מבט מבפנים על המנוע הקרנלי של Predator

מאת ניר אברהם

תקציר מנהלים

זה הוא החלק השלישי ואחרון בסדרת המאמר על Predator, במחקר הקודם שלנו, תיעדנו כיצד תוכנת הריגול Predator מתחמקת מבדיקות אנטי-אנליזה ב-iOS ועוקפת את מחווני ההקלטה של iOS. המחקר חשף מה Predator עושה - אך לא כיצד היא משיגה את הגישה העמוקה למערכת שנדרשת כדי לבצע זאת. חלק זה עונה על השאלה הזו. מומלץ לקרוא את חלק א' וחלק ב' של סדרת מאמרים זו.

באמצעות המשך הנדסה לאחור של דגימות Predator ל-iOS, אנו מתבססים על הדוח הראשון שלנו וחושפים כיצד מנוע ניצול הקרנל של Predator מניע את יכולות המעקב שלה. מנוע זה מעולם לא דווח בעבר - עד עכשיו.

שרשרת הניצול שתוצג במאמר זה מכוונת לגרסאות iOS שקודמות ל-17 ולמכשירים עד דור A16. ההוספה של SPTM (Secure Page Table Monitor) על ידי Apple במכשירי A15, שמעבירה את ניהול טבלאות הדפים אל EL2, מהווה מנגנון הקשחה ארכיטקטוני משמעותי נגד טכניקות שינוי קוד הקרנל המתוארות כאן.

הממצאים המרכזיים כוללים:

- **FDGuardNeonRW** - פרימיטיב קריאה/כתיבה לקרנל, Kernel R/W, שמשתמש ברגיסטרים וקטוריים של ARM NEON כערוץ נתונים סמוי לצורך קריאה וכתיבה של זיכרון קרנל שרירותי.
- **עקיפת PAC באמצעות חיפוש gadgets ב-JavaScriptCore** - Predator מחפשת במסגרת JavaScriptCore של Apple עצמה רצף ספציפי של 20 בתים של פקודות ARM64, כדי לזייף מצביעים חתומים ב-PAC.
- **Cache חתימות PAC בן 256 רשומות** - מצביעים חתומים שחושבו מראש ומאונדקסים לפי בית הכתובת, המאפשרים callbacks של hooks בזמן אמת ללא השהיה קריפטוגרפית.
- **RWTransfer** - מנגנון להעברת יכולות קריאה/כתיבה לקרנל בין תהליכים, תוך שימוש ב-file descriptors מוגנים ובמניפולציה של Mach ports.

- **callFunc** - מסגרת להרצת פונקציות מרוחקות, אשר משתלטת על מצב thread באמצעות הודעות Mach exception.
- **21 דגמי מכשירים נתמכים**, החל מ-iPhone XS ועד iPhone 14 Pro Max, המאורגנים ב-5 מחלקות מכשירים.

רקע: מאפייני האבטחה ש-Predator חייבת להביס

(PAC) Pointer Authentication Codes

החל משבב A12 (iPhone XS, 2018), Apple הציגה את PAC - מאפיין חומרה שמוסיף חתימות קריפטוגרפיות למצביעי קוד. לפני שהמעבד עוקב אחר מצביע לפונקציה או אחר כתובת חזרה, הוא מאמת את החתימה. אם תוקף משחית מצביע, בדיקת החתימה נכשלת והמעבד מעלה חריגה.

(KASLR) Kernel Address Space Layout Randomization

iOS מבצעת רנדומיזציה לכתובת הבסיס של הקרנל בכל אתחול, באמצעות KASLR. המשמעות היא שכל קוד הקרנל וכל מבני הנתונים הסטטיים מוזזים בהיסט בלתי צפוי. גם אם לתוקף יש חולשת קרנל, עליו קודם לקבוע את ה-slide הזה כדי לאתר את המבנים שהוא רוצה לבצע בהם מניפולציה. לשם כך נדרש פרימיטיב אמין לקריאת זיכרון קרנל - וזה בדיוק מה ש-FDGuardNeonRW מספק.

ממצא 1: FDGuardNeonRW - גישה לזיכרון הקרנל דרך רגיסטרים וקטוריים

פענוח השם

שם המחלקה **FDGuardNeonRW** מקודד את כל הטכניקה שלה:

Component	Meaning
FD	File Descriptor – the exploit involves the `change_fdguard_np` syscall
Guard	FD guard manipulation to set up the kernel primitive
Neon	ARM NEON vector registers serve as the data channel
RW	Provides both read and write access to kernel memory

זהו פרימיטיב הליבה שמניע את כל יתר הרכיבים ב-Predator. כל hook, כל הזרקה לתהליך, וכל עקיפת PAC תלויים בסופו של דבר ביכולת של FDGuardNeonRW לקרוא ולכתוב זיכרון קרנל שרירותי.

ערוץ הנתונים של NEON

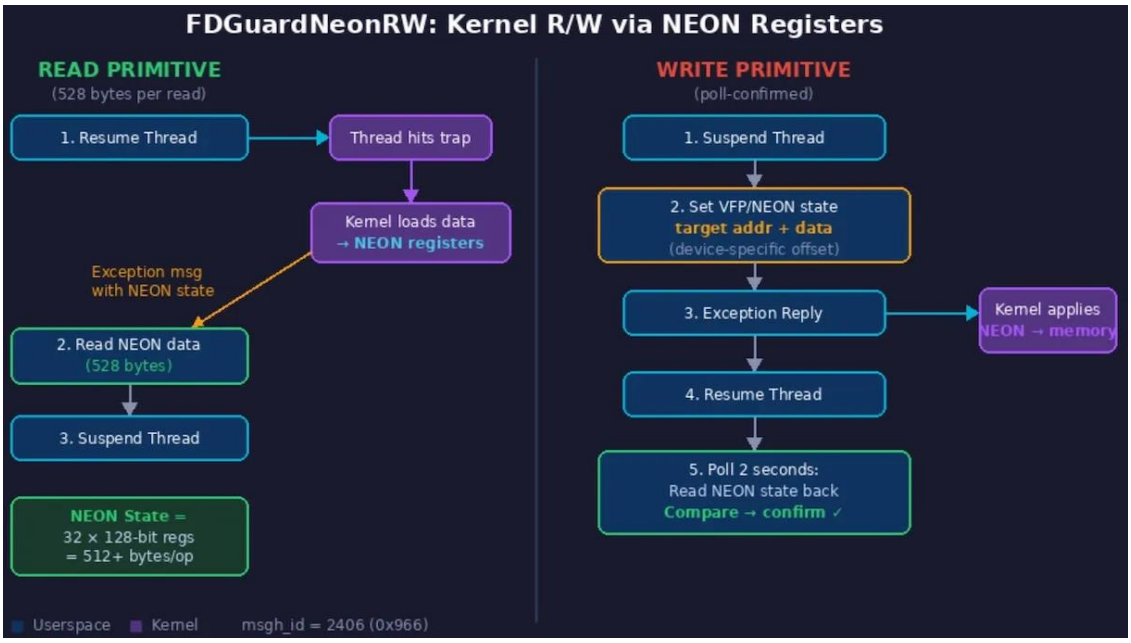
ARM NEON הוא סט של רגיסטרים וקטוריים ברוחב 128 ביט, V0-V31, המשמשים בדרך כלל לפעולות SIMD - חישוב מקבילי על מספר ערכים בו-זמנית. רגיסטרים אלה הם חלק ממצב ה-thread שהקרנל שומר ומשחזר במהלך context switches.

Predator מנצלת זאת באמצעות טכניקה מוכרת היטב: ה-APIs של Mach בשם thread_get_state ו-thread_set_state מאפשרים לקרוא ולכתוב את מצב הרגיסטרים המלא של thread, כולל רגיסטרי NEON, מ-thread אחר שיש לו הרשאות port מתאימות. גישת מניפולציית מצב ה-thread הזו שימשה בשרשראות ניצול של iOS לפחות מאז 2017.

החידוש של Predator הוא ניתוב נתוני קרנל דרך בנק רגיסטרי NEON באופן ספציפי - 32 רגיסטרים וקטוריים בגודל 128 ביט מספקים 528 בתים בכל קריאת thread_get_state, כולל FPSR, FPCR וריפוד יישור, לעומת 272 בתים בלבד הזמינים דרך מצב הרגיסטרים הכלליים.

הערוץ הזה תלוי ברכיב בצד הקרנל שהוקם על ידי שלב הניצול הראשוני - רגיסטרי NEON משמשים כאמצעי ההעברה, אך קריאות וכתיבות זיכרון הקרנל בפועל מבוצעות על ידי קוד שרץ בהרשאות קרנל. הפרטים המלאים של payload זה בצד הקרנל נמצאים מחוץ לתחום הבלוג הזה.

תרשים זרימה המציג את פרימיטיבי הקריאה והכתיבה:



[איור 1: פרימיטיבי הקריאה והכתיבה של FDGuardNeonRW, המשתמשים ברגיסטרי NEON כערוץ נתונים לקרנל ברוחב פס גבוה]

פרימיטיב קריאה - 528 בתים בכל קריאה

פעולת הקריאה מסתמכת על נתיב קוד בקרנל שהותקן על ידי שלב הניצול הראשוני. נתיב קוד זה טוען נתונים מכתובת קרנל נשלטת אל רגיסטרי NEON, ולאחר מכן מגיע לפקודת trap. ה-thread היעד מחודש, מריץ את נתיב הקוד הזה, ומפעיל את ה-trap. מטפל החריגות של Predator מקבל הודעת Mach exception, המוצגת באיור 3 ומזוהה על ידי msgh_id 2406 / 0x966, המכילה את מצב ה-NEON עם נתוני הקרנל. עד 10 ניסיונות חוזרים מטפלים במרוצי תזמון. כל קריאה מחזירה 528 בתים.

פרימיטיב כתיבה - מאומת באמצעות polling

פעולת הכתיבה הופכת את הערוץ. Predator משהה את ה-thread היעד, קוראת את מצב ה-VFP/NEON הנוכחי כקו בסיס, משנה אותו עם כתובת היעד והנתונים במיקום תלוי-מכשיר, ולאחר מכן שולחת תשובת exception עם המצב ששונה. הקרנל משחזר את מצב ה-NEON ששונה אל תוך ה-saved context של ה-thread. כאשר ה-thread ממשיך לרוץ, נתיב הקוד של הניצול בקרנל קורא את הערכים מרגיסטרי NEON וכותב אותם לכתובת הקרנל היעד.

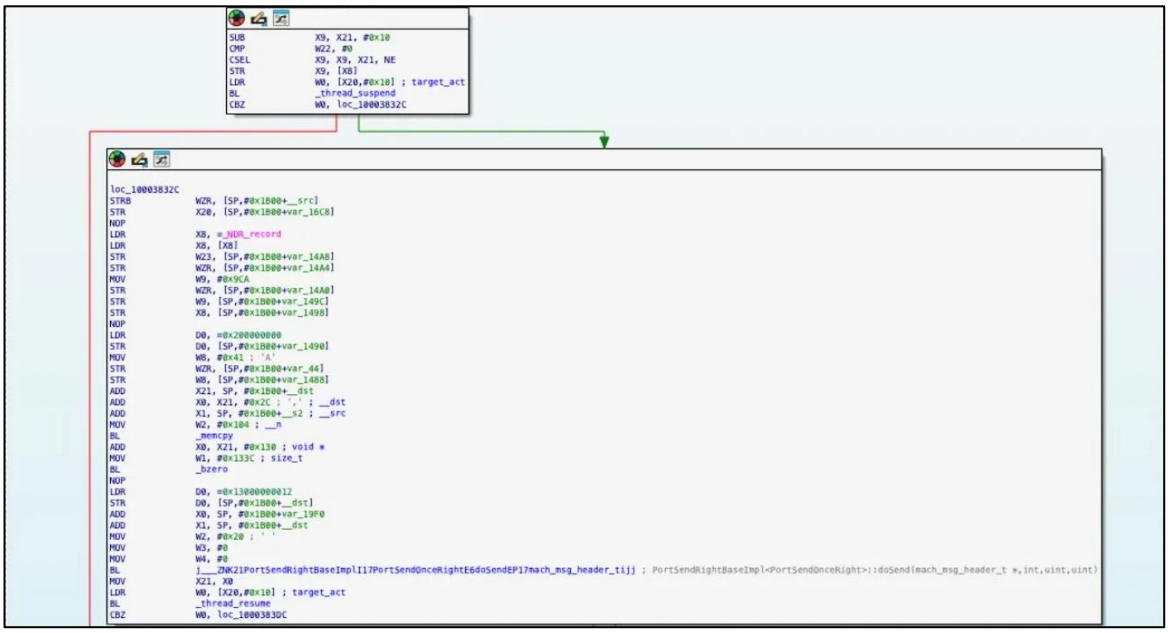
Predator מבצעת polling עד 3 שניות, ומשווה את מצב ה-NEON שנקרא בחזרה מול הנתונים שנשלחו כדי לוודא שהכתיבה הצליחה.

למה רגיסטרי NEON?

מצב NEON הוא:

- **בעל קיבולת גדולה** - יותר מ-512 בתים בכל פעולה.
- **מנוהל על ידי הקרנל** - חלק מהטיפול הרגיל במצב thread.
- **גלוי דרך exceptions** - נכלל בהודעות Mach exception.
- **דו-כיווני** - אותו מנגנון משמש לקריאות ולכתיבות.

פרימיטיב הכתיבה נראה כך:



[איור 2: תצוגת graph של IDA עבור פרימיטיב הכתיבה של NEON - thread_suspend, תשובת exception עם NDR_record, doSend, thread_resume, ולולאת polling של 2 שניות]

```
63    v11 = 10;
64    memset(__src, 0, sizeof(__src));
65    do
66    {
67      __asm { SVC             0 }
68      if ( *v5 )
69      {
70        usleep(0);
71        usleep(0x32u);
72        usleep(0x32u);
73        usleep(0x32u);
74        usleep(0x32u);
75      }
76      v16 = *v3;
77      bzero(v25, 0x1474u);
78      *(_QWORD *)&msg.msgh_bits = v27;
79      msg.msgh_remote_port = v28;
80      msg.msgh_local_port = v16;
81      bzero(&msg.msgh_voucher_port, 0x1478u);
82      msg.msgh_size = 5256;
83      mach_msg(&msg, 258, 0, 0x1488u, v16, 0x3E8u, 0);
84      v25[0] = 0;
85      p_msg = &msg;
86      msgh_remote_port = msg.msgh_remote_port;
87      if ( (msg.msgh_remote_port + 1 > 1 || msg.msgh_local_port - 1 <= 0xFFFFFFFD) && msg.msgh_id == 2406 && v23 == 65 )
88      {
89        memcpy(__dst, v24, sizeof(__dst));
90        memcpy(__src, __dst, sizeof(__src));
91        if ( msg.msgh_remote_port - 1 < 0xFFFFFFFE )
92          break;
93      }
94      else
95      {
96        usleep(0);
97        v29[0] = msg.msgh_remote_port;
98        sub_100009EC8(v29);
99        msgh_remote_port = 0;
100       memset(__src, 0, sizeof(__src));
101     }
102     ThreadPortBase<ThreadPortWeak>::getNeonState(&msg, v10);
103     usleep(0);
104     usleep(0x32u);
105     __asm { SVC             0 }
106     if ( *v5 )
107     {
108       usleep(0);
109       usleep(0x32u);
110       usleep(0x32u);
111       usleep(0x32u);
112       usleep(0x32u);
113     }
114     --v11;
115   }
```

[איור 3: פרימיטיב קריאת NEON לאחר decompilation - לולאת הניסיונות החוזרים (v11 = 10), קבלת mach_msg, אימות של msgh_id == 2406 ושל ספירת מצב VFP השווה ל-65, ולבסוף thread_suspend]

ממצא 2: ציד PAC gadgets ב-JavaScriptCore

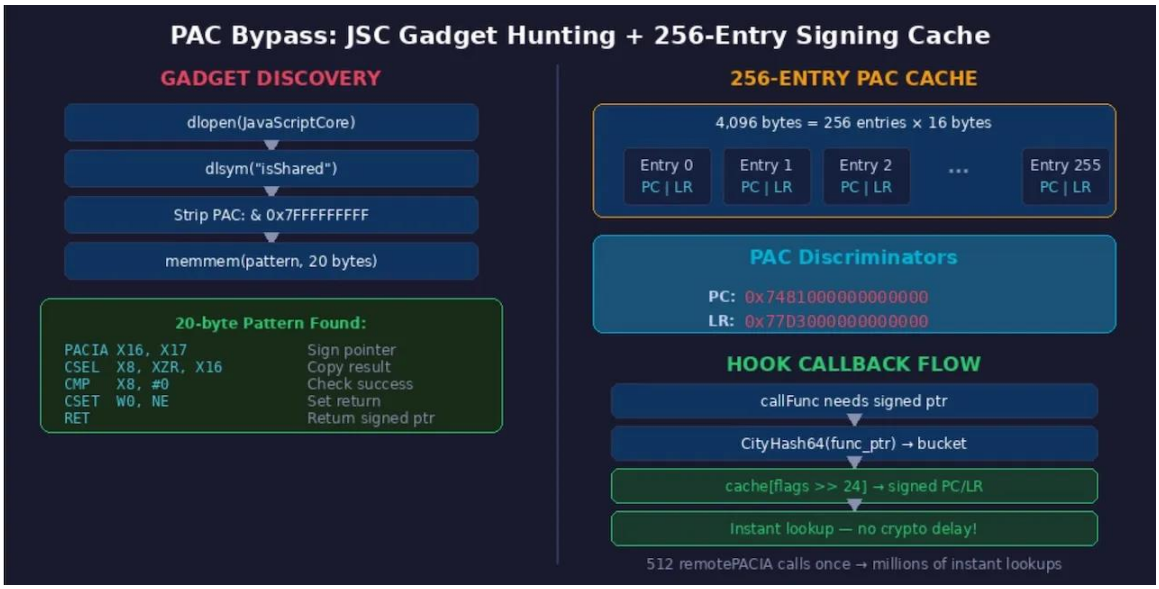
הבעיה

כדי להתקין hooks בתהליכי מערכת, Predator צריכה להסיט את הרצת הפונקציות - כלומר לשנות מצביעי קוד. במכשירים שבהם PAC מופעל, כלומר iPhone XS ומעלה, כל מצביע קוד נושא חתימה קריפטוגרפית. Predator חייבת לזייף חתימות תקפות עבור המצביעים שאליהם היא מנתבת מחדש.

הפתרון: השאלת הקוד של Apple עצמה

במקום להביא מימוש חתימה משלה, Predator מחפשת במסגרת JavaScriptCore של Apple רצף קוד קיים שמבצע חתימת PAC עם קלטים שניתנים לשליטה.

פונקציית היעד היא JSC::JSArrayBuffer::isShared() - פונקציה שקיימת בכל מכשיר iOS כחלק ממנוע ה-JavaScript של Safari:



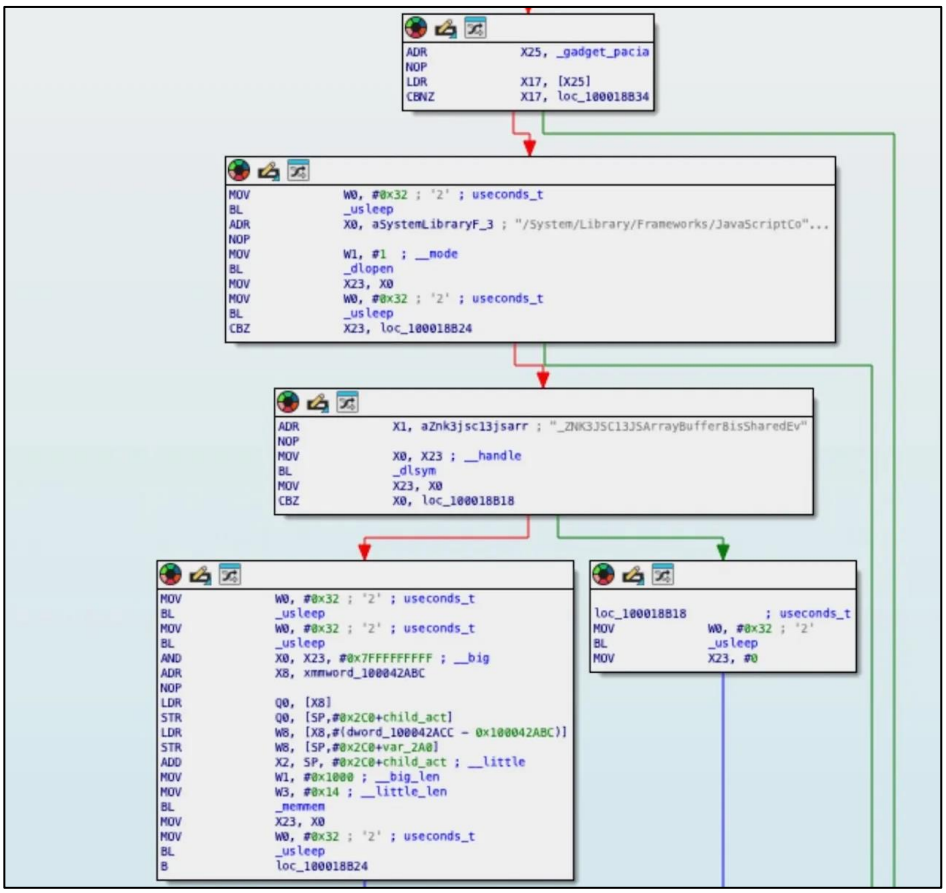
[איור 4: עקיפת PAC באמצעות ציד gadgets ב-JSC ו-cache חתימות שחושב מראש בן 256 רשומות]

ה-gadget בן 20 הבתים

Predator מחפשת בתוך 0x1000 בתים מהסימבול isShared באמצעות memmem() אחר התבנית המדויקת הבאה באורך 20 בתים:

Bytes	ARM64 instruction	Purpose
30 02 C1 DA	PACIA X16, X17	Sign pointer in X16 using X17 as context
E8 03 90 9A	CSEL X8, XZR, X16, EQ	Copy result if valid
1F 01 00 F1	CMP X8, #0	Check if signing succeeded
E0 07 9F 1A	CSET W0, NE	Set return value
C0 03 5F D6	RET	Return signed pointer

פקודת PACIA X16, X17 היא המפתח: היא חותמת את המצביע שנמצא ב-X16 באמצעות X17 כהקשר או discriminator, תוך שימוש במפתח PAC של החומרה. באמצעות שליטה ב-X16 וב-X17, Predator יכולה לזייף חתימות עבור מצביעים שרירותיים:



[איור 5: תצוגת graph של IDA עבור remotePACIA - זרימת ציד ה-gadget: בדיקת cache gadget_pacia ← dlopen(JavaScriptCore) ← dlsym(isShared) ← memmem(pattern, 0x14)]

```
132     v18 = gadget_pacia;
133     if ( !gadget_pacia )
134     {
135       usleep(0x32u);
136       v19 = dlopen("/System/Library/Frameworks/JavaScriptCore.framework/JavaScriptCore", 1);
137       usleep(0x32u);
138       if ( v19 )
139       {
140         v20 = (unsigned __int64)dlsym(v19, "_ZNK3JSC13JSArrayBuffer8isSharedEv");
141         usleep(0x32u);
142         if ( v20 )
143         {
144           usleep(0x32u);
145           *(_QWORD *)child_act = xmmword_100042ABC;
146           v34 = -698416192;
147           v19 = memmem((const void *)(v20 & 0x7FFFFFFFFFLL), 0x1000u, child_act, 0x14u);
148           usleep(0x32u);
149         }
150         else
151         {
152           v19 = 0;
153         }
154       }
155       gadget_pacia = (__int64)v19;
```

[איור 6: חיפוש ה-gadget לאחר decompilation - dlopen, dlsym עבור _ZNK3JSC13JSArrayBuffer8isSharedEv, והתאמת תבנית באמצעות memmem עם תבנית ה-gadget PACIA באורך 20 בתים, המאוחסנת ב-xmmword_100042ABC]

```
0000000100042AA0  45 6E 61 62 6C 65 72 39  73 65 74 75 70 48 6F 6F  Enabler9setupHoo
0000000100042AB0  6B 45 76 45 33 24 5F 31  00 00 00 00 30 02 C1 DA  kEvE3$_1....0...
0000000100042AC0  E8 03 90 9A 1F 01 00 F1  E0 07 9F 1A C0 03 5F D6  ..............._.
0000000100042AD0  4E 53 74 33 5F 5F 31 31  30 5F 5F 66 75 6E 63 74  NSt3__110__funct
```

[איור 7: תצוגת hex בכתובת 0x100042ABC, המציגה את תבנית ה-gadget בת 20 הבתים: PACIA X16, X17 (30 02 C1 DA) מודגשת באזור נתוני הקבועים של הבינארי]

ממצא 3: cache חתימות PAC בן 256 רשומות

כל קריאה ל-remotePACIA כוללת יצירת thread, הגדרת exception ports, שינוי מצב ה-thread בקרנל, הרצת ה-gadget וקבלת התוצאה. תהליך זה אורך מילישניות - הרבה יותר מדי זמן עבור callbacks של hooks שחייבים להסתיים בתוך מיקרו-שניות.

טבלת חתימות שחושבה מראש

Predator בונה בעת האתחול cache של 256 מצביעים חתומים מראש, המכסה כל ערך אפשרי של הבית העליון בכתובת. כאשר hook מופעל, ניתן לשלוף מיידית את המצביע החתום הנכון:

Parameter	Value	Notes
Cache size	4,096 bytes	256 entries × 16 bytes per entry
Entry format	[signed_PC, signed_LR]	8 bytes each
PC discriminator	`0x7481000000000000`	Program Counter signing
LR discriminator	`0x77D3000000000000`	Link Register signing
Index	cache[16 × (flags >> 24)]	Top byte of CPSR flags
Hash function	CityHash64 (Google)	For hashmap bucket lookup

שימוש ב-discriminators נפרדים עבור PC ועבור LR פירושו שכתובת חזרה חתומה אינה יכולה לשמש כתחליף ליעד קפיצה חתום - מנגנון defense-in-depth מצד Apple ש-Predator חייבת לקחת בחשבון. קריאות remotePACIA היקרות, 256 × 2 = 512 קריאות לכל cache, מתבצעות פעם אחת בלבד עבור כל יעד פונקציה ייחודי. כל קריאות ה-hook הבאות הן שליפות מיידיות מה-cache.

ממצא 4: callFunc - הרצת פונקציות מרוחקות

callFunc הוא המנגנון העיקרי של Predator להרצת פונקציות שרירותיות בתהליכים מרוחקים. הוא משלב את ה-PAC cache, Mach exceptions ומניפולציה של מצב ה-thread לכדי פרימיטיב אחד שניתן לקרוא לו, המקבל מצביע לפונקציה ועד 6 ארגומנטים, x0-x5.

המנגנון

"trojan thread" (מונח פנימי של Predator), יושב בתהליך המרוחק על breakpoint ומייצר Mach exceptions באופן רציף. callFunc מקבל את ה-exception, ממלא את רגיסטרי הארגומנטים, ושולף מ-cache ה-PAC את ערכי ה-PC וה-LR החתומים הנכונים. תשובת ה-exception מכילה את מצב ה-thread ששונה - וכאשר הקרנל מוסר את התשובה, ה-trojan thread ממשיך לרוץ בכתובת פונקציית היעד עם הארגומנטים שצוינו.

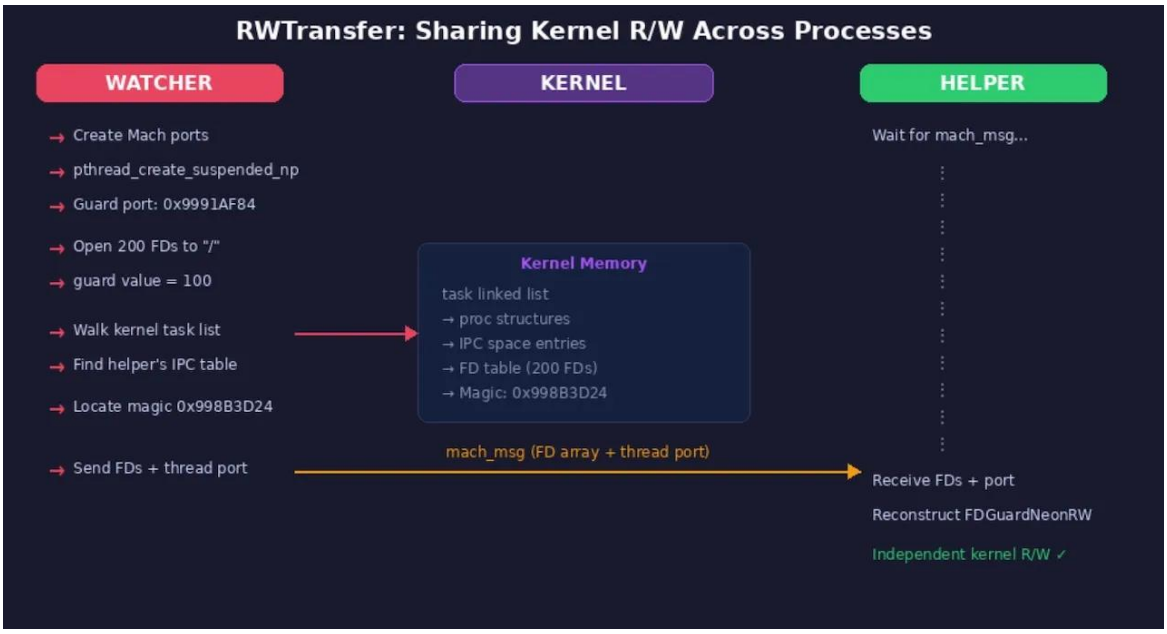
כתובת החזרה מסוג "poison LR" מצביעה בחזרה ל-breakpoint נוסף, כך שכאשר הפונקציה חוזרת, נוצר exception חדש ו-Predator מקבלת שוב שליטה. כך נוצר מנגנון reusable אינסופי לקריאה מרוחקת לפונקציות, כלומר מעין Remote Procedure Call. מקור התייחסות: הפונקציה בכתובת 0x10000B028 מתעדת בלוג את מקור הקובץ שלה כ-"TaskROPDevOff.h"- מה שמאשר שמדובר בטכניקה מבוססת ROP שתוכננה עבור מכשירים שבהם Developer Mode אינו מופעל:

```
LABEL_33:
    bzero(v64, 0x1000u);
    Log::LogManager::Init(v40);
    *(_QWORD *)&msg[0].msgh_bits = "callFunc: building PAC cache for ";
    *(_QWORD *)&msg[0].msgh_remote_port = 34;
    *(_QWORD *)v53 = "...";
    v54[0] = 4;
    sub_10000BA4C(0, "TaskROPDevOff.h", "", 451, msg, &v62, v53);
    v41 = (Log::LogManager *)NSTaskROP::WithoutDeveloperMode::TaskROP<FDGuardNeonRW>::signState(
                               (int)v16,
                               (int)&__dst,
                               v15,
                               1024,
                               0,
                               v64);
    if ( ((unsigned __int8)v41 & 1) == 0 )
    {
      Log::LogManager::Init(v41);
      *(_QWORD *)&msg[0].msgh_bits = "Statement signState(&data, key, sPoisonLRForTrojan, NULL, &cache) failed!";
      *(_QWORD *)&msg[0].msgh_remote_port = 74;
      v32 = msg;
      v33 = 452;
      goto LABEL_40;
    }
    *(_QWORD *)&msg[0].msgh_bits = v15;
    memcpy(&msg[0].msgh_remote_port, v64, 0x1000u);
    sub_10000AB70(&v16[75], msg, msg);
    goto LABEL_35;
  }
  if ( v30.u32[0] < 2uLL )
  {
    v37 = *(_QWORD *)&v29 - 1LL;
    while ( 1 )
    {
      v39 = v36[1];
      if ( v39 == v15 )
      {
        if ( v36[2] == v15 )
          goto LABEL_38;
      }
      else if ( (v39 & v37) != v31 )
      {
        goto LABEL_33;
      }
      v36 = (_QWORD *)*v36;
      if ( !v36 )
        goto LABEL_33;
    }
  }
  while ( 1 )
  {
    v38 = v36[1];
    if ( v38 == v15 )
      break;
    if ( v38 >= *(_QWORD *)&v29 )
      v38 %= *(_QWORD *)&v29;
    if ( v38 != v31 )
      goto LABEL_33;
  }
LABEL_23:
    v36 = (_QWORD *)*v36;
    if ( !v36 )
      goto LABEL_33;
  }
  if ( v36[2] != v15 )
    goto LABEL_23;
LABEL_38:
  memcpy(msg, v36 + 3, 0x1000u);
  Flags = Arm64State::getFlags((Arm64State *)&v50);
  v45 = (char *)msg + 16 * HIBYTE(Flags);
  v46 = *(void **)v45;
  v47 = (void *)*((_QWORD *)v45 + 1);
  Arm64State::setFlags((Arm64State *)&v50, Flags & 0xFFFFFFF1);
  Arm64State::setPC((Arm64State *)&v50, v46);
  Arm64State::setLR((Arm64State *)&v50, v47);
  v42 = (Log::LogManager *)ExceptionMessage<6,7166u,2147483650ull>::Send(&v61, &__dst);
  if ( ((unsigned __int8)v42 & 1) != 0 )
    return 1;
```

[איור 8: callFunc לאחר decompilation - במקרה של cache miss ב cache ה-PAC, הפונקציה מפעילה את signState כדי לבנות cache בן 256 רשומות (LABEL_33). במקרה של cache hit (LABEL_38), מתבצעת שליפה מיידית באמצעות getFlags << 24, ולאחר מכן הגדרת PC ו-LR באמצעות setPC/setLR עם מצביעים חתומים, לפני הקריאה ל-ExceptionMessage::Send]

ממצא 5: RWTransfer - שיתוף פרימיטיבי קרנל בין תהליכים

הארכיטקטורה של Predator מפצלת יכולות בין מספר תהליכים: "watcher" מנהל את מחזור החיים, בעוד שתהליכי "helper" מבצעים את המעקב בפועל. ה-watcher משיג Kernel R/W באמצעות הניצול הראשוני, אך גם ה-helpers זקוקים לכך. RWTransfer פותר את הבעיה הזו:



[איור 9: פרוטוקול RWTransfer לשיתוף יכולות Kernel R/W בין תהליכים]

Component	Value	Purpose
Transfer magic	`0x998B3D24`	Identifies RW transfer structure in kernel memory
Port guard	`0x9991AF84`	Applied to Mach receive port
File descriptors	200 FDs to "/"	Created via `change_fdguard_np` (guard=100)
Thread creation	`pthread_create_suspended_np`	Dedicated thread for transfer
Delivery	`mach_msg`	FD array + thread port sent to helper

ה-watcher עובר על רשימות מקושרות בקרנל, עם offsets ייחודיים למחלקת המכשיר, כדי למצוא את מבנה ה-task של ה-helper, לאחר מכן את מרחב ה-IPC שלו, ולאחר מכן רשומות port בודדות. זהו אחד הרכיבים המורכבים ביותר ב-Predator - והוא כולל מניפולציה סימולטנית של Mach ports, file descriptors, רשימות מקושרות בקרנל ומצבי thread.

ממצא 6: פתרון מרוחק של מתודות Objective-C

כאשר Predator צריכה לבצע hook למתודת Objective-C בתהליך מרוחק, היא לא תמיד יכולה להסתמך על שאילתות runtime מקומיות. בעוד שמתודות שממומשות בתוך ה-dyld shared cache ממופות לאותן כתובות בכל התהליכים באותו boot, מתודות בבינאריים ספציפיים לאפליקציה כפופות ל-ASLR slides נפרדים לכל תהליך. לכן Predator מריצה את שרשרת פתרון ה-Objective-C runtime המלאה מרחוק באמצעות callFunc, וכך מבטיחה תוצאות נכונות ללא תלות במקום שבו מתודת היעד ממומשת:

Stage	Runtime function	Purpose
1	`sel_registerName(methodName)`	Convert method name to selector
2	`objc_getClass(className)`	Get class object
3	`class_getInstanceMethod(cls, sel)`	Get Method structure
4	`method_getImplementation(method)`	Get implementation pointer

כל קריאת callFunc עוברת דרך כל ה-pipeline: Mach exception ← cache PAC ← מניפולציה של מצב thread. ביצוע כל ארבעת השלבים מרחוק מבטיח פתרון נכון גם עבור מתודות שמחוץ ל-shared cache, שבהן הכתובות המקומיות והמרוחקות יהיו שונות עקב ASLR slides עצמאיים.

ממצא 7: מטריצת תמיכה במכשירים - 21 דגמים ב-5 מחלקות

פריסות מבני הקרנל משתנות בין דגמי iPhone עקב הבדלים בדור ה-SoC, במאפייני אבטחה ובו- iOS kernel builds. Predator מחזיקה קונפיגורציות לפי מחלקת מכשיר, עם offsets מדויקים בקרנל.

Class	SoC	iPhone models	Count
0	A12 Bionic	XS, XS Max, XS Max (China), XR	4
1	A13 Bionic	11, 11 Pro, 11 Pro Max, SE (2nd gen)	4
3	A14 Bionic	12 mini, 12, 12 Pro, 12 Pro Max	4
4	A15/A16	13 mini, 13, 13 Pro, 13 Pro Max, SE (3rd), 14, 14 Plus, 14 Pro, 14 Pro Max	9

הערה: מחלקת מכשיר 2 אינה נמצאת בשימוש בדגימה זו - ככל הנראה היא שמורה לגרסת חומרה מסוימת או אוחדה עם מחלקה אחרת. מכשירים שאינם נתמכים גורמים לפונקציה להחזיר 5, ובכך להפסיק את ההרצה במקום להסתכן ב-kernel panic. בסך הכול: 21 דגמי iPhone המשתרעים על חומרה מהשנים 2018-2022.

פענוח השוואות ה-XOR

הקוד שעבר decompilation, המוצג באיור 10, נראה כאילו הוא מכיל קבועים הקסדצימליים אטומים, אך למעשה מדובר פשוט במחרוזות ASCII המקודדות ב-XOR. הפונקציה קוראת ל-uname() כדי לקבל את מזהה המכונה של המכשיר, לדוגמה iPhone15,3, ולאחר מכן מבצעת השוואות XOR - אם התוצאה שווה לאפס, המכשיר תואם. הקידוד ישיר: 0x3531 הוא ASCII עבור "15" ו-0x332C הוא ASCII עבור ",3", שניהם בסדר בתים little-endian. כל תנאי בפלט ה-decompiled ממופה ישירות למזהה דגם iPhone ספציפי.

לדוגמה, התנאי הראשון בבלוק שמחזיר 4:

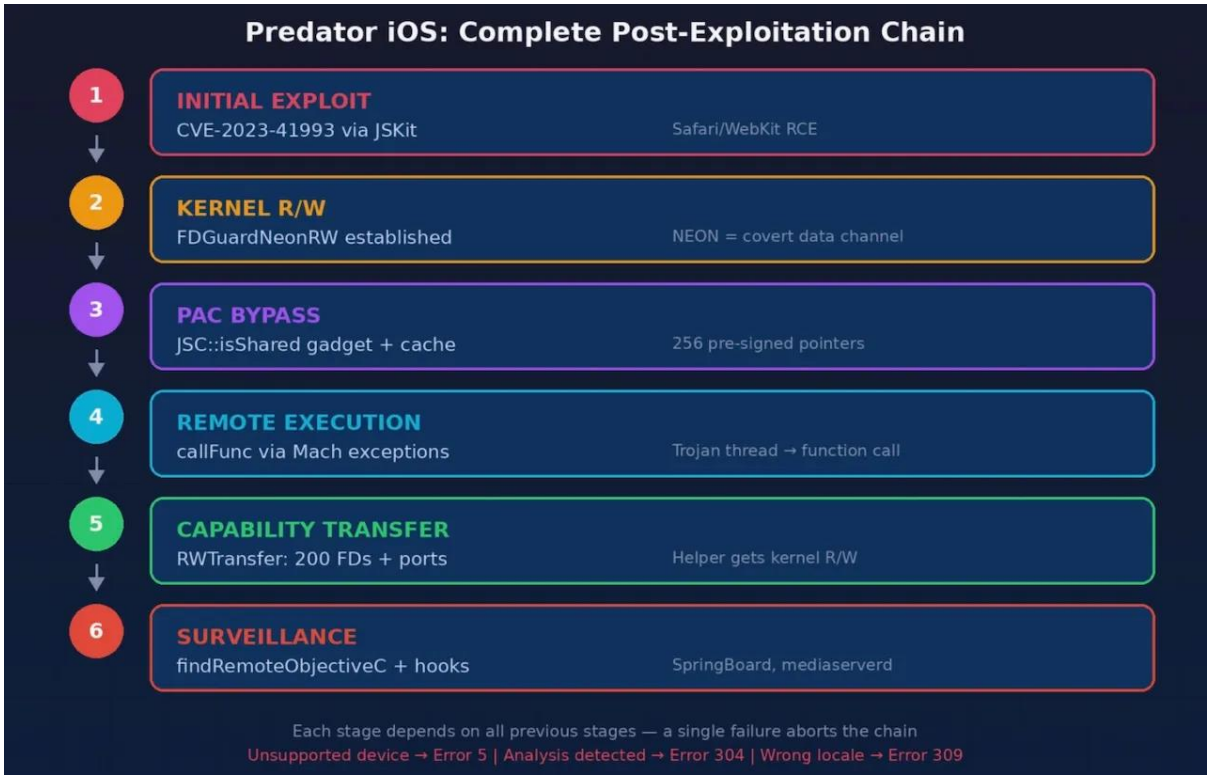
```
qword_1000491A0 ^ 0x3531656E6F685069 → "iPhoneXX"
(char *)&qword_1000491A0 + 3) ^ 0x332C3531656E6F → "oneXX,Y"
```

כאשר שתי תוצאות ה-XOR הן אפס, מחרוזת המכשיר תואמת ל-"iPhone15,3" - כלומר iPhone 14 Pro Max:

```
1  __int64 __fastcall VersionDispatcher::OffsetsVersionByDeviceInit(VersionDispatcher *this)
2  {
3    if ( !(_BYTE)qword_1000491A0 && uname((utsname *)&VersionDispatcher::m_uname) == -1 )
4    {
5      usleep(0);
6      usleep(0x32u);
7      usleep(0x32u);
8  LABEL_27:
9      usleep(0);
10     return 5;
11   }
12   if ( !(qword_1000491A0 ^ 0x3531656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x332C3531656E6FLL)
13     || !(qword_1000491A0 ^ 0x3531656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x322C3531656E6FLL)
14     || !(qword_1000491A0 ^ 0x3431656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x382C3431656E6FLL)
15     || !(qword_1000491A0 ^ 0x3431656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x372C3431656E6FLL)
16     || !(qword_1000491A0 ^ 0x3431656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x362C3431656E6FLL)
17     || !(qword_1000491A0 ^ 0x3431656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x352C3431656E6FLL)
18     || !(qword_1000491A0 ^ 0x3431656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x342C3431656E6FLL)
19     || !(qword_1000491A0 ^ 0x3431656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x332C3431656E6FLL)
20     || !(qword_1000491A0 ^ 0x3431656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x322C3431656E6FLL) )
21   {
22     return 4;
23   }
24   if ( !(qword_1000491A0 ^ 0x3331656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x342C3331656E6FLL)
25     || !(qword_1000491A0 ^ 0x3331656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x332C3331656E6FLL)
26     || !(qword_1000491A0 ^ 0x3331656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x322C3331656E6FLL)
27     || !(qword_1000491A0 ^ 0x3331656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x312C3331656E6FLL) )
28   {
29     return 3;
30   }
31   if ( qword_1000491A0 ^ 0x3231656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x382C3231656E6FLL
32     && qword_1000491A0 ^ 0x3231656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x352C3231656E6FLL
33     && qword_1000491A0 ^ 0x3231656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x332C3231656E6FLL
34     && qword_1000491A0 ^ 0x3231656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x312C3231656E6FLL )
35   {
36     if ( !(qword_1000491A0 ^ 0x3131656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x382C3131656E6FLL)
37       || !(qword_1000491A0 ^ 0x3131656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x362C3131656E6FLL)
38       || !(qword_1000491A0 ^ 0x3131656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x342C3131656E6FLL)
39       || !(qword_1000491A0 ^ 0x3131656E6F685069LL | *(__int64 *)((char *)&qword_1000491A0 + 3) ^ 0x322C3131656E6FLL) )
40     {
41       return 0;
42     }
43     goto LABEL_27;
44   }
45   return 1;
46 }
```

[איור 10: VersionDispatcher::OffsetsVersionByDeviceInit לאחר decompilation - זיהוי מכשיר באמצעות uname() עם השוואת מחרוזות מבוססת XOR, המחזירה מחלקות מכשיר 0-4, ומחזירה 5 עבור דגמים שאינם נתמכים כדי להפסיק את ההרצה באופן בטוח]

שרשרת התקיפה המלאה



[איור 11: שרשרת post-exploitation מלאה, מהניצול הראשוני ועד מעקב פעיל]

כל שלב תלוי בכל השלבים שקדמו לו - כשל יחיד בכל נקודה בשרשרת מבטל את השרשרת כולה. בשילוב עם מערכת קודי השגיאה שתועדה במאמר הראשון שלנו, הדבר מעניק למפעילי Predator מידע דיאגנוסטי מדויק לגבי איזה שלב נכשל ומדוע.

סיכום

FDGuardNeonRW מדגים כי רגיסטרים וקטוריים של ARM NEON, שנועדו לחישוב מקבילי, יכולים להיות מוסבים לשמש כערוץ סמוי לגישה לזיכרון הקרנל. פרימיטיבי הקריאה של 528 בתים והכתיבה המאומתת באמצעות polling אמינים מספיק כדי לתמוך בכל מסגרת ה-post-exploitation של Predator.

עקיפת ה-PAC ממחישה כי מאפייני אבטחה חומרתיים, אף שהם מעלים משמעותית את הרף, ניתנים לעקיפה כאשר לתוקף יש גישת קריאה/כתיבה לקרנל. באמצעות ציד gadgets במסגרות של Apple עצמה וחישוב מראש של cache חתימות, Predator מביסה את pointer authentication עם תקורה זניחה בזמן ריצה.

מנגנון העברת היכולות חושף את רמת התחכום ההנדסי של תוכנות ריגול מסחריות. העברת גישת Kernel R/W בין תהליכים - הכוללת מניפולציה סימולטנית של Mach ports, file descriptors, רשימות מקושרות בקרנל ומצבי thread - משקפת השקעה הנדסית מתמשכת ומקצועית.

ממצאים אלה מדגישים מציאות מפוכחת: ספקי תוכנות ריגול מסחריות משקיעים רבות בהנדסת post-exploitation, ולא רק בגילוי חולשות ראשוניות. הגנה מפני יכולות כאלה דורשת אמצעי אבטחה שפועלים מתחת לרמה שהכלים הללו מצליחים לפגוע בה - attestation שמושרש בחומרה, זיכרון קרנל חתום או אטום, וניטור out-of-band שאינו מסתמך על שלמות מחסנית התוכנה של המכשיר עצמו.

מקורות מידע

1. Jamf Threat Labs, "Predator's Kill Switch: Undocumented Anti-Analysis Techniques in iOS Spyware," January 2026

2. Jamf Threat Labs, "How Predator Spyware Defeats iOS Recording Indicators," February 2026

3. Google Threat Intelligence Group, "Sanctioned but Still Spying: Intellexa's Prolific Zero-Day Exploits Continue," December 2025

4. Apple, Apple Platform Security Guide

לקריאה נוספת

Jamf Threat Labs - Research blog and additional publications - https://www.jamf.com/blog/category/jamf-threat-labs/

על המחבר

ניר אברהם, VP Research, Jamf. לינקדאין:

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

תורגם ונערך על ידי: IL4N10US