digital whisper
Technologic Papers

GhostHook - Hardware Based Hooking Technique

מאת כסיף דקל

קצת רקע

חברת מיקרוסופט נהגה לסבול במשך שנים רבות ממפתחים אשר ביצעו שינויי ליבה בחלקים קריטים במערכת ההפעלה חלונות, שינויים שפגעו ביציבות ובאיכות המוצר. מפתחים אלו בחרו לבצע שינויים אלו מכיוון שרצו לקבל אחיזה ושליטה טובה יותר במתרחש תחת מערכת ההפעלה. אלו פיתחו בעיקר תוכנות זדוניות (rootkits) ואף תוכנות Anti-Virus אשר ביצעו שינויים בקוד ובמבני נתונים קריטיים ב-Kernel על מנת להשיג את מטרותיהם.

כאשר קוד זדוני איכותי נטען למערכת הפעלה ומסווה עצמו היטב, לזהותו הופכת משימה מאתגרת ביותר. כך למעשה קיבלנו מציאות בה rootkits יכלו לעשות כל העולה על רוחם במערכת הפעלה לאורך זמן ובחשאיות גבוהה. כמו שצינתי קודם, לא רק תוקפים ניצלו את פונקצית שינוי גרעין מערכת ההפעלה, אלא גם חברות תוכנה כמו חברות אנטי וירוס שעשו שינויים תכופים בגרסאות השונות של מערכת ההפעלה. אלו, יחד עם התוקפים יצרו באופן עקיף נזק רב לענקית התוכנה מיקרוסופט, שכן, אי יציבות המערכת הייתה נושא שבשיגרה-דבר שלא הטיב עם מיקרוסופט ותדמיתה.

אחת מהבטחותיה הכי גדולות של Microsoft עבור לקוחותיה היא Backward-Compatibility. המשמעות של הבטחה זו היא שתוכנות אשר נכתבו לגרסה מסוימת של חלונות יעבדו גם בגרסאות חדשות יותר של מערכת ההפעלה.

כאשר יצאה גרסאת 64-bit למערכת ההפעלה חלונות, בוצעו שינויים אדירים במערכת ההפעלה. מיקרוסופט זיהתה את ההזדמנות להפטר מבעיית התאימות-לאחור מכיוון שמדובר במערכת הפעלה שונה. כלומר, קוד Kernel-Mode (דרייברים) אשר נכתב עבור מערכת ההפעלה חלונות בגרסאת 32-bit לא יכול לרוץ על גבי מכונה המריצה גרסאת 64-bit של חלונות, לכן היה אפשר לתכנן מחדש ללא משקולות מיותרות והתלות בגרסאות העבר.

אחד השינויים הקשורים בבעיית ה-Kernel Code Patching המוזכרת לעיל, היא הוספה של רכיב נוסף במערכת ההפעלה, אותו רכיב נקרא PatchGuard. תפקידו של הרכיב הוא למנוע מקוד הרץ ב--Kernel

Mode לבצע Hook-ים או כל שינוי אחר בקוד של המערכת הפעלה, כפי שהיו נוהגות בעבר חברות אנטי וירוס ותוכנות זדוניות שונות.

רכיב זה נעקף מספר פעמים מאז השקתו, אך עקיפות אלו נחסמו על ידי מיקרוסופט מכיוון שרובן היו כרוכות בזיהוי של PatchGuard בזיכרון וניטרולו. עד היום, לא קיימת עקיפה ל-PatchGuard שהיא יציבה לטווח ארוך.

חשוב לציין, שלצד הקדמה במערכות 64 ביט, במערכות 32 ביט לא קיימים רכיבי PatchGuard או Code Signing. עם זאת, קיימים פיצ'רים אשר לא מחייבים תאימות לאחור כגון CALLBACKים מסוימים בקרנל אשר מחייבים Code Signing.

Intel PT הינו פיצ'ר חומרתי במעבדי Intel החדשים (יחסית) המאפשר לבצע Execution Tracing בצורה יעילה יותר מאשר פתרונות Software שונים למיניהם.

הטכנולוגיה הוצגה לראשונה במעבדי Broadwell (דור חמישי) ונתמכת במלואה במעבדי SkyLake (דור שישי).

במאמר זה אציג שיטת Hooking חדשה, המאפשרת לבצע Hook על חלקים (קטעים) קריטיים במערכת הפעלה מבלי לגרום לטריגר של PatchGuard, שמגן על המערכת בין היתר מפני Hook-ים לפונקציות Service ו-interrupt-ים.

כיצד PatchGuard עובד

תפקידו של PatchGuard הינו זיהוי של שינויים במערכת ההפעלה וניטרולם.

בזמנים אקראיים ובחלקים מוחבאים בקרנל, ידגום PatchGuard מבנים מסוימים ויתריע במידה ובוצע שינוי כלשהו, דוגמה למבנים אשר נמצאים תחת ההגנה של PatchGuard:

- טבלת ה-SSDT (ראשי תיבות של System Service Descriptor Table), טבלה המכילה את כל פונקציות ה-Service במערכת ההפעלה, למעשה קיימים מספר טבלאות כאלה ולאחרונה נוספה אחת נוספת בעקבות WSL. נושא ה-SSDT כבר נסקר כאן ב-DigitalWhisper במאמר של שחק שלו, מומלץ לקרוא למי שלא מכיר את הנושא. http://www.digitalwhisper.co.il/files/Zines/0x3A/DW58-2-KiFastHooking.pdf
- טבלת ה- IDT (ראשי תיבות של Interrupt Descriptor Table). טבלה זו מכילה את כל ה-Interrupt-ים הקיימים במערכת ואת הקוד שאחראי לטפל בהן (ISRs).
- טבלת ה-GDT (ראשי תיבות של Global Descriptor Table). סגמנטציה בזיכרון, הטבלה נועדה לתאר איזורי זיכרון עבור המעבד.
- משתנים גלובאלים בקרנל שעלולים להיות מנוצלים לרעה (למשל nt!PspPicoRegistrationDisabled).

- הקוד עצמו בקרנל, שינוי של הקוד הינו שקול במידה מסוימת לשינוי של Entry בטבלה מסוימת, למשל שינוי הקוד ב-Nt!KiPageFault שקול לשינוי הרשומה בטבלת ה-IDT.
- רשומות מסוימות ב-MSRs, למשל הרשומה 0xc0000082 אשר מחזיקה את ה-"entry point" בקרנל לפונקציות service.

ועוד.

PatchGuard מחזיק עותק ו/או Checksum של המבנים המוגנים ומנטר שינויים בזמניים אקראיים (בערך כל 5-10 דקות). אם יימצא מבנה אשר ניזוק, PatchGuard "ינטרל" את הנזק בכך שיגרום ל-bugcheck, כלומר מסך כחול (או כפי שכולנו מכירים אותו: BSoD).

למעשה, מהותו של הרכיב הוא למנוע שינויים במערכת ההפעלה, לכן, עליו להיות מוסתר היטב ומשום שאחרת יוכלו כותבי rootkitים לנטרלו ולעשות כרצונם.

כיצד Intel PT עובד

הטכנולוגיה אוספת מידע אודות ריצה של קוד על גבי כל Hardware Thread, באמצעות שימוש בחומרה ייעודית אשר גורמת לירידה מינימאלית בביצועים. כאשר הריצה מסתיימת, יוכל המשתמש לשחזר את רצף הריצה במדויק.

המידע נאסף בצורת data packets דחוסים אשר נשלחים ל-buffer מרכזי המשמש לצורכי הניטור, הפאקטות מכילות מידע אודות ה-flow של התוכנה בזמן הריצה, למשל: יעד ה-branch (קפיצה של המעבד), האם ה-branch נלקח או לא וכו'.

להלן טבלה המכילה את כל הפקודות אשר משפיעות על ה-flow של התוכנית אשר מנוטרות ונשלחות כ-data packets:

סוג	פקודה
CONDITIONAL BRANCH	JA, JAE, JB, JBE, JC, JCXZ< JECXZ, JRCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ, LOOP, LOOPE, LOOPNE, LOOPNZ, LOOPZ
UNCONDITIONAL BRANCH	JMP (E9 xx, EB xx), CALL (E8 xx)
INDIRECT BRANCH	JMP (FF /4), CALL (FF /2)
NEAR RET	RET (C3, C2 xx)
FAR TRANSFERS	INTn, INTO, IRET, IRETD, IRETQ, JMP (EA xx, FF /5), CALL (9A xx, FF /3), RET (CB, CA xx), SYSCALL, SYSRET, SYSENTER, SYSEXIT, VMLAUNCH, VMRESUME

הטכנולוגיה משמשת בעיקר עבור Performance Monitoring, Diagnostic Code Coverage, Debugging, Fuzzing, Malware Analysis ו-Exploit Detection.

קיימים שלושה סוגים שונים של ניטור:

1. ניטור כולל של כל ה-user-mode/kernel-mode (current privilege level).
2. ניטור של תהליך ספציפי (Page Map Level 4).
3. ניטור של טווחי כתובות ספציפיים (ובסוג ניטור זה נתמקד במאמר).

הסיבה העיקרית ש-Intel PT כל כך מעניין אותנו, היא שלא ניתן לזהות אותו במערכת על ידי תוכנה והוא מאפשר לנו "להשתלט" על Threadים במיקום מאוד ספציפי ואסטרטגי. בקרוב נבין כיצד ניתן לממש זאת, אך ראשית נבין מעט איך משתמשים ב-Intel PT.

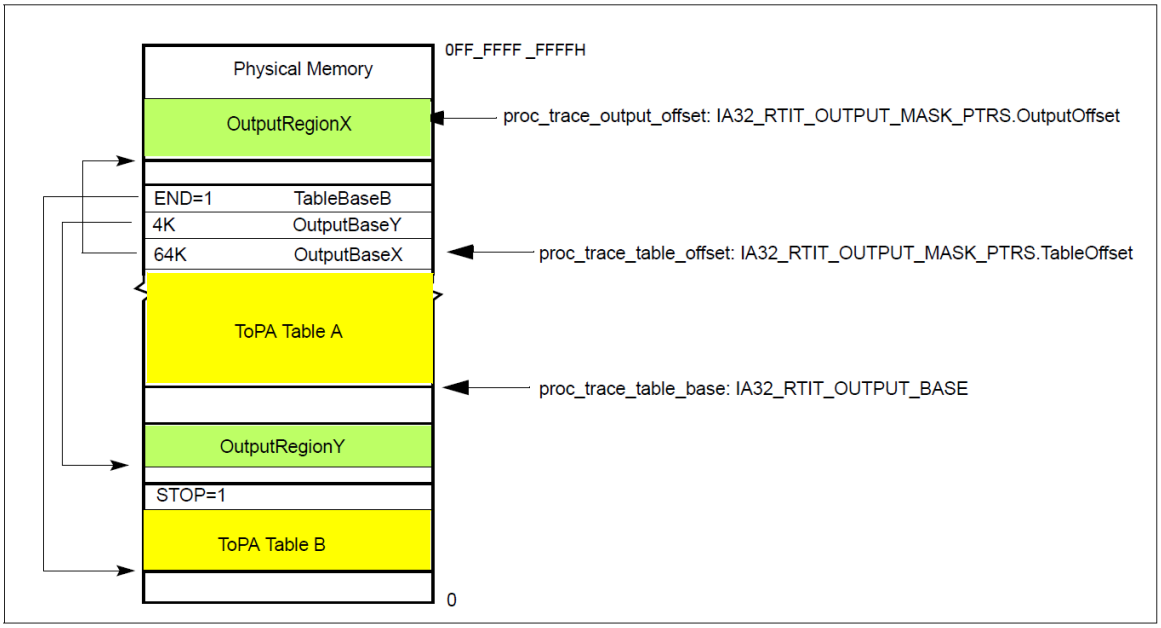
ניטור לפי טווח כתובות ספציפי - במצב זה, הטכנולוגיה מאפשרת לנטר ריצה רק כאשר המעבד מריץ קוד בנמצא בטווח כתובות מסוים הנקבע על ידנו. פילטור לפי טווחי כתובות מאופשר על ידי שינוי השדות ADDRn_CFG ב-IA32_RTIT_CTL MSR כאשר האות 'n' מסמלת את מספר הטווח (ניתן להגדיר מספר טווחים), בכל אחד מהשדות האלה, קיימים "תתי-שדות" IA32_RTIT_ADDRn_A ו-IA32_RTIT_ADDRn_B אשר מסמלים את כתובת ההתחלה וכתובת הסיום. להרחבה בנושא ניתן לקרוא בפרק 35 ב- Intel 64 and IA-32 Architectures Software Developer's Manual.

פלט הניטור של Intel PT מתחלק לשני סוגים עיקריים:

1. אזור יחיד ורציף של מרחב כתובות פיזיות.
2. אוסף של אזורים בזיכרון הפיזי, בגודל משתנה. אזורים אלה מקושרים יחד על ידי טבלאות מצביעים לאזורים אלה, הנקראים טבלת כתובות פיזיות (Table of Physical Addresses או בקיצור: ToPA).

איזור זיכרון רציף - כאשר הביטים ב-IA32_RTIT_CTL.ToPA וב- IA32_RTIT_CTL.FabricEn כבוים, פלט הניטור יישלח לזיכרון אחיד ורציף המוגדר ב-IA32_RTIT_OUTPUT_BASE. על מערכת ההפעלה להקצות את הזיכרון (למשל על ידי MmAllocateContiguousMemory). בשיטת פלט זו ניתן להפנות את הפלט גם לפורט MMIO או JTAG controller.

טבלת כתובות פיזיות - כאשר הביטים שהוזכרו לעיל דולקים, מנגנון ה-ToPA מופעל. המנגנון משתמש ברשימה מקושרת של טבלאות.



כל רשומה בטבלה מכילה מספר תכונות אודות אופן הניטור, מצביע לכתובת של ה-Buffer והגודל שלו. הרשומה האחרונה בטבלה תכיל מצביע לטבלה הבאה. המעבד מתייחס לאזורי הפלט השונים ב-ToPA כמאגר מאוחד. פירוש הדבר, שפאקטה אחת עשויה למלא את אזור פלט אחד שלם.

מנגנון ה-ToPA "נשלט" על ידי שלושה ערכים המנוהלים על ידי המעבד:

- **proc_trace_table_base** - הכתובת הפיזית של תחילת טבלת ה-ToPA הנוכחית. כאשר הניטור החל, המעבד יטען את הערך הזה מתוך MSR שהוזכר לעיל IA32_RTIT_OUTPUT_BASE. ובמהלך הניטור המעבד יעדכן את את הערך ב-MSR כאשר יש שינוי ב-proc_trace_table_base.

- **proc_trace_table_offset** - מכיל את הרשומה הנוכחית בטבלה שכרגע בשימוש (אשר מכילה את הכתובת של ה-Buffer הנוכחי). כאשר מפעילים את הניטור, המעבד יטען את הערך מתוך השדה MaskOrTableOffset ב-MSR שנקרא IA32_RTIT_OUTPUT_MASK_PTRS. כנ"ל לגבי ערך זה, במהלך הניטור המעבד יעדכן את הערך הנוכחי.

- **proc_trace_output_offset** - מצביע ל-offset של הכתיבה הבאה ב-Buffer, נטען מתוך השדה OutputOffset ב-IA32_RTIT_OUTPUT_MASK_PTRS.

בשונה מהראשונה, שיטת Output זו כוללת גם תכונות אשר תומכות בהשהיית והמשכת את הניטור.

על מנת להפעיל את הניטור, יש לעדכן את הערכים המתאימים לסוג הניטור שבשימוש ב-MSR שנקרא IA32_RTIT_CTL. ולאחר מכן להדליק את TraceEn.

במנגנון ToPA, כאשר ה-Buffer מתמלא או עומד להתמלא המעבד יקרא לפונקציית ה-PMI Handler (Performance Monitoring Interrupt) על מנת להודיע לנו שהבאפר מלא. אז אחרי שהבנו פחות או יותר איך Intel PT עובד, ניגש לעניין. לטכנולוגיה יש באמת שימושים נהדרים ולגיטימיים, אך עם זאת, ניתן לנצל את מנגנון ה- ToPA PMI Notify שמתרחש כאשר ה-Buffer מלא, על מנת להשיג שליטה על Threadים במקומות אסטרטגיים.

הבסיס של השיטה הזאת הוא לגרום למעבד "לקפוץ" אל קוד בשליטתנו (אל ה-PMI Handle) במיקום ספציפי, איך נעשה את זה ?

1. נקצה איזור זיכרון קטן באופן קיצוני עבור ToPA, כך המעבד יבצע Interrupt אל ה-PMI ב-no time, ניתן להשיג ייעילות גבוהה יותר באמצעות תכונה נוספת ב-ToPA בשם INT bit, כאשר ה-INT דלוק, המעבד יקפוץ אל ה-PMI Handler עוד לפני שהבאפר מלא. ניתן להרשם אל ה-PMI Handler בצורה הבאה:

```
HalSetSystemInformation(HalProfileSourceInterruptHandler,
sizeof(PMIHANDLER), (LPVOID)&hookroutine);
```

2. נפעיל את Intel PT Trace על איזור אסטרטגי בקרנל בו אנחנו מעוניינים, למשל ה-LSTAR MSR, שמחזיק את ה-kernel entry-point לפונקציות סרביס (nt!KiSystemCall64).

ניתן לקבל את הכתובת בצורה הבאה:

```
ULONG64 LSTAR = ((ULONG64(*)())"\xB9\x82\x00\x00\xC0\x0F\x32\x48\xC1\
xE2\x20\x48\x09\xD0\xC3")();
```

פקודה זו תייצר naked function עם הקוד הבא:

```
mov     ecx,0xc0000082
rdmsr
shl     rdx,0x20
or      rax,rdx
ret
```

כפי שהוזכר למעלה, ה-LSTAR יחזיר לנו את ה-kernel's EP עבור פקודות SYSCALL במערכות 64-ביט. נגדיר את הכתובות הזו ככתובת התחלה עד nt!KiSystemServiceUser ככתובת סיום. ברגע שקוד user-mode (באמצעות SYSCALL) או פונקציות ZW בקרנל יבצעו branch לטווח הכתובות האלה, המעבד יעשה tracing לריצה של הקוד.

3. משום שהקצאנו איזור זיכרון (Buffer) זעיר, הוא יתמלא ובאותו thread המעבד יבצע Interrupt אל ה-PMI Handler אשר בשליטתנו, בקוד שנמצא שם נרצה לבצע את ה-Hook ולאפס חזרה את ה-

MSRים המתאימים. במצב בו אנחנו מקבלים שליטה על ה-Thread, ניתן לשנות לחלוטין את אופן הריצה של הקוד, את האוגרים, את המחסנית ובעצם כל דבר אחר. מה שדומה מאוד להחלפת אופקודים בקוד של הקרנל או Hook-ים מסורתיים כגון החלפה של רשומה ב-SSDT וכו'.

דוגמה:

```
3: kd> k
 # Child-SP          RetAddr           Call Site
00 ffffd480`73ee3f38 fffff801`5625abf8 our_driver!HookRoutine <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
01 ffffd480`73ee3f40 fffff801`55ae1f97 hal!HalpPerfInterrupt+0x28 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
02 ffffd480`73ee3f70 fffff801`55b5e60a nt!KiCallInterruptServiceRoutine+0x87
03 ffffd480`73ee3fb0 fffff801`55b5ea57 nt!KiInterruptSubDispatchNoLockNoEtw+0xea
04 ffffd480`797b38f0 fffff801`55b67b06 nt!KiInterruptDispatchNoLockNoEtw+0x37
05 ffffd480`797b3a80 00007ff8`aa8a5034 nt!KiSystemServiceUser+0xf6 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
06 00000078`c1a7baf8 00007ff8`aa87dd17 ntdll!NtClose+0x14 <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
07 00000078`c1a7bb00 00007ff8`a6f7a31a ntdll!RtlImpersonateSelfEx+0xe7
08 00000078`c1a7bbb0 00007ff8`a7b8af95 KERNELBASE!ImpersonateSelf+0xa
09 00000078`c1a7bbe0 00007ff8`a7b74c60 CRYPT32!I_CryptIsAppContainerToken+0x5d
0a 00000078`c1a7bc30 00007ff8`a7b7403b CRYPT32!I_CertDllOpenSystemRegistryStoreProvW+0x4a0
0b 00000078`c1a7bd20 00007ff8`a7b74595 CRYPT32!CertOpenStore+0x2fb
0c 00000078`c1a7bdb0 00007ff8`a7b737ab CRYPT32!OpenPhysicalStoreCallback+0xd5
0d 00000078`c1a7beb0 00007ff8`a7b750ff CRYPT32!EnumPhysicalStore+0x67b
0e 00000078`c1a7c040 00007ff8`a7b74013 CRYPT32!I_CertDllOpenSystemStoreProvW+0x16f
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for mpengine.dll -
0f 00000078`c1a7c0f0 00007ff8`945a7fa8 CRYPT32!CertOpenStore+0x2d3
10 00000078`c1a7c180 00000000`00018000 mpengine!FreeSigFiles+0x1ef7a8
11 00000078`c1a7c188 00007ff8`00000000 0x18000
12 00000078`c1a7c190 00000000`00000000 0x00007ff8`00000000
```

הטכניקה המתוארת לעיל, ביישום הנוכחי, תעבוד בצורת race-condition, בתמונה ניתן לראות stack trace של קוד user-mode אשר קרא ל-nt!NtClose, אשר "נחטף" על ידי Intel PT.

יש לציין שכאשר המעבד יקרא לפונקציית ה-PMI שלנו זה יקרה ב-IRQL (עדיפות פסיקה) גבוה כך שאנו מוגבלים במה שאנחנו יכולים לעשות לכן כדאי להשתמש ב-WorkItem/DPC וכו'.

לצורך המחשה, את אותו דבר בדיוק ניתן לבצע עבור פונקציות IDT:

```
kd> k
 # Child-SP          RetAddr           Call Site
00 fffff801`8667bf38 fffff801`84643bf8 our_driver!HookRoutine                <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
01 fffff801`8667bf40 fffff801`8475ff97 hal!HalpPerfInterrupt+0x28
02 fffff801`8667bf70 fffff801`847dc60a nt!KiCallInterruptServiceRoutine+0x87
03 fffff801`8667bfb0 fffff801`847dca57 nt!KiInterruptSubDispatchNoLockNoEtw+0xea
04 ffffe400`32cd5840 fffff801`847e45b2 nt!KiInterruptDispatchNoLockNoEtw+0x37
05 ffffe400`32cd59d0 ffffa530`4ac66fb4 nt!KiPageFault+0xf2                    <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
06 ffffe400`32cd5b60 ffffa530`4ac663b2 win32kfull!EngTextOut+0x514
07 ffffe400`32cd6080 ffffa530`4ad3fe23 win32kfull!GreExtTextOutWLocked+0x1a92
08 ffffe400`32cd6820 ffffa530`4ac66a71 win32kfull!GrePolyTextOutW+0x123
09 ffffe400`32cd69e0 fffff801`847e5c93 win32kfull!NtGdiPolyTextOutW+0x381
0a ffffe400`32cd6a80 00007fff`daca6d04 nt!KiSystemServiceCopyEnd+0x13
0b 00000032`109bf7b8 00007fff`daae9004 win32u!NtGdiPolyTextOutW+0x14
0c 00000032`109bf7c0 00007fff`c5e35db8 gdi32full!PolyTextOutW+0x74
```

בתמונה זו אנו רואים ביצוע Hook לפונקציה nt!KiPageFault.

סיכום

ההרשמה ל-PMI Handler שקופה לאימפלמנטציה הנוכחית של PatchGuard. מכיוון שהטכניקה הזו משתמשת בחומרה על מנת לבצע Hook ולא בוצע שום שינוי למבני נתונים/קוד בקרנל, למיקרוסופט יהיה קשה מאוד לזהות ולחסום את השיטה הזו. לכן, שיטה זו תהיה עדיפה משיטות אחרות (על אף המורכבות ביישום שלה), בנוסף, ככל הנראה הטכניקה המוצעת תהיה future-proof ואמינה יותר גם עבור גרסאות הבאות של הקרנל.

על המחבר

כסיף בן 25 עובד כחוקר אבטחה בחברת CyberArk. לכל שאלה, הערה או פניה אחרת ניתן לפנות במייל: kasifdekel@gmail.com.