה מרחוק:

https://www.root-me.org/en/Challenges/App-System/ + helper script

https://www.hackthebox.eu/ (Challenges, Boxes – PT/RedTeam יכולות דורש במתקפה 2-ו 1 חלק)

https://www.pwnable.kr

https://www.pwnable.tw

סקירה של טכניקות Heap exploitation עם Proof of concept ו/או אתגרים

https://github.com/shellphish/how2heap

https://0x3f97.github.io/category/#/how2heap - (הסברים. Use www.deepl.com/translator)

pearcechn.top [PWN notes] how2heap analysis_1+2+3 (הסברים. Use www.deepl.com/translator)

https://guyinatuxedo.github.io/25-heap/

סקירה כללית ומבוא ל-heap exploitation

https://sourceware.org/glibc/wiki/MallocInternals

https://heap-exploitation.dhavalkapil.com/

https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/

https://azeria-labs.com/heap-exploitation-part-2-glibc-heap-free-bins/

AirGap2020.10: Modern Linux Heap Exploitation - Dr. Silvio Cesare

https://hackliza.gal/en/posts/r2heap/ - Heap analysis with radare2

אנליזה של קוד המקור

Overview of malloc and free in glibc - (Use www.deepl.com/translator)

Analysis of GLIBC 2.33 source code (ptmalloc) - (Use www.deepl.com/translator)

https://raw.githubusercontent.com/cloudburst/libheap/master/heap.png - Algorithm flowcharts

Unlink macro exploitation

LiveOverflow - The Heap: dlmalloc unlink() exploit - bin 0x18

GLibC Malloc for Exploiters - Yannay Livneh – Insomni'hack

היכל הבידור (Heap Exploitation against GlibC in 2018) – ינאי ליבנה

Unlink Exploitation - Heap Meta-Data Manipulation (גליון 81, מרץ 2017)

GLIBC Malloc source code

https://elixir.bootlin.com/glibc/glibc-2.32/source/malloc/malloc.c

https://www.gnu.org/software/libc/manual/html_node/Tunables.html

Vulerability assessment&research

https://en.wikipedia.org/wiki/Attack_surface

https://gynvael.coldwind.pl/?id=659 - How to find vulnerabilities

Largebin attack

https://www.anquanke.com/post/183877

Tcache Dumping / Stashing Unlink / TSU+ / TSU++

https://qianfei11.github.io/2020/05/05/Tcache-Stashing-Unlink-Attack/

Safe-Linking mitigation + bypassing it

CPR - Eyal Itkin - 2020 - safe linking eliminating a 20 year old malloc exploit-primitive

Bypassing glibc v2.32 safe linking mitigation - Robert Crandall

https://awaraucom.wordpress.com/2020/07/19/house-of-io-remastered/

House of IO

CSAW 2021 Qualifiers CTF – word_games challenge: Origin, Sol-2, Sol-3 ,Sol-4

House of Corrosion

https://github.com/CptGibbon/House-of-Corrosion

Explanation of House of Corrosion ptr-yudai 19/10/2019 – (Use www.deepl.com/translator)

https://www.notion.so/House-of-Corrosion-cb62019a81734f6bb2fbd294aae17361 – (Use deepl)

Modern day GLIBC heap mitigatoins

[PATCH] Harden tcache double-free check 2021 glibc 2.34 + Mailing list comments

[PATCH] Update tcache double-free check [2020.07[

House of Rust

https://c4ebt.github.io/2021/01/22/House-of-Rust.html

House of Orange

https://www.anquanke.com/post/id/168802 - House of Orange + File stream exploitation

Play with FILE Structure - An-Jie Yang (AngelBoy)

File structure/streams exploitation + File stream oriented programming

Pwning ELFs for Fun and Profit- ASIS CTF – Jim Moriarty - יובל עטיה – גליון 88 + פתרון נוסף

על הדבש ועל הקובץ - Pwning File structures – חלק א – עמית שמואל – גליון 130

על הדבש ועל הקובץ - Pwning File structures – חלק ב – עמית שמואל – גליון 132

https://teamrocketist.github.io/2020/02/05/Pwn-HackTM-2020-Trip-To-Trick/

מנגנוני הקצאה נוספים

A Tale of Two Mallocs: Shmarya Rubenstein, INFILTRATE 2018

A Tale of Two Mallocs: On Android libc Allocators

סיכום

.....

אם מצאתם טעויות במסמך, הסברים לא מספיק ברורים או שיש לכם רעיונות כיצד לשפר או לתקן אותו, אשמח לקבל פידבק מכם. את הגרסה העדכנית ביותר של המסמך, יחד עם חומרים נלווים, ניתן יהיה למצוא בחשבון ה-GitHub שלי תחת binary/heap exploitation.

על המחבר

עידן בנני – חוקר ומפתח Low-level. מתעניין במחקר חולשות התקפי ואקספלוטציה, הנדסה לאחור, מערכות הפעלה Android ו-Linux, IoT, SDR & Wireless RF Hacking, FRIDA, וארכיטקטורת ARM. יוצר סירטונים הדרכתיים ביוטיוב. בזמן הפנוי משתתף בתחרויות CTF כחלק מקבוצת CamelRiders. ניתן ליצור איתי קשר באמצעי המדיה השונים: Telegram, LinkedIn, Twitter או באי-מייל idan.banani@gmail.com