Technologic Papers

מיטיגציות ב-MachO ואיך לזהות אותן

מאת אדי צלוליכין

מה היא מיטיגציה בקבצים בינאריים?

בכל תוכנה הכתובה בשפת תכנות Level Low המאפשרת לעשות פעולות מסוכנות בזיכרון - קיימים באגים. חוקרי חולשות יודעים למצוא ולהשתמש בחולשות אלו בעת שהתוכנה רצה בזיכרון, להשתלט על זרימת הקוד הטבעית בכדי לגרום להרצת קוד משלהם וע"י כך אף להשתלט על השרת או המחשב עליו תוכנה זו רצה.

חולשת הזכרון המפורסמת ביותר, Stack Overflow, תועדה לראשונה ע"י AlpehOne במגזין Phrack[1], במאמר המפורסם: "Smashing the Stack for Fun and Profit". מאז פרסום המאמר ועליית המודעות לחולשות הזיכרון אשר יושבות חבויות בתוכנות שבהן אנו משתמשים כל יום, החלו מפתחי מערכות ההפעלה והמהדרים לפתח מיטיגציות - או דרכים להתמודד ולהקשות על הפיכת באג בניהול זיכרון לחולשה שיכולה לגרום להרצת קוד של תוקף. בזכות המיטיגציות שיש לנו כיום במערכות הפעלה מודרניות, חולשות אשר מאפשרות הרצת קוד מרחוק הינן יחסית נדירות (ולכן עולות הרבה כסף וזמן מחקרי על ידי אנשים אשר מומחים בתחום זה, שזו אומנות בדיוק כמו שזה סוג של הנדסת תוכנה). בהמשך אסקור את המיטיגציות שקיימות במכשירים של Apple, שכן החומרה והתוכנה של אפל הינה המובילה בעולם מבחינת מיטיגציות נגד ניצול חולשות [19,20].

כאשר חוקר חולשות רוצה לתקוף פיסת קוד כלשהי, עליו לזהות איזו מיטיגציות משפיעות עליו, בכדי שידע איזה דרכי מעקף הוא צריך לחפש. אחד הכלים המפורסמים ביותר שכל חוקר חולשות עושה בו שימוש הינו Checksec[5], כלי source open אשר יודע לפרסר קבצים בינארים של מערכות ההפעלה המבוססות Linux ולספק לחוקר את המידע שהוא צריך בכדי לנצל את החולשות המסתתרות בתוך הבינארי.

כחלק מהמחקר שלי גיליתי כי לא קיים כלי בדומה ל-Checksec אשר יודע לזהות את המיטיגציות של Apple, ולכן כתבתי אחד [21,22]. בהמשך הגיליון אפרט על רוב ההקשחות המשמעותיות שקיימות מבוססות הפעלה במערכות XNU כמו wearOS, iPadOS, macOS, iOS ו-visionOS.

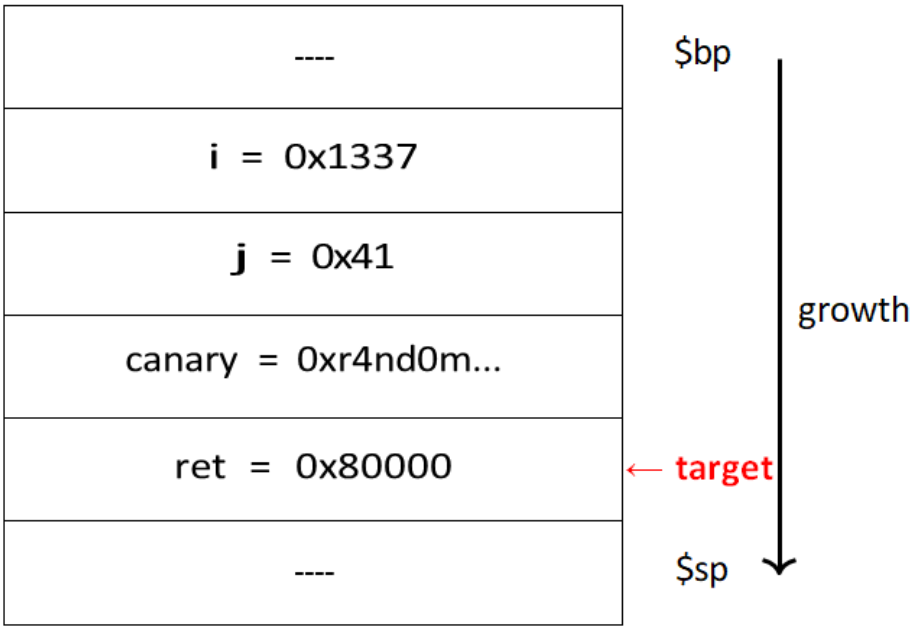
סוגי מיטיגציות

ישנן שלושה סוגים של מיטיגציות[8]: ברמת המהדר (כנריות), ברמת מערכת ההפעלה (ASLR), וברמת החומרה (PAC). מכאן ועד סוף המאמר נרחיב על כל מיטיגציה, איך היא עובדת, ואיך לזהות אותה.

כנריות

מה הן כנריות?

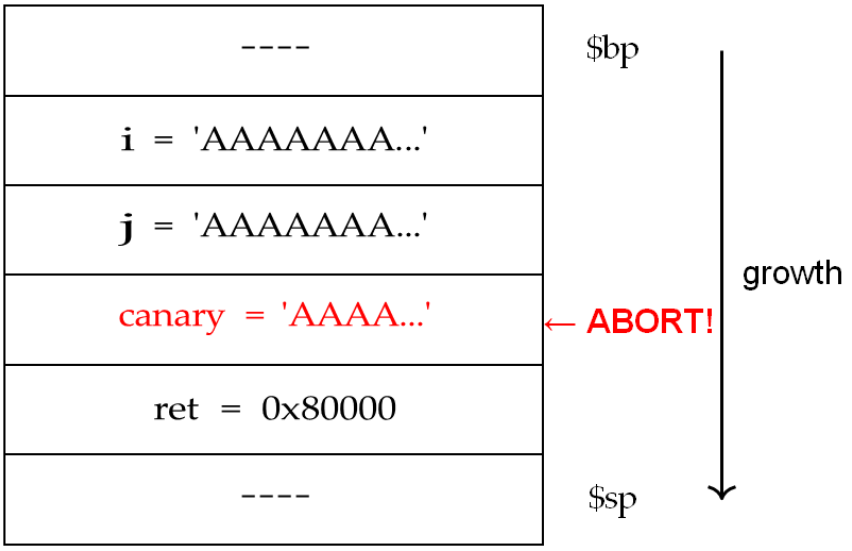
כנריות היא מיטיגציה ברמת המהדר אשר באה במטרה להקשות על תוקף להשמיש חולשת stack smashing בכדי להשתלט על ה-flow של התוכנה. כעת אזכיר לכם את שיעור האסמבלי שעברתם בתיכון בכדי להמחיש כיצד המיטיגציה עובדת. להלן מחסנית:



בתוך המחסנית ישנו סוג מידע אשר התוקף מאוד היה רוצה לשכתב: ה-return address. מתקפות stack smashing מאפשרות לתוקף לדרוס ערכים בתוך המחסנית ולשכתב אותם. הערך המדובר הינו כתובת הזכרון של הפונקציה אשר קראה לפונקציה שכרגע רצה, שכן זו כתובת החזרה לפונקציה שקראה לשלנו. בהינתן והתוקף מצליח לכתוב לחלק הזה של המחסנית, הוא יכול להגיד לתוכנה לבצע כל פעולה שירצה.

כנריות הינן הפתרון שלנו. אם תסתכלו היטב בטבלה, תוכלו לשים לב שיש ערך שנקרא canary לפני ה-ret. לפני שהתוכנה קופצת לכתובת הזכרון, התוכנה ראשית תבדוק שהערך של הכנרית לא השתנה. מה מונע מהתוקף לשכתב את הערך של הכנרית? הכנרית יושבת במקום מיוחד בזכרון אשר הינו read-only ומשתנה בין הרצה להרצה, הכנרית הינה פשוט ערך מספרי כלשהו אשר ארוך דיו בכדי לנחשו. מאחר וניחוש לא נכון יגרום לקריסה של התוכנה - ההגנה יעילה.

כך נראת מחסנית אשר תוקף, ניסה להשמיש בה stack overflow אך כשל בגלל הכנרית:



איך מזהים כנריות?

שיטה ראשונה:

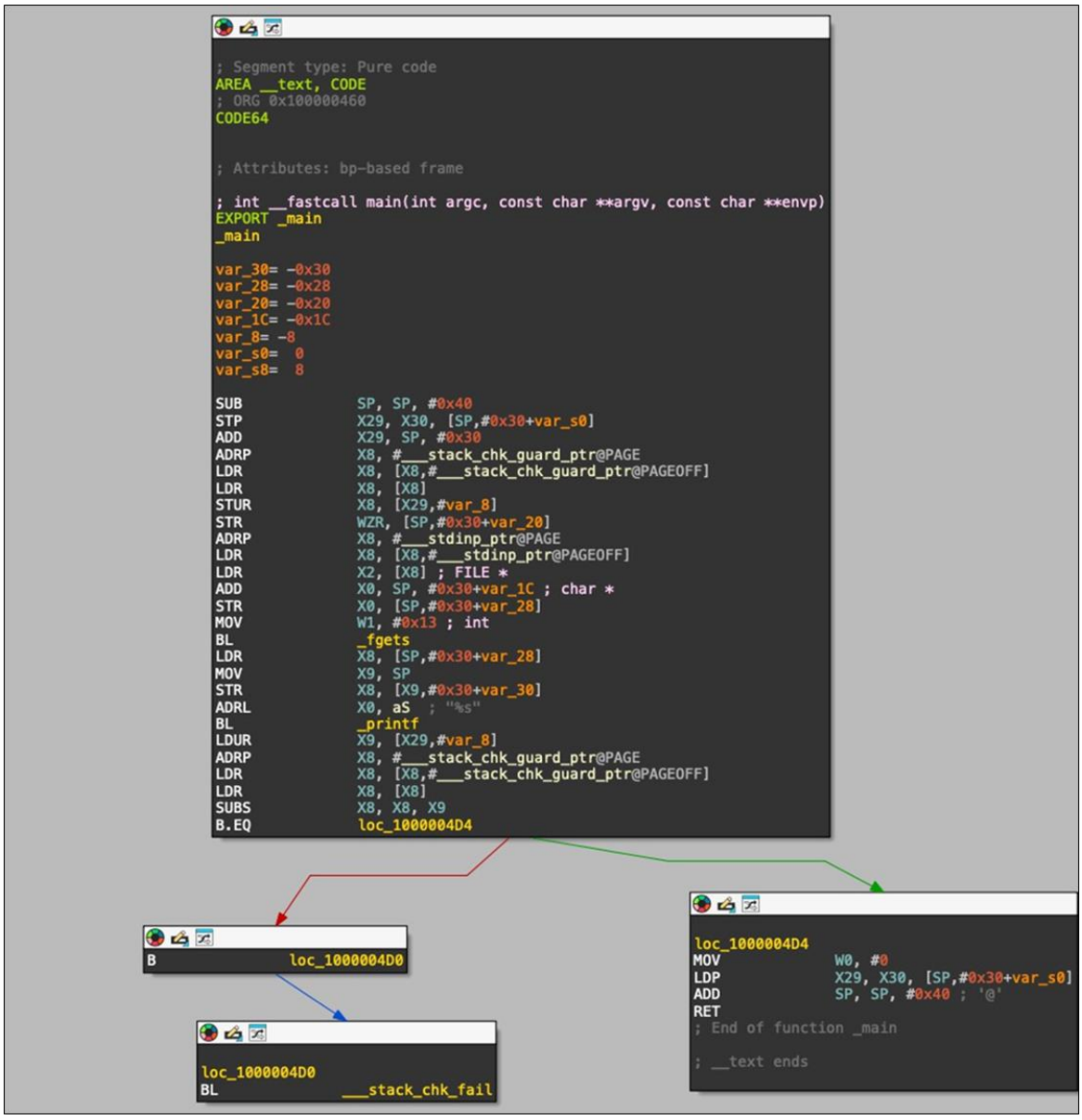
בואו נקח למשל את פיסת הקוד הבאה:

```
# include <stdio.h>

int main()
{
    char buffer[20];
    fgets(buffer,sizeof(buffer)-1,stdin); printf("\% s", buffer);
    return 0;
}
```

כפי שאתם רואים, תוכנה יחסית פשוטה, ללא conditionals רק מדפיסה פלט. ללא שום איזכור לכנריות. אך הן עדיין שם ברמת האסמבלי. כאשר המהדר מיצר את הקובץ הבינארי שלנו, הוא בעצם מזריק פונקציות ועוד קטעי קוד לבינארי שלנו כחלק מהמיטיגציה הזאת. ניתן לראות זאת בתוכנה ב-IDAPro.

להלן דיקומפילציה של התוכנה:



כפי שניתן לראות, פתאום נוספו לזרימת התוכנה הזאת בדיקות. ונוספה פונקציה חדשה:

_stack_chk_fail

זו בעצם הפונקציה אשר בודקת אם הכנרית תקינה לפני שהתוכנה קופצת להריץ קוד מהערך שבמחסנית.

שיטות זיהוי:

שיטה1 : חיפוש סמלים

חיפוש הפונקציה __stack_chk_fail בטבלת הסמלים. אם שמות הפונקציות לא קיימות (הבינארי stripped), ניתן להשתמש בשיטה 2.

שיטה 2: זיהוי דפוסי אסמבלי

חיפוש פקודות אסמבלי ייחודיות שמזיזות את הכנרית:

- x86_64: פקודת mov עם gs:[0x28] או gs:[0x14]
- ARM64: פקודת ldr/ldur עם הרגיסטר 0x18 או tpidr_el
- ARM32: פקודת ldr עם stack_chk_guard

דוגמה לפקודה ייחודית:

```
mov rax , QWORD PTR gs:0 x28
```

במעבדי ARM64 קיימת קונבנציה שהאוגר x18 משמש כמצביע ל-TLS (ה-Thread Local Storage) שבו נשמרת הכנרית (בין ערכים אחרים ספציפיים לתהליכון). קונבנציה זו אפשרית משום שיש ב-ARM64 שלושים רגיסטרים לשימוש כללי (לעומת 16 ב-x86-64), ולכן ניתן להקדיש רגיסטר אחד עבור TLS. הרגיסטר המיוחד נקרא tpidr_el0.

הערה: הכלי machsec [21] משלב את כל שלושת השיטות הללו לזיהוי מדויק של כנריות בבינאריים של macOS/iOS.

PIE

PIE או בשמה המלא Position-Independent Executables הינה מיטיגציה אשר מרנדמת את כתובות הזכרון של הפונקציות בתוך הבינארי. המיטיגציה באה במטרה לעשות חיים קשים יותר לתוקף משום שכעת הוא צריך להדליף את כתובת הזכרון של הפונקציה אליה הוא מעוניין לקפוץ. ככה משפיעה המיטיגציה על כתובות הזכרון של הבינארי:

Without PIE

00	00	00	00	00	00	40	00	00 80

base (fixed) offset

With PIE

00	00	7f	8a	4b	20	00	00	00 80

base (randomized) offset (same)

האופסט משתנה בין הרצה להרצה, ולכן מאוד קשה לנחש את הכתובת הנכונה. כעת, בשביל להשמיש חולשה תוקף צריך עוד שלב, פרימיטיב של דליפת זיכרון.

זיהוי PIE

זיהוי PIE מתבצע באמצעות ניתוח הכותרות (headers) של קובץ הבינארי Mach-O[3,4] הכותרות של הבינארי הן בעצם מטא-דאטה על הבינארי. המטא-דאטה כוללת דברים כמו:

- סוג הבינארי (ELF/Mach-O)
- ארכיטקטורת הבינארי (ARM/x86)
- אילו הגנות מופעלות (NX או PIE)

רק על ידי ניתוח הכותרות, אנו יכולים להשיג מידע רב על הבינארי. מבנה הכותרת של Mach-O נראה כך:

מידע מרכזי:

הבתים הראשונים מזהים את סוג הקובץ:

- 0xfeedface (Mach-O 32-bit)
- 0xfeedfacf (Mach-O 64-bit)
- 0xcafebabe (fat binary) - בינארי המכיל מספר ארכיטקטורות (למשל x86_64 ו-ARM64) בקובץ אחד. זה נפוץ במיוחד אצל Apple בגלל המעבר מ-Intel ל-Apple Silicon, כדי שאותה תוכנה תוכל לרוץ על שני סוגי המעבדים.

מבנה הכותרת (32-bit = 28 bytes, 64-bit = 32 bytes):

```
struct mach_header_64 {
    uint32_t magic; // 0x00: Magic number
    uint32_t cputype; // 0x04: CPU architecture
    uint32_t cpusubtype; // 0x08: CPU variant
    uint32_t filetype; // 0x0C: Executable , library , etc.
    uint32_t ncmds; // 0x10: Number of load commands
    uint32_t sizeofcmds; // 0x14: Size of load commands
    uint32_t flags; // 0x18: FLAGS INCLUDING MH_PIE!
    uint32_t reserved; // 0x1C: (64-bit only)
};
```

זיהוי PIE: הדגל MH_PIE ממוקם בשדה flags בתזוזה 0x18 (offset). ערכו הוא 0x00200000. כאשר ביט זה מופעל, הוא מציין שהבינארי תומך Position-Independent Execution.

תהליך הזיהוי:

הזיהוי מתבצע על ידי בדיקת דגל MH_PIE בשדה ה-flags של כותרת Mach-O.

No eXecute (NX)

מהו NX?

NX (או No Execute) היא הגנה שמסמנת אזורי זיכרון מסוימים כ-"לא ניתנים להרצה". כלומר, אפילו אם תוקף מצליח להזריק קוד מזיק ל-stack או ל-heap, המעבד יסרב להריץ אותו. זה חוסם סוג שלם של התקפות שבהן תוקף מכניס shellcode לזיכרון ואז קופץ אליו.

ההגנה פועלת ברמת החומרה - המעבד בודק את הרשאות ההרצה של כל דף זיכרון (memory page) לפני שהוא מריץ קוד ממנו.

איך מזהים NX?

במערכות הפעלה מבוססות XNU מודרניות, הליבה תמפה את המחסנית של התהליכון כלא ניתן להרצה, כברירת מחדל. לכן, הדרך היחידה לדעת אם NX לא מופעל היא לבדוק האם בזמן הרצה יש משהו אשר הופך את המחסנית ל-executable.

RPATH

מהו RPATH?

RPATH - הם נתיבים שמוגדרים בבינארי ומציינים היכן מאוחסנות ספריות שהתוכנית צריכה לטעון בזמן ריצה (runtime). הבעיה היא שאם הנתיבים האלה מצביעים למקומות שבהם תוקף יכול לכתוב קבצים - למשל אותה תיקייה של הבינארי, או תיקיות משותפות בין משתמשים - התוקף יכול להחליף את הספריות האמיתיות בספריות זדוניות ולהשתלט על הריצה של התוכנית. זו פרצת אבטחה חמורה כי היא מאפשרת Code Execution על ידי החלפת ספריות לגיטימיות בגרסאות מזויפות.

איך מזהים בעיות ב-RPATH?

הזיהוי מתבצע על ידי סריקה של פקודות הטעינה בבינארי לחיפוש:

- פקודות LC_RPATH - נתיבי חיפוש מוגדרים
- מחרוזת @rpath בתוך פקודות LC_LOAD_DYLIB

דוגמה למבנה Mach-O:

```
Load Commands:
LC_RPATH: /usr/local/lib              ←       RPATH
LC_LOAD_DYLIB: @rpath/lib.dylib       ←      @rpath
LC_LOAD_DYLIB: /usr/lib/libc.dylib    ←      @rpath
```

למה זה מסוכן:

נניח שהבינארי מכיל:

- libs LC_RPATH/: (נתיב יחסי באותה תיקייה)
- LC_LOAD_DYLIB: @rpath/important.dylib

תוקף יכול ליצור libs/important.dylib זדוני באותה תיקייה, והבינארי יטען אותו אוטומטית במקום הספרייה האמיתית.

FORTIFY

מהו FORTIFY?

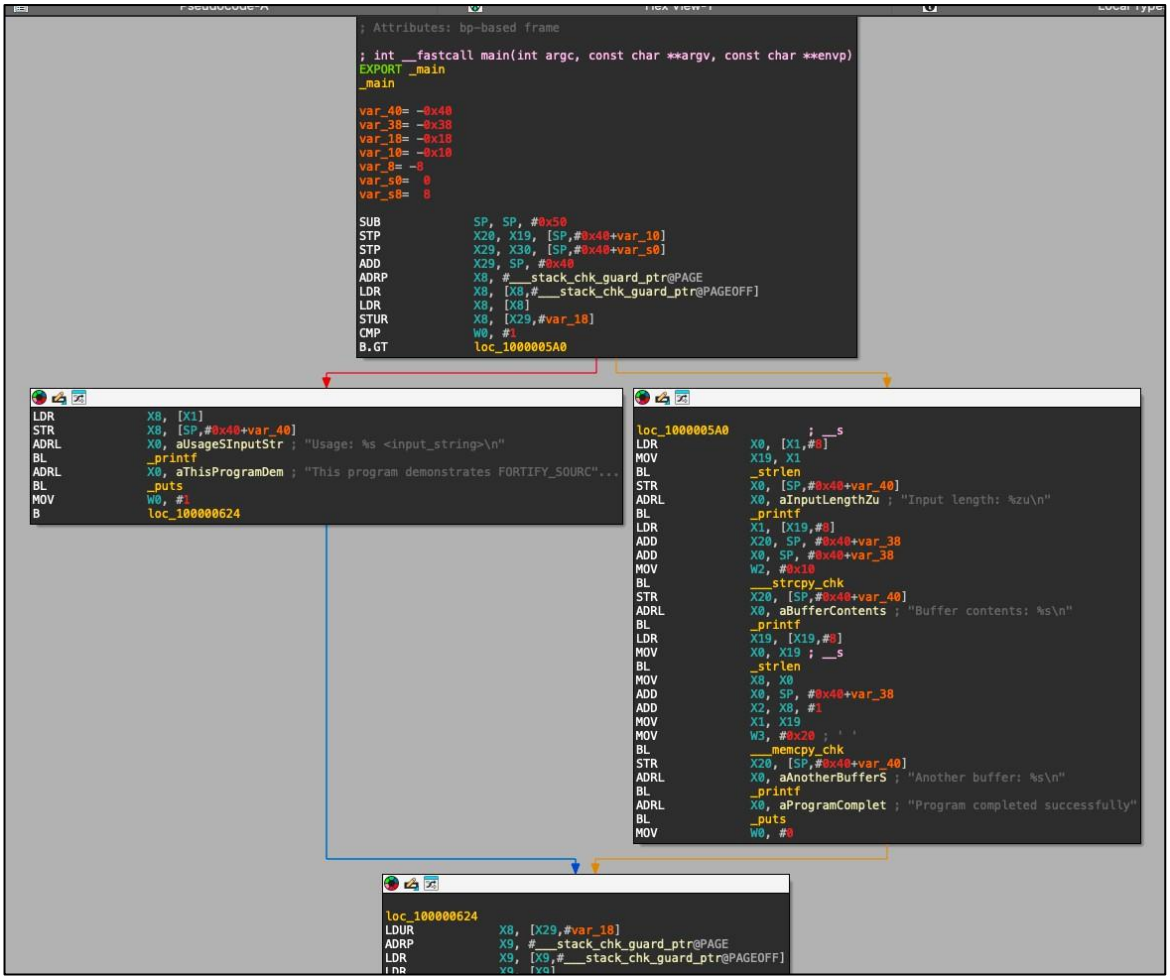
FORTIFY הוא דגל קומפילציה שמחליף פונקציות מסוכנות לזיכרון בגרסאות בטוחות יותר שיש להן בדיקות גבולות (bounds checking). המיטיגציה פועלת ברמת הקומפיילר ונוצרה על ידי GNU compiler, ומאוחר יותר אומצה על ידי Clang.

הפונקציות העיקריות שהיא מטפלת בהן הן פונקציות כמו strcpy(), memmove(), memcpy(), ועוד. זו מיטיגציה חלשה יחסית, אבל עדיין חשוב להכיר אותה.

איך מזהים FORTIFY?

המיטיגציה משנה את האופן שבו הפונקציות עובדות, מה שאומר שהקוד ברמת ה-assembly נראה שונה מבינארי שקומפל בלי המיטיגציה הזו.

כפי שניתן לראות בתמונה הבאה:



פונקציות לא בטוחות לזיכרון כמו memcpy נקראות כעת memcpy_chk(). למעשה הפונקציה memcpy הוחלפה ברמת ה-linking בגרסה מאובטחת יותר. אם פשוט נחפש את הסמלים (symbols) האלה, נוכל לזהות את המיטיגציה.

תהליך הזיהוי:

הזיהוי מתבצע ע"י סריקת ה-symbol table של הבינארי לחיפוש שמות פונקציות מוגנות כגון memcpy_chk, strcpy_chk, sprintf_chk. נוכחות של פונקציות תעיד על הפעלת FORTIFY_SOURCE.

מנגנוני זיכרון מותאמים אישית של Apple

xzone malloc-ו kalloc_type

בנוסף למיטיגציות הסטנדרטיות, Apple פיתחה מנגנוני הקצאת זיכרון מותאמים אישית המספקים הגנה מתקדמת נגד פגיעויות זיכרון. המנגנונים העיקריים הם kalloc_type (ברמת הקרנל, הושק ב-iOS 15) ו-xzone malloc (ברמת המשתמש, הושק ב-iOS 17).

הבעיה עם מנגנוני הקצאה מסורתיים:

מנגנוני הקצאה מסורתיים מארגנים את הזיכרון לפי **גודל** בלבד - כל ההקצאות בגודל 64 בתים נמצאות באותו אזור (ללא קשר למה הן מכילות port, pipe, socket וכו'), הנ"ל מאפשר לתוקפים לבצע grooming heap - טכניקה שבה התוקף ממלא את הזיכרון באובייקטים שהוא שולט בהם כדי לשלוט במה שיוקצה במקום של אובייקט שהוא משחרר.

הפתרון - הקצאה לפי סוג:

kalloc_type מארגן את הזיכרון לפי **סוג האובייקט** ולא לפי גודל. כל סוג מקבל אזור זיכרון ייעודי משלו, גם אם הגודל זהה לסוגים אחרים. לדוגמה:

- כל ה-sockets באזור אחד
- כל ה-pipes באזור נפרד
- כל ה-ports באזור נפרד

התוצאה: grooming heap הופך לקשה משמעותית - אובייקט מסוג אחד לא יכול לתפוס את מקומו של אובייקט מסוג אחר.

מנגנון ההגנה:

- **בידוד וירטואלי קבוע:** כל אזור מקבל טווח כתובות וירטואליות שנשאר שמור לו **לצמיתות**, גם אם הזיכרון הפיזי התרוקן. הכתובות לעולם לא יעברו לשימוש של סוג אחר
- **הגנה מפני UAF:** בגלל הבידוד, פגיעות Use-After-Free לא ניתנות לניצול - אובייקט משוחרר יוחלף רק באובייקט **מאותו הסוג**
- **רנדומיזציה:** בתוך כל אזור, ההקצאות מפוזרות באופן אקראי

בדיקה במציאות - המקרה של SockPuppet:

SockPuppet[17] הינו אקספלויט שפותח על ידי Ned Williamson מ-Google Project Zero. החולשה מנצלת פרימיטיב ראשוני של Use-After-Free ומשרשרת את הפגיעות יחד עם Type Confusion בכדי לבצע Privilege Escalation החולשה התגלתה בקרנל של XNU ב-2019 ב-Subsystem שאחראי על Sockets.

Apple בדקה רטרואקטיבית מה היה קורה אם kalloc_type היה קיים בזמן הפגיעות. התוצאה: **הניצול היה נכשל לחלוטין** - התוקף לא יכול לגרום לאובייקט שונה לתפוס את מקום ה-socket המשוחרר, כי הם באזורי זיכרון נפרדים.

סיכום היתרונות:

- מקשה משמעותית על heap grooming - תוקף כמעט לא יכול לשלוט מה יחליף אובייקט משוחרר
- מפחית משמעותית את יעילות התקפות Use-After-Free ו-Type Confusion
- עובד אוטומטית ללא שינויים בקוד
- הופך פגיעויות קיימות רבות להרבה יותר קשות לניצול

למידע נוסף ראו:

https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/

UBSAN (Undefined Behavior Sanitizer)

מהו UBSAN?

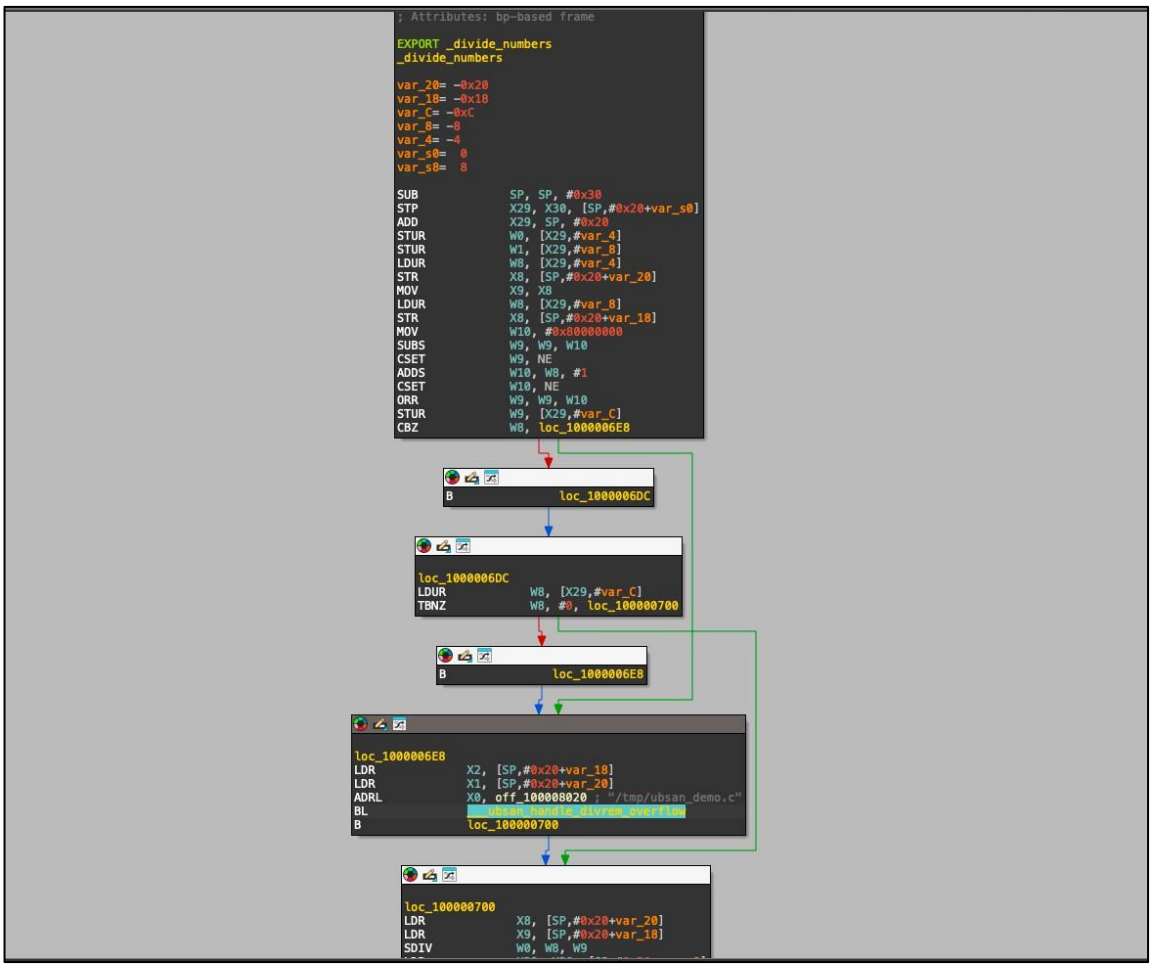
UBSAN היא מיטיגציה ברמת הקומפיילר שמוסיפה בדיקות לתוכנית, בדומה ל-Stack Canaries ו-FORTIFY. המטרה היא לתפוס התנהגויות לא מוגדרות (undefined behavior) בזמן ריצה, כמו:

- Integer overflow/underflow - גלישה של מספרים שלמים
- באגים של סימן (signedness bugs) - בעיות עם מספרים חתומים/לא חתומים
- גישה למערכים מחוץ לגבולות
- שימוש במשתנה לא מאותחל
- המרות סוגים מסוכנות

כל פעם שהקוד עושה משהו שנחשב להתנהגות לא מוגדרת, UBSAN מזהה את זה ויכול לעצור את התוכנית או להדפיס אזהרה.

איך מזהים UBSAN?

בדומה ל-FORTIFY, UBSAN מוסיף פונקציות בדיקה לקוד. אם נחפש את הסמלים האלה בטבלת הסמלים, נוכל לדעת אם UBSAN מופעל:



כפי שניתן לראות, פונקציות בדיקה חדשות מופיעות בקוד, כמו סמלים שמתחילים ב-ubsan או _ubsan_handle.

תהליך הזיהוי:

הזיהוי מתבצע על ידי סריקת טבלת הסמלים לחיפוש פונקציות המכילות ubsan, sanitizer, או _ubsan_handle. נוכחות של סמלים אלו מעידה על הפעלת UBSAN.

ASAN (Address Sanitizer)

מהו ASAN?

ASAN (Sanitizer Address) היא מיטיגציה ברמת הקומפיילר שמשמשת בעיקר יחד עם fuzzers (כלי בדיקה אוטומטיים) כדי לזהות פגיעויות זיכרון בזמן ריצה. המיטיגציה מתמחה בזיהוי:

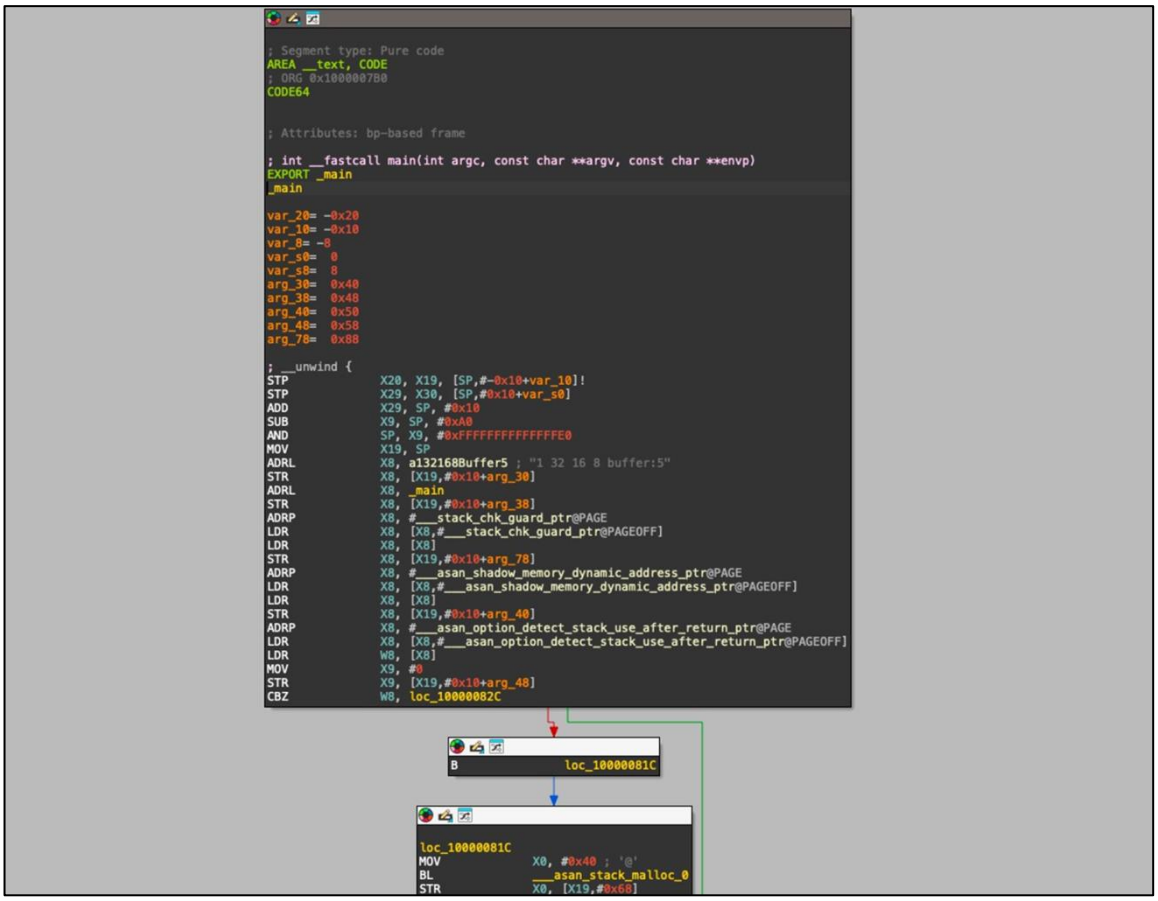
- Out-of-Bounds קריאה/כתיבה - גישה מחוץ לגבולות המערך
- Use-After-Free - שימוש בזיכרון משוחרר

- Double-Free - שחרור כפול של אותו זיכרון
- Memory leaks - דליפות זיכרון
- Heap/Stack Overflow - גלישת מחסנית/זיכרון דינמי

ASAN עובדת בצורה דומה למיטיגציות הקודמות - היא מוסיפה בדיקות ופונקציות לתוכנית. עם זאת, המחיר הוא גבוה: הביצועים של התוכנית יורדים משמעותית, והגודל של הבינארי גדל. בגלל זה, ASAN משמשת בדרך כלל רק בזמן פיתוח ובדיקות, ולא בגרסאות סופיות.

איך מזהים ASAN?

הזיהוי עובד בדיוק כמו UBSAN ו-FORTIFY - חיפוש סמלים ספציפיים בטבלת הסמלים.



כפי שניתן לראות, מספר רב של פונקציות ASAN מתוספות לקוד, כמו asan_load, asan_store, __interceptor_malloc, וכו'

תהליך הזיהוי:

הזיהוי זהה ל-UBSAN - סריקת טבלת הסמלים לחיפוש שמות פונקציות המכילים __asan_, __interceptor_malloc, או sanitizer_cov.

מתי להשתמש ב-ASAN:

בזמן פיתוח, עם fuzzing, ובבדיקות - אך **לא בייצור** בגלל פגיעה קשה בביצועים.

CFI (Control Flow Integrity)

מהו CFI?

CFI (Integrity Flow Control) היא מיטיגציה שמנסה לוודא שהתוכנית עוקבת אחר זרימת הביצוע הנכונה על ידי הוספת בדיקות לקוד, בדומה למיטיגציות כמו ASAN, UBSAN ו-stack canaries. המטרה היא למנוע התקפות שמנסות לשנות את זרימת הביצוע של התוכנית, כמו:

- ROP (Return-Oriented Programming) - קיים בקוד חוזר שימוש
- JOP (Jump-Oriented Programming) - קיים לקוד קפיצות
- שינוי של Function Pointers - החלפת מצביעי פונקציות

חשוב לציין: זו המיטיגציה **היקרה ביותר** מכל המיטיגציות התוכנתיות - היא מגדילה משמעותית את גודל הבינארי ויש לה פגיעה ביצועים עצומה.

רלוונטיות ל-macOS/iOS:

CFI לא עובד במערכות XNU מודרניות! גם אם תנסו לקמפל בינארי עם CFI במערכות macOS/iOS מודרניות, המיטיגציה **תוחלף אוטומטית** ב-PAC (Codes Authentication Pointer).

PAC הוא יישום חומרתי של Control Flow Integrity שמובנה ישירות במעבדי Apple Silicon ו-ARM64. הוא הרבה יותר יעיל, מהיר, ומתקדם מ-CFI התוכנתי (נדון ב-PAC בפירוט בהמשך המאמר). לכן CFI רלוונטי בעיקר ל-Linux/ELF binaries, ולא למערכות Apple מודרניות שמשתמשות ב-PAC.

איך מזהים CFI?

למרות שהמיטיגציה מוחלפת ב-PAC במערכות macOS/iOS מודרניות, הזיהוי עובד בדיוק כמו המיטיגציות הקודמות - חיפוש סמלים ספציפיים בטבלת הסמלים.

תהליך הזיהוי:

הזיהוי זהה למיטיגציות הקודמות - חיפוש סמלים המכילים _cfi, cfi_ או cfi_check.

סיכום: CFI היא המיטיגציה התוכנתית היקרה ביותר, אך **לא עובדת** ב-macOS/iOS מודרניים, היא מוחלפת אוטומטית ב-PAC, יישום חומרתי מתקדם יותר.

סמלים הסרת (Symbol Stripping)

מהי הסרת סמלים?

הסרת סמלים אינה מיטיגציה במובן הקלאסי, אלא הסרת מידע שמקשה על Reverse Engineers כאשר אתם נותנים שם לפונקציה בקוד, הקומפיילר לא באמת צריך את השם כדי להשתמש בפונקציה - רק כתובת זיכרון שבה הפונקציה נמצאת. אנחנו בני האדם, לעומת זאת, זקוקים מאוד לשמות הפונקציות - הם עוזרים לנו להבין מה הפונקציה עושה במבט ראשון, מבלי לקרוא את כל הקוד.

רוב התוכניות יוציאו את הסמלים שלהן (stripped) כדי ש-Reverse Engineer לא יקבל את שמות הפונקציות, וישאר עם שמות תמוהים כמו func_1234() (השמות תלויים בכלי הדה-קומפילציה שבו משתמשים).

איך מזהים הסרת סמלים?

בשונה מהמיטיגציות האחרות שבהן חיפשנו משהו ש**קיים** בבינארי, כאן אנחנו מחפשים משהו ש**לא קיים**. אנחנו בודקים כמה סמלים יש בבינארי על ידי קריאת הכותרות והמטא-דאטה.

תהליך הזיהוי:

הזיהוי מתבצע על ידי ספירת סוגי הסמלים: מקומיים (local), חיצוניים (external), ולא מוגדרים (undefined). לפי זה נקבע: **Not stripped (כל הסמלים), Partially stripped (רק חיצוניים) או Fully Stripped (אין סמלים)**

למה זה חשוב:

בינארי ללא סמלים הרבה יותר קשה לבצע עליו Reverse Engineering - במקום שמות פונקציות כמו authenticate_user() תראו sub_401234(), זה מאט משמעותית את העבודה של Reverse Engineers ושל ותוקפים.

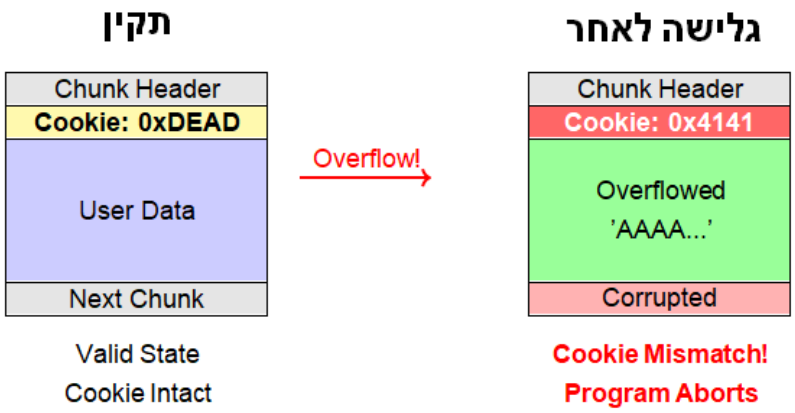
Heap Cookies

מהם Heap Cookies?

Heap Cookies הם בדיוק אותו רעיון כמו Stack Canaries, רק שהם פועלים על ה-heap במקום על ה-Stack. הם בודקים שלא התרחה גלישת Dynamically Allocated Memory Heap Chunks Blocks. ומגנים מפני השחתה של מבני נתונים שמוקצים באופן דינמי.

כשמקצים זיכרון על ה-heap עם calloc ,malloc, וכו', המערכת מוסיפה ערך בדיקה (cookie) ליד הבלוק המוקצה. לפני ששוחררים את הזיכרון או משתמשים בו, המערכת בודקת שה-cookie לא השתנה. אם הוא השתנה - זה אומר שהתרחשה גלישת זיכרון, והתוכנית נעצרת.

וזואליזציה של Heap Cookies:



הסבר: בצד שמאל - מצב תקין שבו ה-cookie שמור ותקף. בצד ימין - לאחר גלישת buffer, ה-cookie נדרס ונשתנה, מה שמאפשר למערכת לזהות את התקיפה ולעצור את התוכנית.

איך מזהים Heap Cookies?

זיהוי Heap Cookies מתבצע על ידי בדיקת סמלים בבינארי שקשורים להקשחת heap, כמו guard_malloc, _malloc_check ו-malloc_zone.

תהליך הזיהוי:

סריקת טבלת הסמלים לחיפוש פונקציות הקשורות להקשחת heap כגון guard_malloc, malloc_zone, _malloc_check.

התוצאה: אם נמצא אפילו סמל אחד של הקשחת heap, המיטיגציה מופעלת.

הבדלים מ-Stack Canaries:

- Stack Canaries מגנים על ה-Stack, בעוד ש-Heap Cookies מגנים על ה-Heap.
- Stack Canaries בודקים בסוף הפונקציה - Heap Cookies בודקים בעת שחרור זיכרון
- שניהם משתמשים באותה הגיון: ערך בדיקה שאמור להישאר קבוע

הגנה מפני גלישת מספרים שלמים (Integer Overflow):

מהי הגנה מפני Integer Overflow?

הגנה מפני גלישת מספרים שלמים היא עוד הגנה מבוססת Instrumentation, שבה אנו מוסיפים פונקציות בדיקה לתוכנית כדי לוודא שמספרים שלמים לא חרגו מהגבולות שלהם (Wrapped Around).

במחשב, מספרים שלמים מיוצגים במספר קבוע של ביטים. לדוגמה:

- uint8_t - הינו גודל של 8bits (טווח: 0-255)
- int32_t - הינ גודל של 32bits (טווח מ-2,147,483,648- עד 2,147,483,647)

כאשר חישוב גורם למספר לחרוג מהטווח שלו, הוא "עוטף" חזרה. לדוגמה:

תוקפים יכולים לנצל זאת כדי:

- לעקוף בדיקות גודל buffer
- לגרום להקצאות זיכרון לא נכונות
- ליצור תנאים לא צפויים בלוגיקה

איך מזהים הגנה מפני גלישת מספרים שלמים?

סריקת טבלת הסמלים לחיפוש פונקציות כגון _ubsan_handle_add_overflow, _addoti4 או muloti4_.wrap_.

הפונקציות שאחריהן אנו מחפשים:

- _muloti4 - בדיקת גלישה בכפל
- _addoti4 - בדיקת גלישה בחיבור
- ubsan_handle_add_overflow מטפל ב-overflow של UBSan
- __wrap - פונקציות עטיפה (wrapper) לבדיקת חישובים

Sandbox (ייחודי ל-XNU)

מהו Sandbox?

באופן כללי, Sandbox הוא יצירת סביבה מוגבלת שבה לבינארי יש גישה מוגבלת למערכת ההפעלה הבסיסית. במקרה של XNU ומיטיגציות, הגנה זו עושה מספר דברים:

- **סינון קריאות מערכת** (דומה ל-seccomp בלינוקס) באמצעות תכונת ליבה בשם seatbelt
- **בדיקת הרשאות** (entitlements) - מה מותר לבינארי לגשת אליו, לאיזה סוג חומרה, וכו'
- **החלת פרופיל Sandbox** - על בסיס ההרשאות וקריאות המערכת שמסוננות

איך מזהים Sandbox?

אנו יכולים לזהות Sandbox באמצעות הסמלים הקיימים, על ידי בדיקה אם הבינארי חתום (שזה גם אינדיקטור חזק שהבינארי ב-Sandbox), וגם על ידי בדיקה של מחרוזות הקשורות להרשאות.

הזיהוי מתבצע בשלושה שלבים:

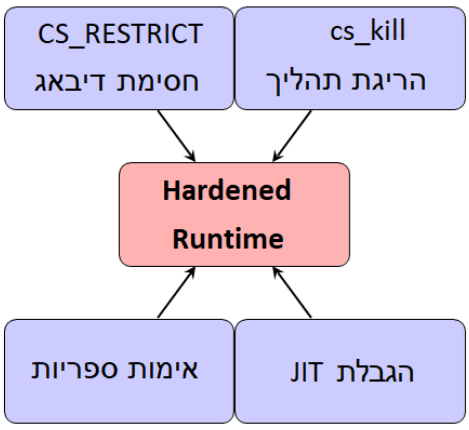
- חיפוש סמלי Sandbox - סריקה עבור _sandbox, _Container וכו'
- בדיקת Entitlements - חיפוש com.apple.security.app-sandbox בחתימת הקוד
- בדיקת חתימת קוד - נוכחות LC_CODE_SIGNATURE

Hardened Runtime + SIP + AMFI (ייחודי ל-XNU)

זוהי חבילת הגנות ברמת מערכת ההפעלה והבינארי, ייחודית ל-macOS, וחלקן ייחודיות רק ל-iOS.

מהו Hardened Runtime?

Hardened Runtime הוא סוויטת הגנות שנועדה להקשיח תהליכים בזמן ריצה נגד טכניקות תוקפניות נפוצות[13]. זוהי הגנה ברמת התהליך שמופעלת דרך חתימת קוד.



רכיבי ההגנה:

- **CS_RESTRICT** - כותרת Mach-O בבינארי, חלק ממדור חתימת הקוד של המטא-דאטה של הבינארי. חוסם task_for_pid(), ובכך הופך את התוכנית ללא ניתנת לחיבור על ידי דיבאגר, אלא אם יש להם את ההרשאות הנכונות
- **cs_kill** - הליבה הורגת את התהליך אם בכל נקודה בזמן ריצה החתימות אינן תואמות
- **אימות ספריות** (Library Validation) - בודק שכל הבינאריים חתומים ולא שונו
- **הגבלת JIT** (Just-In-Time) - קומפילציית JIT דורשת אזור זיכרון שיהיה ניתן לכתיבה והרצה, שזה סיכון אבטחה. ההגנה הזו לא תאפשר mmap() של אזור זיכרון שהוא גם write וגם execute, אלא אם מוגדרת הרשאה ספציפית: com.apple.security.cs.allow-jit
- **חסימת דיבאג** - חוסם ptrace() אלא אם יש הרשאה ספציפית: task-allow-com.apple.security.get

מהם Entitlements?

Entitlements הינם עוד הקשחה אשר קיימת ב-XNU. הרעיון הוא להגביל את ההרשאות שיש לתהליכון כלשהו, כך שהיה לו גישה לדברים שהוא צריך באמת.

(למשל, למחשבון לא צריכה להיות גישה למצלמת הרשת). כך שגם אם ובהינתן יש לכם הרצת קוד על המחשבון של אפל עדיין נדרש איכשהו לצאת מהתהליכון (sandbox escape) ולהריץ קוד משלכם (unsigned code execution). ה-entitlements מובנים לתוך הקובץ, ולא ניתן לגעת בהם בגלל שכל נגיעה בהם תשנה את החתימה של הקובץ, וקובץ לא חתום לא ירוץ על מערכת ההפעלה. נדבר על חתימות יותר לעומק בעמודים הבאים. המיטגציה הינה פשוט קובץ בפורמט שדומה ל-xml שנקרא plist. הקובץ הינו embedded לתוך התהליך.

מהו SIP (System Integrity Protection)?

SIP הוא סוויטה של הגנות אלה, שמבטיחה שמערכת ההפעלה ממשיכה לפעול תוך עמידות בפני תוכנות זדוניות, ניצול חולשות, ומשתמש טיפש שמוחק בטעות את כל מערכת הקבצים שלו כי מישהו אמר לו להריץ:

```
rf -rm /
```

ניתן להשבית את SIP ב-macOS דרך הפקודה:

```
csrutil disable
```

וזה חובה אם רוצים לעשות כל סוג של אבטחה התקפית כמו אינסטרומנטציה דינמית של תוכנית עם FRIDA, הרצת תוכנית פרוצה עם ספריות אימות רישיון מותאמות, וכו'.

הצפנת בינאריים

AMFI (Apple Mobile File Integrity) היא הגנה נוספת הקיימת ב-XNU, במיוחד ב-iOS[9,12] ההגנה מורכבת משני רכיבים שפועלים יחד:

- **amfid** (System Daemon)
- **מודול ליבה** (AppleMobileFileIntegrity.kext)

יחד, שני אלה מבצעים את הפונקציות הבאות:

פענוח בינאריים מוצפנים:

בינאריים בעלי סגמנט TEXT_ מוצפן, צריכים לפענח את הסגמנט כדי לרוץ כראוי. כותרת ה-Mach-O עם LC_ENCRYPTION_INFO מכילה פקודת טעינה עם:

```
cryptid=1   //   encrypted
cryptoff    // offset to encrypted data
cryptsize   // size of encrypted region
```

כאשר הליבה מבצעת את אימות החתימה, amfid מאמת שרשראות אישורים, פרופילי Provisioning, וכו.'

זיהוי AMFI ו-Hardened Runtime

זיהוי אם SIP מופעל ברמת מערכת ההפעלה הוא מחוץ לתחום של machsec, אבל בטוח להניח שכל מערכת הפעלה מבוססת XNU תהיה עם זה מופעל כברירת מחדל.

עם זאת, לא לכל הבינאריים יהיה Hardened Runtime מופעל, אז בהחלט נוכל לבדוק זאת בבינארי. הזיהוי מתבצע על ידי חיפוש הפקודות הבאות:

- LC_CODE_SIGNATURE - חתימת קוד (macOS)
- LC_ENCRYPTION_INFO / LC_ENCRYPTION_INFO_64 - הצפנה (iOS)

חתימת קוד (Code Signing)

מהי חתימת קוד?

בדומה ל-Windows, בינאריים המופצים על ידי ארגונים/חברות ידועים יכולים לקבל חתימה דיגיטלית מ-Apple, שעוזרת למכשיר Apple לדעת אם פיסת תוכנה היא עותק אמיתי מספק התוכנה, אם פיסת התוכנה עברה שינוי, וכו'

איך חתימת קוד עובדת?

חתימת קוד מצרפת blob חתימה בסוף הבינארי, ויש CodeDirectory (מבנה המכיל מטא-דאטה כמו דגלים ומערך של ה-hash-ים, SHA256 אחד לכל עמוד של KB4 של קוד הניתן להרצה) עם ה-hash-ים מחושבים מראש של כל עמוד שייטען לתוך סגמנט ה-TEXT_.

בזמן ריצה, הליבה מאמתת עמודים באופן עצלן דרך page faults - כאשר ניגשים לעמוד ניתן להרצה בפעם הראשונה, הליבה עושה hash שלו ומשווה מול ה-CodeDirectory. אם ה-hash-ים לא תואמים, התהליך נהרג.

שימו לב:

רק סגמנט ה-TEXT_ (קוד ניתן להרצה) חתום. הדברים הבאים **אינם** חתומים:

- Stack
- Heap
- _DATA (נתונים הניתנים לכתיבה)
- _DATA_CONST (עשוי להיות חתום במקרים מסוימים)
- __LINKEDIT
- זיכרון שהוקצה באופן דינמי

זיהוי חתימת קוד

זה פשוט כמו לנתח את כותרות הבינארי: הזיהוי מתבצע על ידי חיפוש פקודת LC_CODE_SIGNATURE בכותרת הבינארי.

PAC (Pointer Authentication Codes) (ייחודי ל-ARM)

מהו PAC?

PAC או Pointer Authentication Codes היא הגנה ייחודית הזמינה רק על חומרת ARM [16,11].

זוהי הגנה שחותמת מצביעי בקרת זרימה (function pointers וכתובות החזרה) באופן קריפטוגרפי, ובכך הופכת את חטיפת זרימת הבקרה לקשה מאוד.

מאחר שבסופו של דבר, רוב התקפות שחיתות זיכרון רוצות לחטוף את זרימת הבקרה של התוכנית, זוהי אחת ההגנות החזקות ביותר הזמינות לנו על פלטפורמות מבוססות ARM, שהן בדיוק מה ש-Apple משתמשת עבור הטלפונים שלה, ולאחרונה, גם המחשבים הניידים שלה.

איך PAC עובד?

PAC עובד על ידי הוספת האלמנטים הבאים:

- **הוראות מותאמות אישית ל-ARM Assembly** שמבצעות את אימות המצביע
- **התאמות ברמת ה-Microprocessor Circuitry** במעבד כדי לסייע בחישובים הקריפטוגרפיים הנדרשים ל-PAC לעבוד. גורם לך לתכונת האבטחה לא להיות נטל על ביצועים
- **רגיסטרים מותאמים אישית** למעבד, שבהם מאוחסנים מפתחות. כל רגיסטר הוא בגודל 128 ביט ואינו נגיש לקוד משתמש
- **הצפנה מותאמת אישית** - משתמש באלגוריתם hash חדש בשם APLPAC שתוכנן במיוחד להיות מהיר.

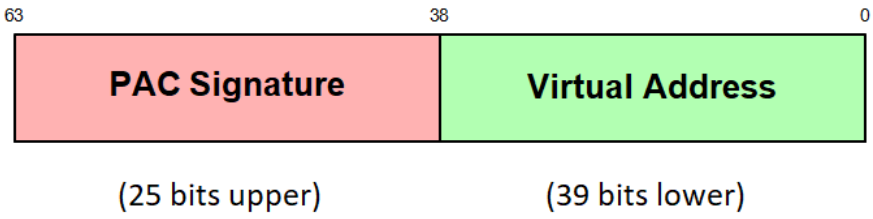
PAC עובד על ידי ניצול הביטים הלא בשימוש של כתובות מצביע (הביטים 48-63 ב-ARM64). אף מחשב אין לו באמת 2^{64} או 16 exabytes של זיכרון. אז כברירת מחדל, מצביעים נראים משהו כזה:

0x00000002fa89efa8

שבו יש הרבה אפסים שנשארים ללא שימוש. PAC עושה שימוש בזה כדי להכניס חתימה קטנה, כך שמצביע חתום עשוי להיראות כך:

0xA123A312fa89efa8

ויזואליזציה של מבנה המצביע:



חישוב החתימה

ה-hash מחושב באמצעות המשוואה הבאה:

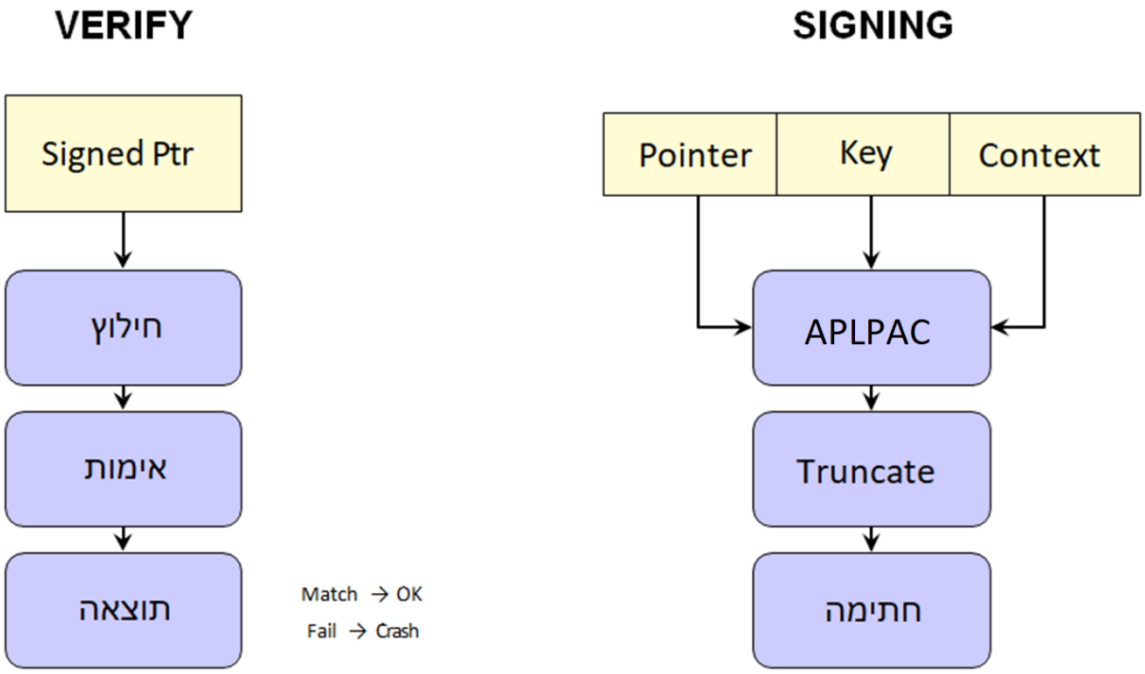
truncate(APLPAC_encrypt(key, pointer, context))

כאשר:

- **truncate** - אנחנו לא יכולים להכניס את כל התוצאה לתוך הביטים המעטים שיש לנו, אז אנחנו קוטעים ל-16 או 24 הביטים הזמינים
- **APLPAC (Apple PAC)** - אלגוריתם ה-hash המותאם של ARM, אלגוריתם צופן בלוק הניתן לכיוונון (tweakable block cipher). לכאורה מבוסס על PRICE.

- **key** - מפתח סודי היושב ברגיסטרים שיכול להיות מוגדר רק על ידי הליבה
- **context** - ערך נוכחי של מצביע המחסנית בד"כ (למרות שיש עוד Context-ים), הרעיון הוא שהיה ניתן לקרוא למצביע הפונקציה רק בהקשר של מסגרת המחסנית שבה הוא שוכן. הנ"ל מונע לעשות שימוש חוזר במצביע חתום ממקום אחר בתוכנית. המצביע החתום לא יעבוד מחוץ למסגרת המחסנית הנכונה, החתימה המחושבת לא תתאים, והתהליך יקרוס.

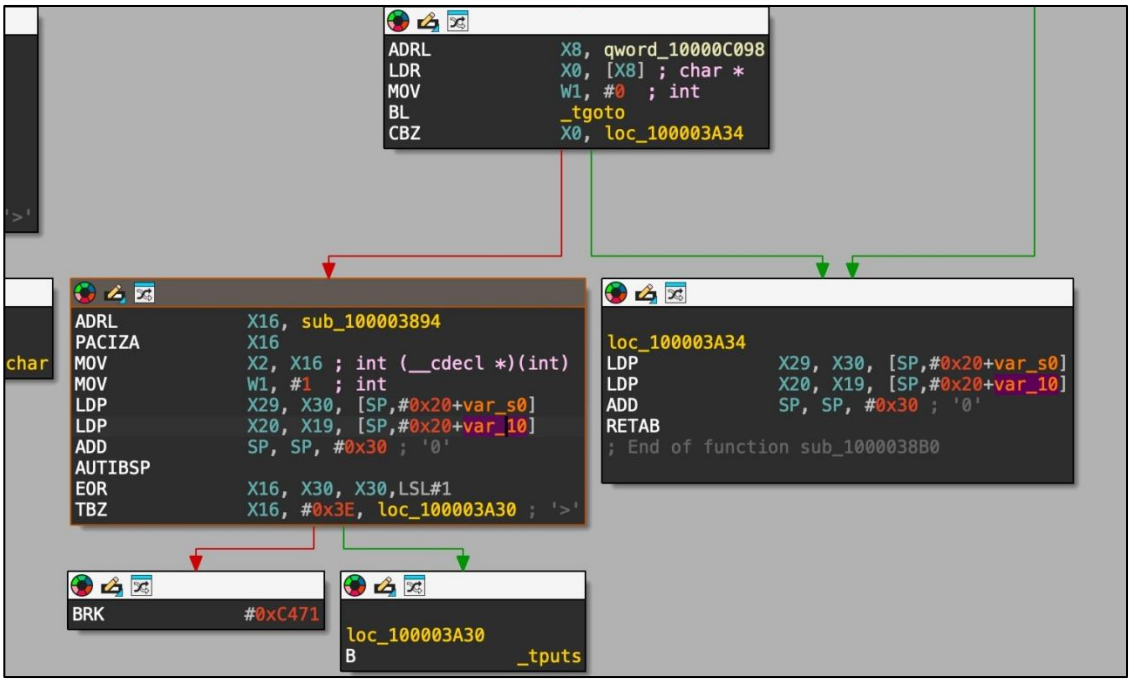
תרשים זרימה של PAC:



חשוב לציין: PAC חותם בעיקר מצביעי הוראות, כתובות החזרה ומצביעי פונקציות. מצביעי נתונים יכולים להיחתם עם PAC בחלק מהיישומים, אך זה מוגבל על ידי אילוצי שפת C (arithmetic, pointer, casts, pointer-to-integer וכו') למרבה הצער, בגלל האופן שבו מפרט C מוגדר, אי אפשר לחתום את כל המצביעים מבלי לשבור הכל לחלוטין.

זיהוי PAC

להלן תמונה של איך PAC נראה בתוכנית (/bin/ls):



כפי שאתם יכולים לראות, אנחנו מקבלים הוראות ARM חדשות ומגניבות כמו:

- PACZIA - חתום מצביע עם הקשר של "0"
- AUTIBSP - אמת מצביע באמצעות מפתח B

אז מנגנון הזיהוי רק צריך לזהות את ההוראות החדשות. או אם ה-disassembler לא עודכן לטיפול בהוראות החדשות הללו, אנחנו יכולים פשוט לחזור לסמלים.

הזיהוי מתבצע על ידי בדיקת:

- סוג המעבד - חייב להיות ARM64
- תת-סוג המעבד - בדיקה עבור CPU_SUBTYPE_ARM64E
- דגל ה-ABI PtrAuth בכותרת הבינארי

הכלי machsec[21] מזהה PAC באמצעות חיפוש הוראות ARM64 ספציפיות ובדיקת סמלי PAC בבינארי.

MTE/EMTE (Memory Tagging Extension) - (ייחודי ל-ARM)

מהו MTE/EMTE/MIE?

Memory Tagging Extension או MTE היא הגנה חדשה לגמרי ש-Apple/ARM פיתחו כדי להגן על המצביעים ש-PAC לא יכול [14,15,19]. שניהם משתמשים בעיקרון שיש ביטים לא בשימוש בכתובת של כל מצביע שאנחנו יכולים לנצל, אבל שם נגמרות הדמיון.

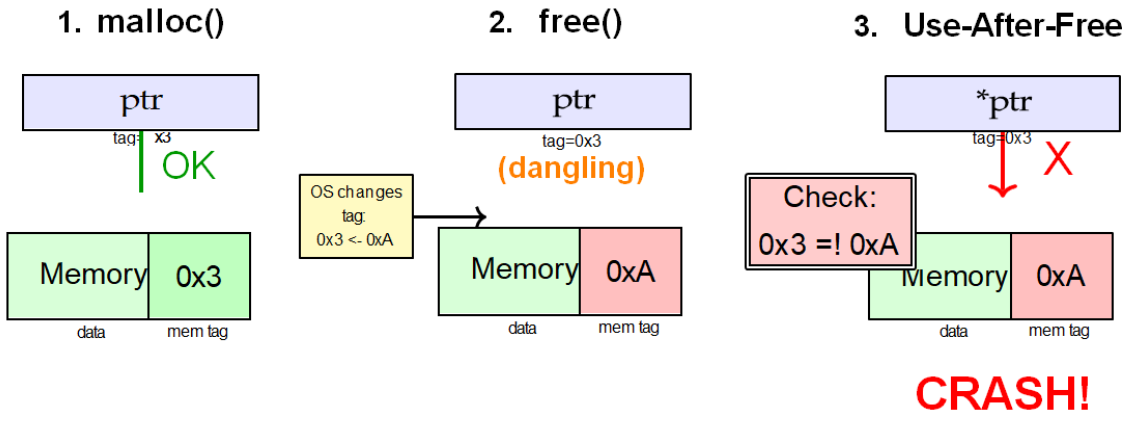
זמינות: MTE זמין כרגע רק בשבבים A17 Pro ומעלה (iPhone 15 Pro) וב-M3+ ומעלה. המינוח המדויק משתנה: Apple מכנה את היישום שלה EMTE (Enhanced MTE או Memory MIE). ו-Extension Integrity במקומות שונים בתיעוד.

איך MTE עובד?

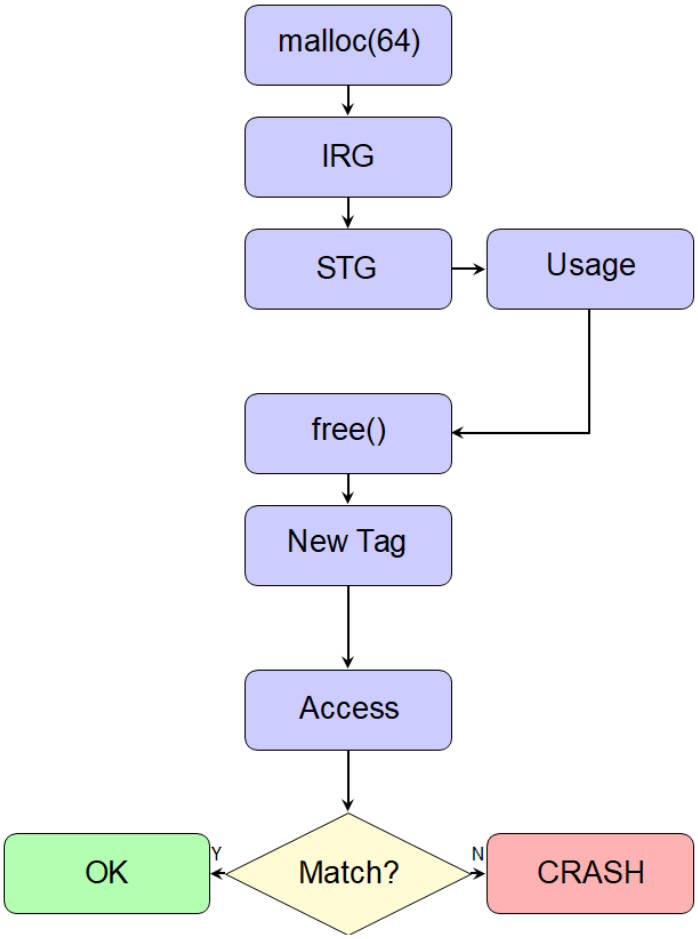
תג אקראי בן 4 ביטים ממחולל חומרה RNG מתווסף גם למצביע וגם נשמר באזור זיכרון ייעודי לתגים. MTE משתמש באזור זיכרון נפרד שנקרא Memory Tag - לכל 16 בתים של זיכרון רגיל, יש 4 ביטים (חצי-בית) של תג זיכרון. זיכרון זה מנוהל על ידי החומרה ומאפשר לשמור תגים עבור כל הקצאה. כאשר מצביע משוחזר (dereferenced), החומרה משווה אוטומטית את התג במצביע לתג בזיכרון התגים - אם הם לא תואמים, הגישה נכשלת מיידית. בואו ניתן דוגמה מעשית לאיך MTE יעצור באג UAF (Use-After-Free). הנה קטע הקוד שלנו:

```
char *ptr = malloc(64); // alloc chunk on heap size 64
ptr[0] = 'A';    //  set  value
free(ptr);       // mark chunk as empty
ptr[0] = 'B';    // UAF bug!
```

ויזואליזציה של תהליך MTE:



תרשים זרימה מפורט:



הוראות ARM64 ל-MTE:

MTE מוסיף הוראות חדשות ל-ARM64:

- IRG - פעולת Insert Random Tag, יצירת תג אקראי
- STG - פעולת Store Allocation Tag, שמירת תג הקצאה
- ST2G - פעולת Store Allication Tags (Double) - שמירת תגי הקצאה (כפול)
- STZG - פעולת Store Allocation Tag and Zero - שמירת תג הקצאה ואיפוס
- STZ2G - פעולת Store Allocation Tags and Zero (Double) - שמירת תגי הקצאה ואיפוס (כפול)
- LTG - פעולת Load Allication Tag - טעינת תג הקצאה
- LDG - פעולת Load Allocation Tag - טעינת תג הקצאה
- ADDG - פעולת Add with Tag - חיבור עם תג
- SUBG - פעולת Subtract with Tag - חיסור עם תג
- GMI - פעולת Tag Mask Insert - הכנסת מסכה
- SUBP / SUBPS - פעולת Subtract Pointer - חיסור מצביע (משמש עם MTE)

זיהוי MTE

זיהוי MTE דרש עבודת מחקר, מכיוון שזו הגנה חדשה. היה צורך לחלץ את מנוע JavaScriptCore מ-1.18 iOS על ה-iPhone החדש מכיוון שזה בין הדברים היחידים שקומפלו עם ההגנה הזו כרגע.

הזיהוי מתבצע על ידי סריקת ה-disassembly של בינארי ARM64 לחיפוש הוראות MTE ספציפיות כגון irg, subg, addg, ldg, st2g, stg, ו-gmi. נוכחות של הוראות אלו מעידה על שימוש ב-MTE. שיטה זו היא הדרך היחידה לזהות MTE במהימנות, מכיוון שההוראות הללו חדשות וספציפיות לחומרה.

ARC (Automatic Reference Counting) (רלוונטי ל-Objective-C/Swift)

מהו ARC?

ARC או Automatic Reference Couting, הוא הגנה זמינה בשפות Swift / Objective-C שמבצעת ספירת הפניות (Reference Counting) על מנת להקשות על באגי Use-After-Free ב-Objective-C ו-Swift, מאחר שכל האובייקטים והמבני הנתונים ברמה גבוהה שאתה יוצר מאוחסנים ב-heap.

איך ARC עובד?

ARC עוקב אחרי מספר ההפניות לכל אובייקט:

- כאשר נוצרת הפניה חדשה לאובייקט, המונה עולה
- כאשר הפניה יוצאת מ-scope, המונה יורד
- כאשר המונה מגיע לאפס, האובייקט משוחרר אוטומטית

הנ"ל שונה מ-Garbage Collection מלא, ARC עובד בזמן קומפילציה ולא דורש תהליך רקע בזמן ריצה.

דוגמה:

```
// Objective-C with ARC
MyObject *obj = [[ MyObject alloc] init]; // refcount = 1
MyObject *obj2 = obj;     // refcount = 2
obj = nil;        // refcount = 1
obj2 = nil;       // refcount = 0, object
        ' → freed
```

זיהוי ARC

ניתן לזהות ARC על ידי חיפוש פונקציות ניהול זיכרון אופייניות של Objective-C:

- objc_retain - הגדלת מונה הפניות
- objc_release - הקטנת מונה הפניות
- objc_autorelease - שחרור אוטומטי דחוי
- objc_retainAutorelease - שמירה ושחרור אוטומטי

- objc_storeStrong - אחסון עם שמירה חזרה
- objc_loadWeakRetained - טעינה עם שמירה חלשה

נוכחות של פונקציות אלה בבינארי מעידה על שימוש ב-ARC.

סיכום

במסמך זה סקרנו את המיטיגציות העיקריות שקיימות במערכות הפעלה מבוססות XNU ראינו איך כל הגנה עובדת, למה היא חשובה, ואיך ניתן לזהות אותה בבינאריים באמצעות הכלי machsec.

המיטיגציות העיקריות שסקרנו:

- **כנריות** - הגנה על המחסנית מפני גלישת מאגר
- **PIE** - אקראיות מיקומי קוד
- **NX** - סימון אזורי זיכרון כלא ניתנים להרצה
- **RPATH/RUNPATH** - זיהוי נתיבי ספריות לא בטוחים
- **FORTIFY** - החלפת פונקציות לא בטוחות בגרסאות מאובטחות
- **UBSAN/ASAN** - כלי אבחון לזיהוי באגים
- **CFI** - שלמות זרימת בקרה
- **הסרת סמלים** - הקשחה על ידי הסרת מידע
- **Heap Cookies הגנה על הערימה**
- **הגנת גלישת מספרים שלמים** - מניעת באגי Integer Overflow
- **Sandbox** - בידוד תהליכים
- **Hardened Runtime/SIP/AMFI -** מערכת ברמת הגנות חבילת
- **חתימת קוד** - אימות אותנטיות של בינאריים
- **PAC** - אימות קריפטוגרפי של מצביעים (ייחודי ל-ARM)
- **MTE/EMTE** - תיוג זיכרון לזיהוי באגים (ייחודי ל-ARM)
- **ARC (Objective-C/Swift)** - ניהול זיכרון אוטומטי

מערכות ההפעלה של Apple מציעות את רמת ההגנה המתקדמת ביותר בשוק כיום, במיוחד בזכות השילוב של הגנות חומרה ייחודיות כמו PAC ו-MTE המובנות במעבדי ARM של Apple Silicon. הבנת ההגנות הללו חיונית לכל חוקר אבטחה העוסק בפלטפורמות אלה.

על המחבר

אדי צלוליכין, בן 21. נמצא בתחום כבר כמעט שש שנים, בוגר יחידת 8200, עוסק במחקר הגנתי וחקר חולשות. ניתן ליצור איתי קשר בלינקדאין

מקורות מידע

1. Phrack. “Smashing the Stack for Fun and Profit”. In: Phrack Magazine 7.49 .(1996) Article ,14 by Aleph One. http://phrack.org/issues/49/14.html
2. M. Abadi et al. “Control-flow integrity”. In: Proceedings of the 12th ACM conference on Computer and communications security. ACM. ,2005 pp. -340.353
3. Singh. Mac OS X Internals: A Systems Approach. Addison-Wesley Professional,.2006
4. OS X ABI Mach-O File Format Reference. Apple Inc. .2009 https://github.com/aidansteele/osx-abi-macho-file-format-reference
5. T. K. Shawn et al.checksec.sh - Bash script to check security properties. Open Source Tool. 2009, https://github.com/slimm609/checksec.sh
6. Francillon, D. Perito, and C. Castelluccia. “Return-oriented programming without returns on ARM”. In: Black Hat USA. 2010
7. R. Roemer et al. “Return-oriented programming: Systems, languages, and appli-cations”. In: ACM Transactions on Information and System Security (TISSEC). Vol. .15 .1 ACM, .2012
8. L. Szekeres et al. “SoK: Eternal War in Memory”. In: 2013 IEEE Symposium on Security and Privacy. IEEE. ,2013 pp. -48.62
9. J. Levin. Apple Mobile File Integrity. *OS Internals, Volume I: User Mode. .2016
10. Q. S. Advisory. The Stack Clash. .2017 https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt
11. Q. T. Inc. “ARMv8.3 Pointer Authentication”. In: ARM Architecture Reference Man-ual ARMv8. ARM Limited, 2017 https://developer.arm.com/documentation/102433/0100/Pointer-authentication
12. J. Levin. *OS Internals, Volume III: Security & Insecurity. Technologeeks.com,.2017
13. Hardened Runtime. Tech. rep. Apple Inc. 2018, https://developer.apple.com/documentation/security/hardened_runtime
14. ARM Memory Tagging Extension. ARM Limited .2019, https://developer.arm.com/documentation/102925/0100/

15. Armv8.5-A Memory Tagging Extension. ARM Limited .2019, https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety
16. H. Liljestrand et al. “PAC it up: Towards pointer integrity using ARM pointer authentication”. In: 28th USENIX Security Symposium .2019
17. N. Williamson. SockPuppet: A Walkthrough of a Kernel Exploit for iOS 12.4. Google Project Zero Blog. 2019, https://googleprojectzero.blogspot.com/2019/12/sockpuppet-walkthrough-of-kernel.html
18. Towards the next generation of XNU memory safety: kalloc_type. Tech. rep. Apple Inc. 2021, https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/
19. Apple Silicon Security. Apple Inc. 2023, https://support.apple.com/guide/security/apple-silicon-security-sec87716a080/web
20. Apple Platform Security. Apple Inc.2024, https://support.apple.com/guide/security/welcome/web
21. gracecondition. machsec - Security mitigation detection tool for Mach-O binaries. Open Source Tool. 2024, https://github.com/gracecondition/machsec
22. gracecondition. machsec Documentation - XNU Security Mitigations. Personal Blog. 2024, https://gracecondition.github.io/posts/machsec-documentation/
23. https://i.blackhat.com/BH-US-23/Presentations/US-23-Zec-Apple-PAC-Four-Years-Later.pdf