Technologic Papers

MTE כמיקרוסקופ

מאת ניר אברהם והו קי

הקדמה

בחודשים האחרונים החל להופיע בצינור הניתוח שלנו סוג חדש של אות, קריסות קרנל (kernel panics) שמעולם לא נתקלנו בהן בעבר. לא משום שהן מייצגות מתקפה חדשה, אלא משום שמנגנון אכיפת שלמות הזיכרון (Memory Integrity Enforcement – MIE) של Apple חושף כעת באגים שבעבר לא הותירו אחריהם כל עקבה. מקרה אחד משך במיוחד את תשומת לבנו: קריסת קרנל שנגרמה כתוצאה ממצב מירוץ (race condition) עמוק בתוך מחסנית הרשת של iOS. בכל מכשיר שאינו תומך בהרחבת תיוג הזיכרון (Memory Tagging Extension – MTE), הבאג הזה היה נותר בלתי נראה לחלוטין. לא הייתה מתרחשת קריסה, לא היה נוצר לוג, ולא הייתה כל ראיה לכך שהאירוע התרחש.

התופעה הזו היא שהניעה אותנו לכתוב את הפוסט הזה. לדעתנו, MTE של ARM הוא הרבה יותר ממנגנון הגנה אבטחתי, הוא מסמן שינוי תפיסתי באופן שבו אנו מאתרים ומבינים פגיעויות בקרנל, ומחייב את קהילת המחקר לאמץ דרכי עבודה וכלי מחקר חדשים. בקצרה: הרחבת תיוג הזיכרון (MTE) של ARM, והמימוש שלה באפל (Apple), MIE, מסוגלים לזהות פגיעויות בטיחות זיכרון כבר ברמת החומרה, עוד לפני שמתרחש corruption של הזיכרון (memory corruption). בפוסט נסביר כיצד הטכנולוגיה פועלת לעומק, נמחיש כיצד היא חושפת פגיעויות מסוג use-after-free הנגרמות כתוצאה ממצבי מירוץ , פגיעויות שהיו נותרות בלתי נראות ללא MTE , ונטען כי יש לראות ב-MTE לא רק מנגנון הגנה, אלא גם כלי מחקר רב־עוצמה.

ההבטחה של תיוג זכרון בחומרה

פגיעויות memory safety, כגון buffer overflow, use-after-free ו-type confusion, ממשיכות להיות מחלקת הפגיעויות הנפוצה והמנוצלת ביותר בעולם התוכנה. הן עומדות בבסיסן של כמעט כל שרשראות הניצול (exploit chains) המתקדמות , החל מ-jailbreaks ועד לתוכנות ריגול ברמה מדינתית כמו Pegasus. למרות עשרות שנים של מנגנוני הגנה מבוססי תוכנה, כגון ASLR, stack canaries ו-CFI, תוקפים ממשיכים למצוא דרכים לעקוף אותם. הסיבה לכך פשוטה: מנגנוני הגנה מבוססי תוכנה הם, מטבעם, משחק הגנה שבו התוקף הוא זה שמכתיב את הכללים.

הרחבת תיוג הזיכרון (Memory Tagging Extension – MTE) של ARM, שהוצגה לראשונה בשנת 2019 כחלק מ-ARMv8.5-A, מציעה גישה שונה מן היסוד: בטיחות הזיכרון נאכפת ברמת החומרה. במקום לנסות לזהות ניצול לאחר שכבר התרחש קוראפשן של הזיכרון (memory corruption), MTE מצמיד תג (tag) לכל הקצאת זיכרון ומאמת את התג בכל גישה לזיכרון, במהירות החומרה וללא תקורת תוכנה עבור פעולת האימות עצמה.

בספטמבר 2025 השיקה Apple את מנגנון אכיפת שלמות הזיכרון (Memory Integrity Enforcement – MIE) בסדרת iPhone 17. המנגנון מבוסס על גרסה משופרת של MTE, שפותחה בשיתוף עם ARM. השקה זו הייתה שיאו של מאמץ הנדסי שנמשך כחמש שנים וכלל את צוותי הסיליקון, מערכת ההפעלה וה-frameworks של Apple. בחודשים שחלפו מאז החלו להצטבר עדויות מהשטח שאיששו את מה שחוקרי אבטחה העריכו במשך שנים: מעבר ליכולתו למנוע ניצול של פגיעויות, MTE חושף מחלקות שלמות של באגים שבעבר נותרו בלתי נראים.

בפוסט זה נצלול לעומק אופן הפעולה של MTE, נבחן את ההרחבות ש-Apple בנתה מעליו, ונסביר מדוע תיוג זיכרון הופך לכלי חיוני לא רק להגנה על מערכות, אלא גם למחקר פגיעויות.

MTE מתחת למכסה המנוע: כיצד תגיות מגינות על הזיכרון

המנגנון המרכזי

MTE מתבסס על תכונה בארכיטקטורת ARM64 בשם TBI -Top Byte Ignore. בכתובת וירטואלית של 64 סיביות, רק הביטים התחתונים משמשים בפועל לתרגום הכתובת. הבייט העליון (ביטים 63:56) אינו נלקח בחשבון על ידי יחידת ניהול הזיכרון (MMU), ולכן ניתן לנצל אותו לאחסון מטא־נתונים. MTE משתמש בארבעה ביטים מתוך אותו בייט עליון (ביטים 59:56) כדי לאחסן תג (tag) שערכו נע בין 0 ל-15.

בנוסף, עבור כל גרנולה (granule) של 16 בתים בזיכרון הפיזי, המיושרת לגבול של 16 בתים, החומרה שומרת תג בן 4 ביטים במבנה זיכרון ייעודי בשם Tag RAM. זהו מבנה זיכרון נלווה (sidecar memory) שהמעבד בודק בכל פעולת טעינה (load) או כתיבה (store).

מנגנון הפעולה פשוט למדי: בכל פעם שהמעבד מבצע פעולת load או store, הוא מחלץ את התג מהמצביע (pointer) ומשווה אותו לתג המאוחסן ב-Tag RAM עבור כתובת היעד. אם שני התגים תואמים, הגישה לזיכרון מתבצעת כרגיל. אם הם אינם תואמים, החומרה מייצרת Tag Check Fault.

```
Pointer:    0xF6FFFE1979509XXX
            ^^
            Tag = 0xF6 (embedded in pointer's upper byte)

Memory:     [16-byte granule at 0xFFFFFE1979509000]
            Tag RAM entry = 0xF7

On load/store: 0xF6 ≠ 0xF7 → TAG CHECK FAULT
```

מחזור החיים של תג (Tag Lifecycle)

התגים משתנים בכל הקצאה מחדש של זיכרון. כאשר kalloc() מקצה אזור זיכרון, המקצה (allocator) מבצע את הפעולות הבאות:

- בוחר ערך תג (tag) אקראי בטווח 0–15 עבור אזור ההקצאה.
- כותב את התג ל-Tag RAM עבור כל גרנולה (granule) בהקצאה, באמצעות ההוראות STG או ST2G.
- מחזיר מצביע (pointer) שהתג מוטמע בביטים העליונים שלו.

כאשר kfree() משחרר את אזור הזיכרון, המקצה כותב תג חדש ושונה ל-RAM. כתוצאה מכך, כל dangling pointer שעדיין נושא את התג הישן לא יתאים עוד לתג המאוחסן ב-RAM, והחומרה תעצור את הגישה באופן מיידי.

התובנה המרכזית

מאחר שגודל התג הוא 4 ביטים בלבד, קיימים 16 ערכי תג אפשריים. משמעות הדבר היא שהתנגשות אקראית של תג (tag collision), שבה מצביע מיושן (stale pointer) "פוגע במקרה" בתג החדש, מתרחשת בהסתברות של 1 ל-16 (כ-6.25%).

חשוב להדגיש: אין מדובר במנגנון אבטחה קריפטוגרפי. MTE הוא מנגנון הסתברותי (probabilistic), ולא דטרמיניסטי (deterministic). עם זאת, גם הסתברות של 93.75% לקריסה בכל ניסיון הופכת פיתוח של exploit אמין למשימה קשה במיוחד, בייחוד כאשר מדובר בשרשראות ניצול (exploit chains) מרובות שלבים, שבהן כל שלב חייב להצליח כדי שהניצול כולו יושלם.

לדוגמה, בשרשרת ניצול בת שלושה שלבים, שבה כל שלב דורש התאמה של תג, הסתברות ההצלחה הכוללת יורדת לכ- $(1/16)^3 \approx 0.024\%$ זו אחת הסיבות המרכזיות לכך ש-MTE יעיל במיוחד נגד exploit chains מלאות, אף שההסתברות להצלחה בכל שלב בודד נראית, במבט ראשון, לא נמוכה במיוחד.

בדיקה סינכרונית לעומת בדיקה א־סינכרונית

MTE תומך בשני מצבי בדיקה:

- מצב סינכרוני (Synchronous Mode): חריגת Tag Check Fault נוצרת מיד עם ביצוע הוראת ה-load או ה-store שגרמה לכשל. כתובת ה-PC המתועדת בנתוני החריגה מצביעה במדויק על ההוראה הבעייתית. מצב זה מספק מידע מדויק במיוחד לצורכי דיבוג, אך כרוך בעלות ביצועים מסוימת.
- מצב א־סינכרוני (Asynchronous Mode): כשלים בבדיקת התג מתועדים, אך החריגה אינה נוצרת באופן מיידי. גישה זו מפחיתה את תקורת הביצועים, אך מקשה על זיהוי מדויק של מקור הבעיה.

Apple בחרה להפעיל את MIE במצב סינכרוני. לעומתה, המימוש של Google במכשירי Pixel משתמש במצב א־סינכרוני עבור הקרנל, ובגישה משולבת (mixed mode) במרחב המשתמש (user space). לבחירה זו יש השלכות משמעותיות הן על רמת ההגנה שמספק המנגנון והן על התועלת שלו לצורכי מחקר.

מ-MTE ל-MIE: הגישה רחבת-המחסנית של Apple

כדי לאפשר MTE בשבבים שלהם, Apple ארכיטקטרה מחדש את כל המחסנית סביבו, ויצרה מערכת הגנה תלת-שכבתית שהיא מכנה מנגנון אכיפת שלמות הזיכרון (MIE).

שכבה 1: מקצי זיכרון מאובטחים

עוד לפני ש-MTE נכנס לתמונה, מקצה האזורים (zone allocator) של Apple מספק הגנה משמעותית. מקצה הקרנל של iOS מארגן את הזיכרון בבאקטים ספציפיים לפי הטיפוס ("אזורים", zones), כך שאובייקטים מטיפוסים וגדלים שונים מבודדים זה מזה. כאשר זיכרון משוחרר, הוא מאופס ומוחזר לאזור המתאים. הדבר מקשה משמעותית על heap spraying.

זהו קו ההגנה הראשון, והוא עובד בכל המכשירים, כולל חומרה ישנה יותר ללא תמיכה ב-MTE.

שכבה 2: MTE משופר (EMTE)

הערכת המפרט המקורי של MTE שביצעה Apple זיהתה חולשות שנחשבו בעיניה בלתי מקובלות לפריסה הגנתית בזמן אמת. בשיתוף עם ARM, הן פיתחו במשותף את MTE משופר (EMTE), הידוע גם בשם FEAT_MTE4 במפרט של ARM.

השיפורים המרכזיים כוללים:

- בדיקת תגית קנונית (Canonical tag checking): ב-MTE הסטנדרטי, גישה לזיכרון לא-מתויג (כגון משתנים גלובליים) מתוך אזור מתויג אינה נבדקת. EMTE מוסיף בדיקה למעברים אלה.
- דיווח תקלות משופר: כל הסיביות שאינן כתובת מדווחות בעת תקלה, מה שמעניק לקרנל מידע רב יותר לאבחון.
- בדיקת תגית לאחסון בלבד (store-only): מצב נוסף שבו רק פעולות אחסון נבדקות, שימושי בהקשרים ספציפיים הרגישים לביצועים.

שכבה 3: אכיפת סודיות תגיות (TCE)

חולשה קריטית בכל מערכת מבוססת-תגיות היא שאם תוקף יכול ללמוד את ערך התגית, הוא יכול לזייף מצביע תואם. מחקר כמו מתקפת TikTag הדגים שניתן להדליף תגיות MTE דרך ערוצי-צד של ביצוע ספקולטיבי (speculative execution side channels). Apple פיתחה את אכיפת סודיות התגיות (TCE) כדי לטפל בכך, כולל אמצעי הגנה נגד מתקפות מסוג Spectre V1 ב"עלות CPU אפסית כמעט".

אופטימיזציה ברמת הסיליקון

שבב ה-A19 תוכנן מהיסוד תוך התחשבות בעומסי עבודה של MTE. אפל (Apple) מידלה את הביקוש המדויק לבדיקת תגיות של מערכת ההפעלה כולה, ועיצבה את הסיליקון כדי לספק אותו. במקום לקבל קנס ביצועים, Apple עיצבה את החומרה כדי לחסל אותו.

היקף הפריסה

במכשירי A19 מנגנון MIE פעיל באופן קבוע עבור הקרנל ועבור יותר מ-70 תהליכי user space. מפתחי צד שלישי יכולים לאפשר תמיכה באמצעות entitlement ייעודי ב-Xcode. נכון למועד כתיבת שורות אלו, MIE זמין רק בשבבי A19 ו-A19 Pro, המשולבים במכשירי iPhone 17, iPhone 17 Pro, iPhone 17 Pro Max ו-iPhone Air. נכון לעכשיו, Apple טרם הכריזה על תמיכה ב-MIE בשבבי סדרת M למחשבי Mac.

כשהתגים חושפים רוח רפאים: מקרה בוחן של מצב מירוץ

כדי להמחיש מדוע MTE משנה את כללי המשחק עבור חוקרי אבטחה, נבחן מקרה אמיתי שבו נתקלנו: קריסת קרנל במכשיר iPhone 17 Pro המריץ את iOS 26.4, שנגרמה כתוצאה מפגיעות מסוג use-after-free שנוצרה בעקבות מצב מירוץ (race condition) באחת מתתי־המערכות האחראיות על רכיבי הרשת בקרנל. מטעמי Responsible Disclosure, נתאר את מבנה הפגיעות ואת אופן פעולתה, אך לא את מסלול הקוד (code path) המדויק שבו היא התרחשה.

האנטומיה של מצב המירוץ

תת־המערכת שבה התגלתה הפגיעות מנהלת אובייקטי מדיניות רשת (network policy objects) עבור כל תהליך. כמו תתי־מערכות רבות בקרנל, גם היא משתמשת בשני מנגנוני נעילה שונים, שכל אחד מהם מגן על אינווריאנטים (invariants) אחרים:

- Lock A – מנגנון reader-writer lock המגן על מאגר גלובלי (עץ) המכיל את כל אובייקטי מדיניות הרשת הפעילים במערכת.
- Lock B – מנעול מסוג mutex המשויך לכל אובייקט בנפרד ומגן על המצב הפנימי של אותו אובייקט.

לא ניתן לקחת את שני המנעולים בסדר קבוע, משום שמסלולי קוד שונים לוקחים אותם בסדר הפוך. מצב כזה עלול להוביל ל-deadlock. כתוצאה מכך, קוד הזקוק לשני המנעולים נאלץ לשחרר את הראשון לפני שהוא לוקח את השני. בין שתי פעולות הנעילה הללו נוצר חלון זמן שבו אף אחד מהמנעולים אינו מגן על האובייקט, ובדיוק בחלון הזה נמצאת הפגיעות.

הפונקציה הפגיעה

במבט ראשון, הפונקציה הפגיעה מבצעת רצף פעולות שנראה סביר לחלוטין:

```c
// Pseudocode — actual function redacted
int subsystem_operation(uuid_t object_id, ...) {
    // Phase 1: look up the object under the global lock
    READ_LOCK(&global_registry);
    obj = tree_find(&global_registry, object_id);
    if (!obj) { READ_UNLOCK(&global_registry); return NOT_FOUND; }
    client = obj->owning_client;          // save references
    saved_obj = obj;                      // saved in register (stack-local)
    READ_UNLOCK(&global_registry);

    // *** UNPROTECTED WINDOW ***
    // No lock protects saved_obj here. ~5 instructions wide.

    // Phase 2: operate on the object under the per-client lock
    LOCK(&client->mutex);                 // may block if contended
    flags = saved_obj->flags;             // ← LOAD from saved_obj
    /* ... more reads from saved_obj ... */
    UNLOCK(&client->mutex);
    return OK;
}
```

דפוס העבודה הזה נפוץ מאוד בקוד קרנל: תחילה מאתרים אובייקט תחת מנעול גס (coarse-grained lock), לאחר מכן משחררים אותו, ולבסוף מבצעים את הפעולות העדינות יותר תחת המנעול של האובייקט עצמו (fine-grained lock).

מאחורי הדפוס הזה עומדת הנחה מובלעת: שהאובייקט אינו יכול להיעלם בפרק הזמן שבין שחרור המנעול הראשון לבין לקיחת המנעול השני.

אלא שהנחה זו אינה נכונה אם במהלך אותו חלון זמן תהליכון (thread) אחר יכול לשחרר את האובייקט.

נתיב ההשמדה

במקום אחר באותה תת־מערכת, קוד הניקוי (cleanup), המופעל כאשר תהליך סוגר מתאר קובץ (file descriptor) או מסיים את פעולתו, מבצע את רצף הפעולות הבא:

```c
// Pseudocode — destroyer side
void destroy_object(client, obj) {
    WRITE_LOCK(&global_registry);
    tree_remove(&global_registry, obj);    // unlink from tree
    WRITE_UNLOCK(&global_registry);

    LOCK(&client->mutex);
    kfree_type(obj_type, obj);             // ← MEMORY FREED, tag changes
    /* ... more cleanup still holding lock ... */
    UNLOCK(&client->mutex);
}
```

שימו לב לפרט הקריטי: ה-kfree מתרחש בעוד ה-mutex לכל-לקוח עדיין מוחזק. הדבר חשוב הן עבור הבאג והן עבור האופן שבו MTE תופס אותו.

השתלשלות מצב המירוץ (Race Interleaving)

כאשר בוחנים את שני מסלולי הקוד יחד, ניתן לראות כיצד מצב המירוץ מתרחש:

#	Thread A (operation)	Thread B (destroy)	State
1	save ,Find object in tree pointer in X26	,	A holds stale pointer after releasing registry lock
2	(unprotected window)	tree_remove(obj)	Object gone from tree; A's pointer still valid-looking
3	,	LOCK(client->mutex) + kfree(obj)	**MTE tag .Memory freed changes 0xF6 → 0xF7.** B still holds mutex.
4	LOCK(client->mutex) → CASA fails → block in lck_mtx_lock_contended	,	A sleeps on turnstile waiting for B
5	,	UNLOCK(client->mutex)	B releases
6	LDRB ,acquire mutex ,Wake [X26+offset]	,	**TAG CHECK FAULT**

חשוב להדגיש כי מדובר במצב מירוץ בין שני תהליכונים (threads) בלבד, אין צורך בגורם שלישי. ההתנגשות (contention) שגורמת ל-Thread A להיכנס למסלול האיטי (slow path) ולהיחסם בתוך lck_mtx_lock_contended נגרמת ישירות על ידי Thread B, שהוא גם התהליכון שמשחרר את הזיכרון.

כיצד נראית הקריסה

קריסת הקרנל הנוצרת ממירוץ זה בעלת חתימה ייחודית. הנה תצוגה מצונזרת של החלקים הרלוונטיים:

```
panic(cpu X caller 0xfffffe00XXXXXXXX):
    Kernel tag check fault (expected tagged address: 0xF7FFFEXXXXXXXXX)
    at pc 0xfffffe00XXXXXXXX, lr 0xXXXXXXXXXXXXXXXX

    x19: 0xFAFFFEXXXXXXXXX  ; client pointer (loaded under Lock A)
    x26: 0xF6FFFEXXXXXXXXX  ; STALE object pointer — tag 0xF6
    far: 0xF6FFFEXXXXXXXXX  ; faulting address, tag 0xF6
    esr: 0x0000000096000011  ; DFSC = 0x11 → Synchronous Tag Check Fault

panicked thread schedFlags: TH_SFLAG_RW_PROMOTED
    (priority promoted: 31 → 46, rw-lock contention witness)
```

ארבע תצפיות מתוך מצב האוגרים (register state) מספרות למעשה את כל הסיפור שמאחורי הבאג:

- **המצביע המיושן (stale pointer) שבאוגר X26** נושא את התג 0xF6 , התג שהוקצה בזמן ההקצאה המקורית.
- **השדה "expected tagged address"** בהודעת ה-panic מכיל את התג 0xF7 , התג החדש שנכתב ל-Tag RAM על ידי kfree לאחר שחרור הזיכרון.
- **ערך ה-LR** (שהושמט לעיל ומסומן כ-0xXXXXXXXXXXXXXXXX lr) מתפענח ככתובת החזרה מהפונקציה lck_mtx_lock_contended. פונקציה זו מופיעה רק כאשר רכישת ה-mutex נכנסת למסלול האיטי (slow path), כלומר כאשר המנעול כבר היה תפוס בזמן שהתהליכון ניסה לבצע את פעולת ה-CASA. זוהי עדות ישירה ברמת החומרה לכך ש-Thread A היה חסום על ה-mutex בדיוק בזמן שבו Thread B שחרר את הזיכרון.
- **הדגל TH_SFLAG_RW_PROMOTED** בתהליכון שקרס מצביע על כך שהתהליכון קיבל העלאת עדיפות (priority promotion) בעקבות התנגשות על reader-writer lock. זהו סימן מובהק לכך ש-Thread B ניסה לקחת WRITE_LOCK על המאגר הגלובלי, בעוד ש-Thread A עדיין החזיק ב-READ_LOCK.

כאשר מחברים את ארבעת הסימנים הללו יחד, ניתן לשחזר את ציר הזמן של מצב המירוץ מתוך לוג panic יחיד. כל אחד מהסימנים הללו זמין אך ורק משום ש-MTE לכד את הכשל באופן סינכרוני, בדיוק בהוראה שביצעה את הגישה באמצעות המצביע המיושן.

כיצד נראה אותו באג ללא MTE

אותו מצב מירוץ בדיוק מוביל לתוצאה שונה לחלוטין במכשיר שאינו תומך ב-MTE:

1. Thread A שומר מצביע מיושן (stale pointer) ומשחרר את מנעול המאגר (registry lock).
2. Thread B מסיר את האובייקט מהעץ, לוקח את ה-mutex, משחרר את האובייקט באמצעות kfree, מקצה ה-zones מאפס את תוכן הזיכרון, ולבסוף משחרר את ה-mutex.
3. Thread A מתעורר, לוקח את ה-mutex ומבצע גישה באמצעות המצביע המיושן.
4. Thread A קורא את הערך 0x00000000 מהשדה flags<-stale_obj.
5. הביטוי if (flags & FLAG_X) מוערך כ-false.
6. הפונקציה מסתיימת בהצלחה , ללא קריסה, ללא לוג וללא כל סימן לכך שהתרחשה תקלה.

למרות זאת, הבאג אכן התרחש, ובוצעה גישה לאובייקט שכבר שוחרר. אולם מכיוון שמקצה ה-zones איפס את תוכן הזיכרון, ומכיוון שמסלול הקוד הספציפי הזה מסוגל להתמודד עם קריאה של ערך אפס, לא הייתה לבאג שום השפעה נצפית.

מנקודת המבט של fuzzer, בדיקות יחידה (unit tests) ואפילו לוגי kernel panic , הבאג מעולם לא התרחש.

שחזור הבאג: בעיית הבאגים השקטים

ניסינו לשחזר את מצב המירוץ בסביבה מבוקרת. לצורך כך פיתחנו PoC היוצר את אובייקטי הקרנל הרלוונטיים מתוך אפליקציה הפועלת ב sandbox, מפעיל את מנגנון ה-qualification באמצעות content filter, ולאחר מכן משחרר את האובייקטים במהירות , בדיוק רצף הפעולות שהוביל ל-kernel panic המקורי.

במכשיר התומך ב-MTE אנו מצפים שבשלב מסוים יתרחש kernel panic. לעומת זאת, במכשירים שאינם תומכים ב-MTE, ובכלל זה סביבות מחקר וירטואליות כגון Corellium, שאינן מדמות MTE/MIE, מצב המירוץ עדיין מתרחש, אך הכשל נותר בלתי נראה:

- מקצה ה-zones מאפס את הזיכרון לאחר שחרורו, ולכן הגישה באמצעות המצביע המיושן מחזירה אפס.
- מסלול הקוד הספציפי מסוגל להתמודד עם קריאה של ערך אפס.
- לא מתרחשת קריסה, לא נוצר לוג, ולא נותר כל סימן לכך שהבאג התרחש.

זוהי המחשה מעשית וישירה לרעיון של "Dark Matter". הבאג הזה פעל, ככל הנראה, אי שם בין "מדי פעם" ל"לעיתים קרובות" במערכות ייצור במשך פרק זמן שאיננו יודעים לאמוד. אולם עד להופעת MIE, לא היה קיים מנגנון שהיה רגיש מספיק כדי להבחין בקיומו.

מסגרת האבחון: tag check fault מול data abort

עבור חוקרי פגיעויות המנתחים קריסות קרנל במכשירים שבהם MTE מופעל, סוג ה-kernel panic עצמו הופך לאחד מסימני המיון (triage) הראשונים והחשובים ביותר. הבחנה זו יוצרת מסגרת חדשה לניתוח ולאבחון של קריסות קרנל.

קריאת אוגר Exception Syndrome Register (ESR)

ה-ESR מקודד את סיבת החריגה. עבור תקלות הקשורות ל-MTE:

```
// Tag Check Fault (MTE)
ESR = 0x0000000096000011
  EC[31:26]  = 0x25 → Data Abort from current EL
  DFSC[5:0]  = 0x11 → Synchronous Tag Check Fault
```

```
// Data Abort (non-MTE)
ESR = 0x0000000096000021
  EC[31:26]  = 0x25 → Data Abort from current EL
  DFSC[5:0]  = 0x21 → Alignment Fault
```

מה כל אחד אומר לכם

TAG CHECK FAULT	DATA ABORT
Stage: Pre-corruption .MTE blocked the access. **Exploitability:** Often low .The corruption needed for exploitation was prevented. **Research value:** Very high . Reveals bugs that are otherwise invisible. **Implication:** A real vulnerability exists ,but MTE has mitigated it.	**Stage:** Post-corruption .Memory is already damaged. **Exploitability:** Potentially high . Attacker may control corrupted data. **Research value:** High ,but harder to trace root cause. **Implication:** More urgent .May be exploitable on non-MTE devices.

שחזור תגים מתוך לוגי Kernel Panic

בעת ניתוח של Tag Check Fault, ניתן לשחזר את השתלשלות האירועים מתוך מצב האוגרים (register state):

- **זיהוי המצביע המיושן (stale pointer):** אתר את האוגר המכיל את המצביע לאובייקט. ארבעת הביטים העליונים של התג (upper nibble) מייצגים את התג הישן שנשא המצביע.
- **זיהוי התג הצפוי:** הודעת ה-kernel panic עצמה כוללת את השדה expected tagged address, המכיל את התג הנוכחי המאוחסן ב-Tag RAM.
- **אימות ביצוע retagging:** אם שני התגים שונים בערך קטן, הדבר תואם את מנגנון ה-retagging הרציף של המקצה (allocator). אם ההפרש נראה אקראי, סביר שהאובייקט כבר שוחרר, והגרנולה (granule) הוקצתה מחדש עבור אובייקט אחר.
- **בדיקת דגלי התזמון (Scheduling Flags):** דגלים כגון TH_SFLAG_RW_PROMOTED בתהליכון שקרס מהווים אינדיקציה חזקה לכך שבזמן הקריסה הייתה פעילות נעילה מקבילית (concurrent locking), כלומר שתהליכון נוסף היה מעורב , חתימה אופיינית של מצב מירוץ.
- **פענוח ה-LR:** אם אוגר ה-LR (Link Register) מצביע לפונקציית slow path מוכרת של מנגנון נעילה (לדוגמה, פונקציה המסתיימת ב-_contended), ניתן להסיק שה-mutex היה תפוס ברגע שבו התרחש הכשל. זוהי עדות ישירה לכך שתהליכון אחר החזיק במנעול באותו זמן.
- **איתור הקריאה ל-()free**: עבוד לאחור מהפונקציה שבה התרחשה הקריסה כדי לזהות איזה מסלול קוד שחרר את האובייקט, ואיזה מסלול המשיך להחזיק מצביע מיושן אליו. כך ניתן לשחזר את מצב המירוץ שהוביל לפגיעות.

MTE ככלי מחקר: חשיפת הבלתי נראה

מבחינה היסטורית, MTE מוסגר כאמצעי הגנה, מנגנון המיועד להקשות על אקספלויטציה. אך ניסיון המחקר שלנו מצביע על תפקיד חשוב לא פחות: MTE ככלי לאיתור באגים.

בעיית "החומר האפל"

נבחן את מחלקת הפגיעויות שתוארה בפרק 4: מצבי מירוץ המובילים לפגיעויות מסוג use-after-free, כאשר מקצה ה-zones מסתיר את תסמיני הבאג. ללא MTE, פגיעויות מסוג זה:

- אינן גורמות לקריסה.
- אינן מייצרות לוג.
- אינן משאירות כל עקבה פורנזית.
- עוברות בהצלחה את כל מערכי הבדיקות הקיימים.
- עלולות לשרוד במשך שנים בסביבת ייצור (production).

אלו הם ה-"Dark Matter" של עולם באגי הקרנל. הם קיימים, וייתכן שמספרם אף גדול. אולם בהיעדר מנגנון המסוגל לחשוף אותם, הם נותרים בלתי נראים לכלי הניתוח המסורתיים , לרבות fuzzers, אנליזה סטטית (static analysis), סקירות קוד ידניות ובדיקות דינמיות המבוססות על sanitizers.

MTE משנה את כללי המשחק. באמצעות תיוג של כל הקצאת זיכרון ובדיקת כל גישה לזיכרון ברמת החומרה, הוא הופך באגים בלתי נראים לכשלים הניתנים לזיהוי. התהליך הזה מתרחש באופן רציף, במערכות ייצור ועל כל מכשיר שבו MTE מופעל.

ההשלכות על מתודולוגיית מחקר הפגיעויות

1. מחלקות חדשות של באגים הופכות לברות גילוי

פגיעויות מסוג use-after-free הנגרמות כתוצאה ממצבי מירוץ בתתי־מערכות קרנל המטפלות באירועים בתדירות גבוהה , כגון רכיבי רשת, מערכת הקבצים או מנגנוני IPC , עשויות להתבטא רק בתנאי תזמון מסוימים. במקרים רבים, fuzzer לא יצליח לשחזר את השזירה (interleaving) המדויקת של התהליכונים. לעומת זאת, MTE ילכוד את הכשל ברגע שהוא יתרחש לראשונה על כל מכשיר שבו הוא מופעל.

2. ניתוח לוגי קריסה הופך לכלי מחקר

עבור ארגונים המנתחים לוגי קריסה בהיקף רחב , כגון יצרני מכשירים (OEMs), חברות אבטחה וספקי MDM , אירועי Tag Check Fault מהווים ערוץ מידע חדש. כל אירוע כזה הוא אינדיקציה ישירה לקיומה של פגיעות memory safety.

3. הערכת יכולת הניצול משתנה

פגיעות המובילה ל-Tag Check Fault במכשיר התומך ב-MTE עדיין עשויה להיות ניתנת לניצול במכשירים שאינם תומכים ב-MTE. לכן, גם אם MTE מנע את התוצאה החמורה ביותר, הפגיעות עדיין ראויה לדיווח ולתיקון.

במילים אחרות, תהליך ה-triage המבוסס על MTE מספק דרך מהירה לאתר פגיעויות המשפיעות על בסיס ההתקנות הגדול בהרבה של מכשירים שאינם תומכים ב-MTE , ושבהם אותן פגיעויות עשויות להיות ניתנות לניצול.

מגבלות של פלטפורמות מחקר

ישנו גם שיקול מעשי: לא כל סביבות המחקר תומכות ב-MTE.

לדוגמה, Corellium, אחת מפלטפורמות המחקר הנפוצות ביותר לחקר קרנל iOS , אינה תומכת כיום באמולציה של MTE/MIE. המשמעות היא:

- באגים המתבטאים אך ורק כ-Tag Check Fault אינם ניתנים לשחזור ב-Corellium.
- מקצה ה-zones ממשיך לאפס זיכרון גם באמולטור, ולכן פגיעויות use-after-free ממשיכות להיות מוסתרות.
- לעיתים נדרש להזריק באופן ידני מצביעים פגומים (corrupted pointers) כדי לאלץ את התרחשות הכשל לצורכי בדיקה.
- לצורך אימות מקיף של פגיעויות, מכשירים פיזיים התומכים ב-MTE הופכים לכלי מחקר הכרחי.

שינוי פרדיגמה במחקר פגיעויות

MTE הופך את מודל העבודה המסורתי של מחקר פגיעויות על פניו. במקום הגישה הקלאסית , "מצא את הבאג, ואז בדוק אם ניתן לנצל אותו", MTE מאפשר גישה הפוכה: "זהה את הכשל, ואז שחזר את הבאג שהוביל אליו."

החומרה מבצעת את החלק הקשה: זיהוי הגישה הבלתי חוקית לזיכרון. תפקידו של החוקר משתנה בהתאם , מזיהוי ראשוני של הכשל לניתוח שורש הבעיה (root cause analysis) ולהערכת השפעתה impact (assessment).

השלכות ומבט קדימה

עבור המגינים

MTE/MIE מסמן קפיצת מדרגה אמיתית באבטחת מכשירים. לראשונה, מנגנון חומרה מספק הגנה רציפה, שאינה דורשת כל תצורה מצד המשתמש, מפני שתיים ממחלקות פגיעויות ה-memory safety הנפוצות ביותר.

ארגונים המנהלים צי של מכשירים התומכים ב-MTE צריכים לשלב במערכי ניתוח ה-kernel panic שלהם זיהוי של אירועי Tag Check Fault. כל אירוע כזה הוא עדות לקיומו של באג אמיתי.

עבור התוקפים

MTE מעלה משמעותית את עלות הניצול של פגיעויות. כאשר ההסתברות להתנגשות תג (tag collision) היא 1 ל-16 בכל ניסיון, ההסתברות להצלחתן של שרשראות ניצול (exploit chains) מרובות שלבים יורדת באופן אקספוננציאלי. בשילוב עם בדיקה סינכרונית (synchronous checking) ו- Tag Confidentiality Enforcement, מרחב הפעולה של התוקף מצטמצם באופן משמעותי.

עם זאת, MTE אינו פתרון מושלם. תגים בני 4 ביטים מספקים אנטרופיה מוגבלת בלבד. באגים לוגיים, פגיעויות מסוג type confusion בתוך אותו zone ודליפות מידע שאינן כרוכות ב-memory corruption נמצאים מחוץ להיקף ההגנה של MTE. מצב המירוץ שתיארנו ממחיש זאת היטב. במכשיר התומך ב-MTE, הפגיעות מובילה באופן עקבי ל-kernel panic. לעומת זאת, בבסיס ההתקנות העצום של מכשירים שאינם תומכים ב-MTE, אותה פגיעות עצמה גם ניתנת לניצול וגם נותרת בלתי נראית.

עבור האקוסיסטם

המימושים של MTE ב-Apple וב-Google Pixel מייצגים שתי תפיסות שונות של אותה טכנולוגיה. הגישה של Apple מקיפה יותר, MIE פעיל תמיד, משתמש בבדיקה סינכרונית וכולל גם את EMTE ואת Tag Confidentiality Enforcement, אך זמינה רק בדור החומרה החדש ביותר. מנגד, Google פרסה את MTE מוקדם יותר ועל מגוון רחב יותר של מכשירים, אך בחרה במודל אכיפה שמרני יותר.

מנקודת מבט מחקרית, ההבדל משמעותי אף יותר. ב-Pixel, השימוש ב-MTE הוא opt-in עבור כל אפליקציה (או מופעל במסגרת Advanced Protection), והוא יכול לפעול גם במצב א־סינכרוני. במצב זה, חוסר התאמה בין תגים מוביל ל-SIGSEGV בכניסה הבאה לקרנל, מבלי שכתובת ההוראה שגרמה לכשל תישמר.

לעומת זאת, ב-Apple, מנגנון MIE פועל תמיד, באופן סינכרוני, הן בקרנל והן ביותר מ-70 תהליכי user space, כך שכל Tag Check Faultמקושר במדויק להוראת ה-load או ה-store שגרמה לו. בנוסף, Tag Confidentiality Enforcement חוסם גם מתקפות Side Channel ספקולטיביות, כגון TikTag, היכולות לחשוף תגים במימוש של Pixel.

השאלה האמיתית היא באיזו מהירות יאמצו מפתחי אפליקציות צד שלישי את MTE. הטכנולוגיה מסוגלת לחשוף פגיעויות memory safety רדומות בכל קוד C/C++, ולכן ייתכן שאפליקציות מסוימות יתחילו לקרוס לאחר הפעלתה, לא משום שנוצרו באגים חדשים, אלא משום שבאגים שהיו קיימים לאורך כל הדרך הפכו סוף סוף לגלויים. כל קריסה כזו היא באג שניתן לזהות ולתקן.

סיכום

MTE עושה הרבה יותר מאשר לצמצם את יכולת הניצול של פגיעויות. הוא משמש כמיקרוסקופ החושף מחלקה שלמה של באגים שלקהילת האבטחה פשוט לא היו הכלים לראות עד היום.

ככל שחומרה התומכת ב-MTE תהפוך לנפוצה יותר, אנו מצפים לעלייה זמנית במספר פגיעויות הקרנל שיתגלו. לא משום שהתוכנה הופכת לפחות בטוחה, אלא משום שלראשונה ניתן לראות את מה שהיה שם מאז ומתמיד.

המקרה שהצגנו, פגיעות use-after-free שנגרמה כתוצאה ממצב מירוץ בתת־מערכת קרנל עמוסת תעבורה, ונחשפה רק בזכות מנגנון ה-Tag Check הסינכרוני של MIE , הוא ככל הנראה רק דוגמה אחת מתוך רבות.

במובן מסוים, כל iPhone 17 שנמצא כיום בשטח מריץ באופן רציף fuzzer של memory safety ברמת החומרה כנגד קרנל iOS. הבאגים שהוא חושף היו קיימים לאורך כל הדרך. כעת, משהם ניתנים לזיהוי, ניתן גם לתקן אותם , והמערכת האקולוגית כולה יוצאת נשכרת.

עבור מי שמבלה את ימיו בניתוח לוגי kernel panic, MTE שינה בשקט את אופי העבודה. Tag Check Fault אינו רק אינדיקציה לכך שמנגנון ההגנה עבד כפי שתוכנן , הוא גם נקודת הפתיחה לחקירת הפגיעות הבאה.

על המחבר

ניר אברהם, VP Research, Jamf.

לינקדאין: https://www.linkedin.com/in/nir-avraham-95b96b36

Hu Ke, Security Researcher, Jamf

ברצוני להודות לחברי היקר Hu, שעזר לאורך הדרך ותרם רבות למחקרים אותם ביצענו לאורך השנים.

תורגם ונערך על ידי: IL4N10US