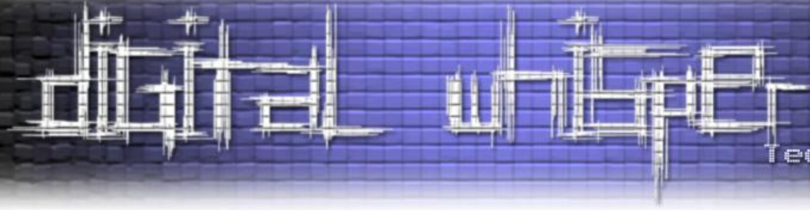


אלא לראותם בלבד: Mitigating (semi) new class of advanced threats

מאת עומר שליו

הקדמה

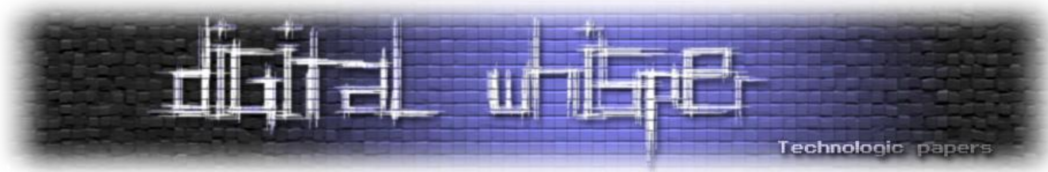
נתחיל כמיטב הקלישאה: בעולם אבטחת המידע, המרדף בין תוקפים למגנים הוא משחק בלתי פוסק של חתול ועכבר. בעולמות של כתיבת כלים שוהים, כולנו מכירים את אותם "הוקים" או טרמפולינות אשר תוקפים מציבים בקוד לגיטימי מסוים על מנת לנווט את ההרצה לקוד זדוני שלהם, שיבצע התערבות מסוימת ויחזור למסלול ההצרה התקין. דוגמא קלאסית, היא עריכת טבלת ה-syscall table (טבלת קריאות המערכת) על ידי תוקף. תוקפים שמשקיעים, גם יממשו חבילת הסתרה אשר תסתיר אותם מפני פונקציות מערכת ההפעלה של מיפוי קבצים/תהליכים וכו'.

עם זאת, בשנים האחרונות אנו ערים לשימוש גובר בכלים המבצעים קריאה ישירה של הזיכרון, בין אם על ידי מוצרי EDR, ובין אם על ידי כלים פורנזיים המבצעים Memory Dump וניתוח שלו בעזרת מערכות תומכות (מי מאיתנו לא שמע על הפרוייקט Volatility?). כיוון זה מתעצם עקב כניסת יכולות AI משמעותיות לתחום שככל הנראה מתישהו יוכלו לנתח את אותם Memory Dumps בצורה טובה ואוטומטית, ולאתר יכולות שהייה מתקדמות.

הכלים הנפוצים כיום (לפחות ב-github ☺) חשופים אל מול איומים כאלו, ולכן תוקפים אשר צריכים להיות יותר חשאיים ירצו לייצר איזשהי מעטפת הסתרה מפני פעולות הגנה. הדרך הטובה ביותר לא להיות מוקפץ על ידי EDR או כלי איתור היא שהוא פשוט לא ייראה שום זכר אליך.

במאמר זה, נבחן שתיים מהדרכים המובילות שבהן תוקפים משתמשים כדי לייצר כזאת חבילה וכך הם מסוגלים להסתיר את הקוד שלהם (לגמרי) מפני מרבית כלי האיתור ומוצרי ה-EDR הקיימים בחוץ. בנוסף, נציג כיצד אנו, כאנשי הגנה וחוקרים, יכולים להתמודד עם טכניקות אלו ולאתר אותן בכל זאת. בין היתר נדגים כיצד בעזרת פיצ'רים מסויימים של מעבדים מודרניים (כגון הרצה ספקולטיבית) אנו מסוגלים לאתר ולהתריע על איומים כאלו. השיטות שהוצגו במאמר ועוד רבות נוספות היו הבסיס למיזם CoreSights, אשר נועד לעזור לחברות האבטחה וה-EDR להיות מוכנות אל מול איומים שכאלו.

המאמר ידון בנושאים בסביבת לינוקס תחת ארכיטקטורת מעבד x86_64, אם כי הכל רלוונטי גם למערכות הפעלה נוספות (ואף ארכיטקטורות מעבד נוספות).



גלגול הדברים

בערך לפני חצי שנה, איכשהו במסגרת פרץ נוסטלגיה לא מוסבר, שוטטתי לי באתר מגזין Phrack. מצאתי מאמר ישן שאהבתי מאד - Mystifying the debugger for ultimate stealthness. המאמר מדבר על שימוש באוגרי דיבאג של המעבד ליצירת תוואי הרצת קוד חשאי בתוך מהערכת ההפעלה לינוקס.

ביום־יום שלי, אני עוסק בעולמות איתור קוד עוין, ונתקל במגוון רב של מוצרי איתור ו-EDR, המציגים יכולות מתקדמות, שונות ומגוונות. כשחשבתי שוב על ה-rootkit ההוא, הגעתי למסקנה שאל מול כל כלי מסחרי או פומבי שראיתי עד היום, ה-rootkit הזה לא ייתפס. כמובן שבתור תוקף אפשר "לעשות שטויות" במימוש, שיטת העלייה וציר הכניסה (אבל אלו אתגרים אחרים ומקבילים). פה הכוונה היא לליבה הטכנולוגית של הקוד השוהה, שהוא מושא חיפוש מרכזי עבור מוצרי אבטחה ו-EDR.

בשלב הזה התחלתי לחשוב על כל מיני דברים אשר יאפשרו לאתר איומים כאלו, מתוך הבנה אישית (ואולי אני טועה) שאיומים כאלו יתגברו בשנים הקרובות אל מול ההפתחות שאנו רואים בשוק ה-EDR שגדל באופן מרשים בשנים האחרונות וצפוי להמשיך לגדול.

שימוש ב-Debug Registers להסתרה במה"ע

כאמור, ממליץ לקרוא את המאמר המצוין Mystifying the debugger for ultimate stealthness ממגזין Phrack המפרט את הרעיון התיאורטי של מימוש כלי שכזה. דוגמת מימוש קונקרטית (בסביבת לינוקס) ניתן למצוא בפרוייקט subversive בגיטהאב.

הרעיון המרכזי הוא לנצל את מנגנון ה-debug במעבדי x86. כדי לגרום ל-Debug Exception, בנוסף לחריגה תוכנתית באמצעות האופקוד 0xcc. המעבד כולל אוגרים ייעודיים (DR0-DR7) המאפשרים להגדיר "נקודות עצירה" (Breakpoints) בחומרה. כאשר הקוד המורץ ניגש לכתובת זיכרון שצוינה באחד מה-Debug Registers, המעבד עוצר את הריצה הרגילה ומפעיל פונקציית טיפול ייעודית, הנמצאת מספר 1 ב-IDT (Interrupt Descriptor Table). הטכניקה הנ"ל יכולה להיעשות גם באופן זדוני על מנת להריץ קוד עוין כאשר המערכת קוראת לפונקציות מסוימות או אף ניגשת לכתובות זיכרון שהתוקף רוצה ליירט את הגישה אליהן.

נסביר איך זה מתבצע.

הגדרת נקודת העצירה (Breakpoint) בחומרה:

השלב הראשון הוא בחירת "כתובת עניין" במערכת, וקביעת נקודת עצירה עליה באמצעות ה- Debug Registers. כתובות כאלה יכולות להיות כתובות של פונקציות שנרצה להתערב לפני ההרצה שלהן (נגיד do_syscall_64) או מיקומים קריטיים כמו טבלת קריאות המערכת (Syscall Table), מבני נתונים של ה-VFS וכו'.

התוקף מגדיר את כתובת היעד באחד מארבעת אוגרי הכתובות (DR0-DR3). לאחר מכן, באמצעות אוגר הבקרה DR7, הוא קובע את תנאי העצירה: גישה לאיזה טווח כתובות מהכתובת שסומנה תגרום לחריגה (1,2,4,8 בתים) ואיזה סוג גישה יגרום לחריגה (קריאה,כתיבה,הרצה). בנוסף, הוא יגדיר את הדגל Global Enable (GE) באוגר DR7 כפעיל, כדי שנקודת העצירה תופעל.

דריסת ה-IDT

במצב רגיל, כאשר מתרחשת חריגת דיבאג (INT 1), מערכת ההפעלה מפעילה את שגרת הטיפול הסטנדרטית שלה. כדי ליירט את התהליך, התוקף דורס את הכניסה המתאימה ב-IDT ומפנה אותה לפונקציה זדונית משלו. מרגע זה, כל חריגת דיבאג במערכת תעבור תחילה דרך הקוד של התוקף.

טיפול בחריגה ויירוט ההרצה

כאשר תהליך כלשהו ניגש לכתובת המוגנת, המעבד יפעיל את חריגת הדיבאג והקוד של התוקף יקבל את השליטה. פונקציית הטיפול החדשה בודקת תחילה את אוגר הסטטוס DR6, המציין איזו מבין נקודות העצירה גרמה לחריגה.

לאחר זיהוי המקור, הקוד יכול להחליט כיצד לפעול. לדוגמה, הוא יכול להריץ פונקציה ייעודית המתאימה לכתובת שגרמה לחריגה. בקוד המקור של subversive ניתן לראות לוגיקה זו:

```
for (int i = 0; i < 4; i++) {
    if ((dr6 & (DR6_TRAP0 << i)) && bps.handlers[i]) {
        bps.handlers[i](regs);
        /* FIXME: RF only if exec breakpoint */         regs->flags |=
X86_EFLAGS_RF;
        return 0;
    }
}
```

הלולאה עוברת על ארבע נקודות העצירה האפשריות. אם היא מזהה שהחריגה נגרמה מנקודת העצירה שהוגדרה (bps.handlers[i]) היא מפעילה את שגרת הטיפול הרלוונטית.

בסיום, היא מגדירה את דגל Resume Flag (RF) באוגר הדגלים כדי למנוע מהמעבד להפעיל את אותה חריגה שוב מיד עם החזרה לביצוע רגיל.

לדוגמה, אם נקודת העצירה הוגדרה על הפונקציה do_syscall_64, הקוד הזדוני יכול לבדוק את הפרמטרים של קריאת המערכת. ה-subversive rootkit משתמש בטכניקה זו כדי להעניק הרשאות root לתהליך ספציפי או להפעיל פונקציות נסתרות אחרות. הקוד הבא מדגים כיצד subversive בודק אם קריאת המערכת uname מופעלת עם "מספר קסם" ספציפי, ובהתאם מעלה את הרשאות התהליך:

```
void handle_do_syscall_64_breakpoint(struct pt_regs *regs)
{
    struct pt_regs *do_sys_regs = (struct pt_regs *)regs->si;
    int nr = regs->di;
    long magic_number = do_sys_regs->di;

    if (nr == __NR_uname) {
        if (magic_number == MAGIC_NUMBER_GET_ROOT && ksyms.commit_creds &&
ksyms.prepare_kernel_cred) {
            pr_debug("%s: commit root creds\n", __func__);
            ksyms.commit_creds(ksyms.prepare_kernel_cred(NULL));
        } else if (magic_number == MAGIC_NUMBER_DEBUG_RK) {
            x86_hw_breakpoint_debug();
        }
    }
}
```

טכניקה זו יעילה במיוחד מכיוון שהיא מסתמכת על מנגנון חומרתי שקוף למרבית כלי הניתוח ברמת התוכנה. היא מאפשרת לתוקף להימנע משינויים קבועים בקוד המקור של הקרנל (Patching), ובכך מקשה על איתורו.

אבל זה לא הסוף: מנגנון ההסתרה העצמית

עד כה, ראינו כיצד ניתן לנצל את ה-Debug Registers כדי ליירט את זרימת ההרצה. אך מה קורה אם כלי איתור או EDR מנסה לקרוא את ה-Debug Registers כדי לבדוק אם נעשה בהם שימוש זדוני? כאן נכנס לתמונה מנגנון הגנה נוסף, המאפשר ל-rootkit להסתיר את עצם קיומו (גם הוא מבית היוצר אינטל כמובן ☺).

במעבדי אינטל קיים דגל מיוחד באוגר DR7 הנקרא GD (General Detect). כאשר דגל זה מופעל, כל ניסיון גישה לאחד מה-Debug Registers (DR0-DR7) – בין אם לקריאה או לכתיבה – יגרום באופן מיידי לחריגה (Debug Exception) גם הוא. יכולת זו מספקת ל-rootkit מנגנון רב-עוצמה: הוא יכול לדעת מתי מישהו מנסה לבחון אותו ויכול להגיב בהתאם כדי להסתיר את עקבותיו.

ניתן למצוא במימוש של subversive כיצד מפעילים את הדגל הזה כחלק מתהליך הגדרת נקודת העצירה. הפונקציה x86_hw_breakpoint_register לא רק מגדירה את כתובת המעקב אלא גם דואגת להפעיל את דגל ה-GD באוגר DR7, כפי שניתן לראות בקוד:

```c
int x86_hw_breakpoint_register(int dr_nr, unsigned long addr, int type,
                               int len, bp_handler handler)
{
    unsigned long dr7 = 0;

    if (dr_nr >= 4 || dr_nr < 0)
        return -1;

    bps.dr[dr_nr] = addr;
    bps.handlers[dr_nr] = handler;

    dr7 = (len << 2) | type;
    dr7 <<= (16 + dr_nr * 4);    /* len and type */
    dr7 |= 0x2 << (dr_nr * 2);  /* global breakpoint */
    // Set the General Detect (GD) flag along with other settings
    bps.dr7 |= bps.dr7 | dr7 | DR7_GE;
    pr_debug("%s: dr%d=0x%lx dr7=%lx\n", __func__, dr_nr, addr, bps.dr7);
    on_each_cpu_set_dr(dr_nr, bps.dr[dr_nr]);
    on_each_cpu_set_dr(7, bps.dr7);

    return 0;
}
```

כאמור הפעלת דגל ה-GD יוצרת שכבת חשאיות כמעט מושלמת. אם קוד כלשהו ינסה לקרוא את ה- Debug Registers כדי לגלות את נקודות העצירה שהוגדרו, ה-rootkit יידע על כך ויקבל את השליטה. בשלב זה, עומדות בפניו שתי אפשרויות עיקריות להגיב:

1. **הפתרון הפשוט: קפיצה קדימה.** הפתרון הקל הוא פשוט לקדם את מצביע ההוראות (RIP) בארבעה בתים (אורך פקודת <mov <gp register>, <debug register>), ובכך "לדלג" מעל פקודת ה-mov שניסתה לקרוא את אוגר הדיבאג. (נציין כי פקודת mov כפי שתואר היא הדרך היחידה לקרוא את ה- Debug Registers ב-ISA). עם זאת, בעיניים של תוקף, גישה זו בעייתית. האוגר הכללי שאליו ניסו לקרוא ישאר עם ערכו המקורי, מה שעלול לעורר חשד ולחשוף את ההתערבות.
2. **הפתרון הנכון: אמולציה והחזרת ערך נקי.** הגישה ה-(קצת) מתוחכמת והנכונה יותר היא לבצע אמולציה של הפקודה המקורית. ה-rootkit מנתח (סוג של disassemble) את הפקודה בכתובת שממנה הגיעה החריגה (הכתובת המאוחסנת ב-regs->rip). באמצעות ניתוח זה, הוא מזהה לאיזה אוגר כללי (למשל, RAX, RBX) ניסתה הפקודה להעביר את תוכן ה-Debug Register. לאחר מכן, במקום להעביר את הערך האמיתי (והזדוני), הוא מחזיר ערך "נקי" – ממבנה נתונים פנימי שהוא שומר. כך, הקוד הבודק מקבל את הרושם שה-Debug Registers אינם בשימוש.

המימוש של טכניקה זו בפרויקט subversive נמצא בפונקציה emulate_cpu. פונקציה זו אחראית על פיענוח הפקודה המקורית וביצוע המניפולציה הנדרשת כדי להחזיר ערך מזויף.

כך, בחסות ארכיטקטורת אינטל, אנו מקבלים מנגנון המאפשר ל-rootkit להפוך ל-(כמעט) בלתי נראה. הוא לא רק מסתיר את פעולתו, אלא גם מסתיר את עצם קיומו אפילו מפני קוד איתור שרץ ברמת הקרנל ומנסה באופן אקטיבי לחשוף אותו. נחמד מאוד.

שיפורים למתקדמים

טכניקת ההסתרה באמצעות ה-Debug Registers מרשימה ביעילותה, אבל לא לגמרי סיימנו, יש מקום לשיפורים. ישנן שתי דרכים עיקריות לשפר את החמקמקות של רוטקיט כזה עוד יותר.

1. הגנה על ה-IDT מפני קריאה

שינוי הכניסה של חריגת הדיבאג (INT 1) ב-IDT לכתובת תוקף היא התנהגות רועשת למדי (ואפשר אף לטעון שאין פואנטה בכזה rootkit מתוחכם אם בסוף דורסים את ה-IDT..).

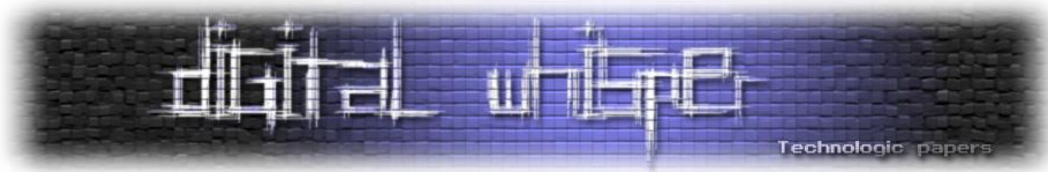
ניתן להשתמש באותו מנגנון בדיוק כדי להגן על ה-IDT עצמו. על ידי הגדרת נקודת עצירה נוספת (למשל, באמצעות DR1) על הכתובת הספציפית ב-IDT שבה נמצאת הכניסה של INT 1, אנו יכולים ליירט כל ניסיון קריאה של הכתובת הזו. כאשר כלי האיתור ינסה לקרוא את ערך הכניסה, תופעל חריגת דיבאג, וה-rootkit יקבל את השליטה. בדומה לאופן שבו הוא מסתיר את ה-Debug Registers, הוא יחזיר לכלי האיתור את הערך המקורי וה"נקי" של הכניסה, כאילו לא שונתה מעולם.

הערה: אם ממומש נכון, הנ"ל לא יגרום ל-double fault.

2. דריסת מיקום פחות "רועש" בשרשרת הטיפול בחריגה

ה-IDT הוא מבנה נתונים מרכזי ומוכר היטב בליבת מערכת ההפעלה, ולכן הוא מהווה יעד נפוץ לבדיקה של על ידי כלי איתור. כדי להפחית את הסיכוי להתגלות, ניתן להימנע מלשנות את ה-IDT ישירות ולבחור בנקודת התערבות עמוקה ופחות צפויה בשרשרת.

בלינוקס, למשל, קיים מנגנון שנקרא die notifier chain. זוהי רשימה של פונקציות (notifiers) הנקראות בזו אחר זו כאשר מתרחשת חריגה חמורה במערכת (die event), כולל חריגת דיבאג. במקום לדרוס את ה-IDT, תוקף יכול לרשום "notifier" משלו בשלב מוקדם בשרשרת. כך, הקוד הזדוני שלו יופעל בכל פעם שתתרחש



חריגת דיבאג, מבלי להשאיר עקבות ב-IDT עצמו. שינוי כזה קשה יותר לאיתור מכיוון שהוא מתבצע במבנה נתונים פנימי ופחות מנוטר.

הערה חשובה: למרות כל שכבות ההסתרה, לא ניתן להימנע לחלוטין מביצוע שינוי כלשהו בזיכרון (hook) כדי לבסס את תשתית ה-rootkit (ניתן רק להסתיר אותו). בין אם מדובר ב-IDT, ב-notifier chain או במנגנון אחר, חייבת להיות נקודה ראשונית שבה הקוד הזדוני משתלב. לא ניתן להגן על התשתית הזו באמצעות נקודת עצירה (breakpoint) נוספת, מכיוון שניסיון כזה ייצור לולאה אינסופית: גישה לתשתית הטיפול בחריגת דיבאג, שדרסנו, תגרום לחריגה, שתטופל על ידי אותה תשתית, שתגרום שוב לחריגה, וחוזר חלילה.

שיטות איתור של איומים כאלו

כפי שראינו, אם הכלי ממומש נכון, איתורו באמצעות סריקת זיכרון פשוטה יהיה מאתגר. ההסתמכות על מנגנוני חומרה והסתרת עקבות בזמן אמת הופכים אותו לכמעט בלתי נראה מתוך מערכת ההפעלה. עם זאת, פיתחנו מספר שיטות המאפשרות גילוי של איומים כאלו.

1. מיפוי מחדש של כתובת ה-IDT

הרעיון הבסיסי הוא שאם אנו חושדים שהגישה לכתובת וירטואלית מסוימת (כמו זו של ה-IDT) מנוטרת או מנותבת מחדש, נוכל לעקוף זאת על ידי יצירת מיפוי וירטואלי חדש לאותה כתובת פיזית. כך, נוכל לקרוא את התוכן האמיתי של הזיכרון מבלי להפעיל את מנגנון ההסתרה של ה-rootkit (המנגנון של ה-Debug Registers עובד ברמת הכתובת הוירטואלית ולא הכתובת הפיזית), שסביר להניח מגן רק על הכתובת הווירטואלית המקורית.

בלינוקס, ניתן לבצע זאת באמצעות הפונקציה kmap, המקבלת דף זיכרון פיזי וממפה אותו לכתובת וירטואלית חדשה במרחב הכתובות של הקרנל. לדוגמה, כדי לקרוא את התוכן האמיתי של הכניסה INT 1 ב-IDT, נוכל לבצע את הפעולות הבאות:

```
p = pfn_to_page(__pa(entry_ptr) >> PAGE_SHIFT);
if (!p) {
    // Error
    ...
}
entry_remap = kmap(p);
```

לאחר מכן, נוכל לקרוא מהכתובת entry_remap ולהשוות את התוכן לערך המצופה. אם קיים פער, חשפנו את ההתערבות. כך נראה הפלט הצפוי מהרצת קוד כזה (הדוגמא הזאת קוראת את הכניסה הראשונה ב-syscall table והיא כמובן __x64_sys_read קודם כל בצורה רגילה ואז בעזרת מיפוי מחדש).

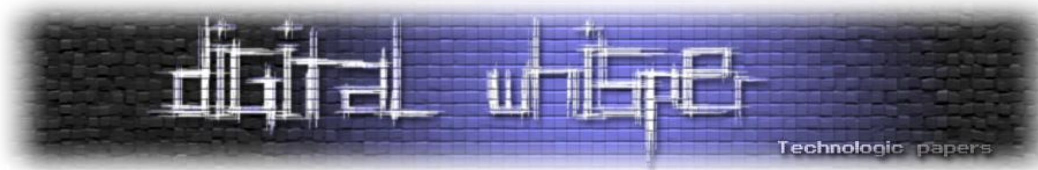
```
=== READ REGULAR (8 BYTES) ===
Step 1: Discovering syscall table address...
Syscall table discovered at: 0xffffffff82000260
Step 2: Reading 8 bytes from syscall table using regular (non-speculative) access...
Successfully read 8 bytes from syscall table!
Regular read result: 0x00000000deadbeef
root@syzkaller:~# /ctrl read_remap
DR-Detecto[ 1020.221947] found 'sys_call_table' address: ffffffff82000260
r Control Program
===========================

=== SYSCALL TABLE REMAP READ ===
Step 1: Discovering syscall table address...
Syscall table discovered at: 0xffffffff82000260
Step 2: Reading 8 bytes using page remapping (pfn_to_page + kmap)...
Successfully read syscall table using page remapping!
Remap read result: 0xffffffff811c7ed0
root@syzkaller:~# grep ffffffff811c7ed0 /proc/kallsyms
ffffffff811c7ed0 T __x64_sys_read
root@syzkaller:~#
```

לתוקף יש אפשרויות רבות לטפל גם בזה, ביניהן:

- **Bp על פונקציית מיפוי פנימית של מערכת ההפעלה** - ניתן לשים נקודת עצירה (bp) על פונקציית המיפוי הפנימית של מערכת ההפעלה, כמו kmap, וכך להתערב במיפוי הדף עצמו. במצב כזה, התוקף יכול לעקוב אחר הקריאה לכתובת החדשה שממופה ולבדוק האם כתובת העניין שלו נמצאת בדף שממופה.
- **התערבות בקבלת הכתובת הפיזית מלכתחילה** - ניתן להתערב בשלבים מוקדמים יותר, בעת קבלת הכתובת הפיזית, ולמפות לדף דמה שנמצא בשליטת התוקף. הכוונה היא להתערב בתהליך תרגום הכתובת הווירטואלית ולא במיפוי עצמו. נציין כי גישה זו עלולה להיות לא גנרית כי עבור כתובות רבות תרגומן אינו עובר הליכה מלאה בטבלת הדפים (למשל בלינוקס בשביל איזורים הנמצאים ב-image הקרנלי נחסר פשוט __START_KERNEL_map מהכתובת ולא נלך בטלבת הדפים.. כמו כן, כתובות שנמצאת ב-TLB גם מצריכות טיפול נפרד). אך מצב כזה אפשרי אם התוקף יודע בוודאות שמגן המערכת יבצע תרגום לכתובת פיזית טרם השימוש. לדוגמה, במקרים שבהם נוצר צורך לגשת לכתובת פיזית והדבר מצריך הליכה בטבלת הדפים, התוקף יכול לשים bp על כתובת ה־PTE הרלוונטית.

2. שימוש בדיבאגר להגדרת Hardware Breakpoints

שיטה פשוטה אך יעילה היא לנסות להשתמש בעצמנו במשאבים שה-rootkit מנצל. אם נפעיל דיבאגר (כמו GDB) וננסה להגדיר מספר נקודות עצירה בחומרה (hardware breakpoints), נוכל להיתקל בכמה תרחישים שיעידו על קיום ה-rootkit.בהתאם למימוש של ה-rootkit, הניסיון שלנו עלול לגרום לקריסות, התנהגות לא צפויה, או פשוט להיכשל. כמו כן, מכיוון שלמעבד יש רק ארבעה Debug Registers, אם ה-rootkit כבר משתמש באי אילו מהם, לא נוכל להגדיר ארבע נקודות עצירה משלנו. מוזמנים לנסות ☺



3. קריאה ישירה של ה-Debug Registers (Debug Registers)

גישה נוספת היא לנסות לקרוא ישירות את ה-Debug Registers, ובפרט את DR7. אוגר זה מכיל את קונפיגורציית נקודות העצירה: אילו מהן פעילות, סוג הגישה שהן מנטרות (קריאה, כתיבה, הרצה), והאם הן גלובאליות או לוקאליות. אנו נתעניין במיוחד בנקודות עצירה גלובאליות. בנוסף, הדגל GD. DR7 יציין אם מופעלת הגנה על ה-Debug Registers עצמם.

הבעיה היא, כפי שראינו, ש-rootkit שבנוי כהלכה מיירט את הגישה לאוגרים אלה ומחזיר ערכים נקיים. עם זאת, קיים תרחיש מעניין: במעבדי אינטל ישנים יותר, או כאשר תכונת ה-Debug Extensions אינה מופעלת (CR4.DE=0), ה-Debug Registers DR4 ו-DR5 משמשים כ-alias (שם חלופי) לאוגרים DR6 ו-DR7.

מתוך: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

כותבי כלים רבים, כולל subversive שניתחנו, לא תמיד מטפלים במקרה קצה זה, מכיוון שפקודת המכונה (opcode) לקריאת DR5/DR4 שונה מזו של DR7/DR6. לכן, ייתכן שנוכל לקרוא את הערך האמיתי של DR7 דרך DR5 ולעקוף את מנגנון ההסתרה. חשוב לציין שבמערכות הפעלה מודרניות, הדגל CR4.DE כמעט תמיד דלוק, מה שהופך שיטה זו לרלוונטית בעיקר במכונות וירטואליות מסוימות או על חומרה ישנה.

הנה דוגמא למכונה בה זה עובד

זה המצב שה-rootkit אינו טעון:

```
root@syzkaller:~# /ctrl dr6_dr7
DR-Detector Control Program
===========================

Reading DR6/DR7 debug registers...

=== DR6/DR7 DEBUG REGISTERS ===
DR6 (Debug Status):  0x00000000ffff0ff0
DR7 (Debug Control): 0x0000000000000400
===============================

root@syzkaller:~# /ctrl dr_alias
DR-Detector Control Program
===========================

Testing DR4/DR5 alias read...

=== DR4/DR5 ALIAS READ RESULTS ===
CR4 register value: 0x00000000000006f0
CR4.DE bit: CLEAR (0)
DR4/DR5 aliases available: YES
DR4 alias value: 0x00000000ffff0ff0
DR5 alias value: 0x0000000000000400

Note: When CR4.DE=0, DR4 is an alias for DR6 and DR5 is an alias for DR7
=====================================
```

וזה המצב שהוא כן טעון:

```
root@syzkaller:~# /ctrl dr6_dr7
DR-Detector Control Program
===========================

Reading DR6/DR7 debug registers...

=== DR6/DR7 DEBUG REGISTERS ===
DR6 (Debug Status):  0x00000000ffff0ff0
DR7 (Debug Control): 0x0000000000000400
===============================

root@syzkaller:~# /ctrl dr_alias
DR-Detector Control Program
===========================

Testing DR4/DR5 alias read...

=== DR4/DR5 ALIAS READ RESULTS ===
CR4 register value: 0x00000000000006f0
CR4.DE bit: CLEAR (0)
DR4/DR5 aliases available: YES
DR4 alias value: 0x0000000000000000
DR5 alias value: 0x0000000000000000

Note: When CR4.DE=0, DR4 is an alias for DR6 and DR5 is an alias for DR7
=====================================
```

4. מדידת זמנים של פקודת הקריאה ל-DR7

טכניקה זו מבוססת על ההנחה שלכל התערבות של תוקף יש פגיעה מסוימת בזמני הריצה. קריאה רגילה של אוגר במעבד היא פעולה מהירה ביותר. לעומת זאת, אם כלי מיירט את הקריאה, הוא צריך לבצע סדרת פעולות: לזהות את מקור החריגה, לנתח את הפקודה המקורית, להכין ערך מזויף, ולהחזיר את השליטה. כל התהליך הזה, גם אם הוא מהיר, לוקח יותר זמן מאשר קריאת חומרה ישירה.

על ידי מדידת זמן הביצוע של פקודת קריאה ל-DR7, נוכל לזהות חריגות. נבצע את קוד המדידה הבא:

```
u64 time_before, time_after, delta;

asm volatile ("rdtscp" : "=A"(time_before) : : "rcx");
asm volatile ("mov %%dr7, %0" : "=r"(response->dr7_value));
asm volatile ("rdtscp" : "=A"(time_after) : : "rcx");

delta = time_after - time_before;

pr_info("DR7 timing: before=%llu, after=%llu, elapsed=%llu cycles, dr7=0x%016llx\n",
      time_before, time_after, delta, response->dr7_value);
```

כאשר ה-rootkit מותקן, נראה הבדל משמעותי בזמני הביצוע (המשתנה delta), מכיוון שנוסף interrupt והפעלת שגרת טיפול – מה שמאריך את זמן הקריאה מהאוגר. בעזרת מדידה זו ניתן לחשוף מנגנוני הסתרה שכאלו.

```
=== DR7 TIMING MEASUREMENT ===
Measuring cycles required to read DR7 register using rdtscp...

=== DR7 TIMING RESULTS ===
DR7 register value:     0x0000000000000400
TSC before DR7 read:    996052734
TSC after DR7 read:     996057241
Cycles elapsed:         4507
```

נראה דוגמא. באותה מכונה לפני ואחרי התקנת הכלי:

```
=== DR7 TIMING MEASUREMENT ===
Measuring cycles required to read DR7 register using rdtscp...

=== DR7 TIMING RESULTS ===
DR7 register value:     0x0000000000000400
TSC before DR7 read:    3370592577
TSC after DR7 read:     3370616467
Cycles elapsed:         23890
```

כפי שניתן לראות, הבדל משמעותי בזמנים. על ידי השוואה עם תזמון של פקודות אחרות/ניתוח חכם ניתן לזהות את האנומליה.

שימוש בהרצה ספקולטיבית כדי להתמודד עם איומים כאלה

השיטות שהצגנו עד כה התמקדו בעיקר באיתור אנומליות סביב השימוש ב-Debug Registers. גם אם נצליח לזהות שנעשה בהם שימוש זדוני, זה לא מבטיח שיהיה לנו קל להבין את מלוא היקף התקיפה או לצייר את שרשרת הפעולות המלאה של התוקף. אנו זקוקים לדרך מהימנה לקרוא מיקומים רגישים בזיכרון, כמו ה-IDT, באופן שיהיה חסין בפני מנגנוני ההסתרה של ה-rootkit, כדי לראות את תוכנם האמיתי ולזהות שינויים (hooks) שבוצעו בהם.

כאן נכנסת לתמונה תכונה ארכיטקטונית המלווה את מעבדי x86 כבר קרוב ל-30 שנה (ולעיתים גם עושה צרות לא צפויות ☺): והיא כמובן הרצה ספקולטיבית (Speculative Execution).

מהי הרצה ספקולטיבית?

הרצה ספקולטיבית היא טכניקת אופטימיזציה שנועדה להאיץ את ביצועי המעבד. במקום להמתין לתוצאה של פעולה איטית (כמו קריאה מהזיכרון או תוצאה של תנאי), המעבד "מהמר" מה תהיה התוצאה וממשיך להריץ פקודות עתידיות על בסיס אותו הימור. אם ההימור היה נכון, המעבד חסך זמן יקר. אם ההימור היה שגוי, המעבד זורק את התוצאות של החישוב השגוי וממשיך מהנקודה הנכונה, כאילו כלום לא קרה. התהליך כולו שקוף לחלוטין למערכת ההפעלה ולתוכנה הרצה. למידע מפורט יותר (בעברית) על המנגנון והשלכותיו, מומלץ לעיין במאמר Meltdown and Spectre מהגיליון ה-91 (במאמר זה לא אתן רקע בסיסי בנושא, ובפרט לא אסביר בכלל על Spectre, יש לוודא ששולטים בנושא לפני המשך הקריאה).

הרעיון המרכזי הוא שניצול של מנגנון זה מאפשר לנו לבצע פעולות "מבלי באמת לעשות אותן". אף על פי שהתוצאות של הרצה ספקולטיבית שגויה נזרקות, הפעולות עצמן משאירות עקבות ב-cache של המעבד. באמצעות טכניקות ניתוח שנציג, ניתן לשחזר מידע שנוצר/עובד בעת ההרצה הספקולטיבית.

תכנון Spectre Gadgets לצורכי איתור

התקפות המנצלות הרצה ספקולטיבית, המכונות Spectre, משתמשות ב"גאדג'טים" – רצפי קוד קצרים הגורמים למעבד לבצע באופן ספקולטיבי פעולות שלא היו אמורות להתבצע. ישנם מספר סוגים של גאדג'טים, ביניהם:

- **Type 1 (Bounds Check Bypass)**: גורם למעבד לקרוא באופן ספקולטיבי מחוץ לגבולות של מערך בזיכרון (ובכך לקרוא כתובות זיכרון בשליטת התוקף, לעיתים מחוץ ל-context שלו).
- **Type 2 (Branch Target Injection)** מרעיל את מנגנון חיזוי הקפיצות של המעבד כדי שיקפוץ ספקולטיבית לכתובת נשלטת.
- **Type 4 (Speculative Store Bypass)**: מנצל מצב שבו המעבד קורא ערך ישן מהזיכרון באופן ספקולטיבי, לפני שערך חדש הספיק להיכתב אליו.

הנקודה החשובה ביותר לענייננו היא שבעת הרצה ספקולטיבית, חריגות (Exceptions) כמו חריגת הדיבאג (Debug Exception) לא מופעלות באופן מיידי. המעבד רושם את קיום החריגה אך ממשיך בהרצה הספקולטיבית, ומפעיל את הטיפול בחריגה רק אם מתברר שההרצה הייתה נכונה.

עובדה זו פותחת בפנינו דלת: אנו יכולים לבנות Spectre Gadget משלנו שיגרום למעבד לקרוא באופן ספקולטיבי את כתובת היעד שלנו (למשל, כניסה ב-IDT). מכיוון שחריגת הדיבאג שה-rootkit שתל לא תופעל במהלך ההרצה הספקולטיבית, נוכל לדלות את התוכן האמיתי של הזיכרון דרך תופעות הלוואי שהקריאה

משאירה במטמון. במימוש שלנו, השתמשנו בגאדג'ט SpectreRSB ווריאנט, כמו כן מימשנו גם גאדג'ט מסוג Spectre Type 1, אך היות והמימוש של ה-SpectreRSB פשוט וקצר משמעותית (אינו דורש שלב "אימון" – נסביר אותו).

חשוב להדגיש: אנו לא עושים שימוש בשום חולשה כאן, אלא מנצלים את אופן הפעולה המובנה של כל מעבד מודרני. בסוף הרצה ספקולטיבית הינה חשובה ומשמשת כפיצ'ר להאצת ביצועי המעבד. לכן הכלים שנציג יעבדו בכל גרסת מעבד מודרני. מסיבה זאת אגב, ההגנות הקיימות כיום נגד Spectre מתמקדות במעבר בין contexts: בין פרוססים שונים, מעבר בין יוזר לקרנל וכו'.

הערה נוספת: היות וכלי האיתור שלנו פועל ברמה קרנלית, יש לנו את היכולת להרכיב את הגאדג'טים בעצמנו ולשלוט בסביבת הריצה, ובכך להבטיח שהפקודה שאנו רוצים לבדוק אכן תרוץ באופן ספקולטיבי.

כמה מילים על ה-RSB

ה-RSB (Return Stack Buffer) הקיים הוא באפר חומרתי ייעודי הנמצאת בתוך מעבדים מודרניים. מטרתו היא לחזות כתובות חזרה מפונקציות במהלך הרצה ספקולטיבית. במעבדים מודרניים חלון הספקולציה הינו יחסית גדול, בפרט גדול יותר מאורך פונקציה טיפוסית. קריאת ערך ה-return address מהמחסנית מערבת גישה לזיכרון, מה שלוקח (ביחס לגישה ל-cache) זמן רב. לכן, כמנגנון ייעול המעבד שומר בנוסף ב-cahe את כתובת החזרה כך שיוכל להמשיך להריץ בצורה ספקולטיבית גם לאחר החזרה מהפונקציה. בעת פקודת call בנוסף לכתיבה למחסנית, המעבד רושם את כתובת החזרה גם ב-rsb. כאשר הוא נמצא בריצה ספקולטיבית ונתקל בפקודת ret, הוא ישלוף מה-rsb את כתובת החזרה וימשיך בספקולציה. ובכן, בואו נראה איך זה נראה בפועל. נשתמש בפונקציה speculate_read_byte, שמממשת קריאה לכתובת זיכרון באופסט מסוים.

```
// Speculative execution function for reading a byte from memory using inline GCC assembly
static void noinline __attribute__((naked)) speculate_read_byte(const char* detector, const void* target_addr, size_t offset)
{
    // call the inner function in C (not in inline assembly)
    speculate_read_byte_inner();

    // The rest of the code in inline assembly (speculatively executed after mispredicted return)
    asm volatile (
        "movzbl (%1, %2, 1), %%eax\n\t"             // Read byte from target_addr + offset
        "and $255, %%rax\n\t"                       // Keep only LSB
        "shl $0xc, %%eax\n\t"                       // Multiply by 4096
        "movzbl (%%rdi, %%rax, 1), %%eax\n\t"       // Load byte from detector[byte_value * 4096]
        :
        : "D" (detector), "r" (target_addr), "r" (offset)
        : "rax", "memory"
    );
}
```

הפונקציה speculate_read_byte_inner, אחראית ליצירת חלון הרצה ספקולטיבית על ידי הרצת רצף פקודות איטיות יחסית (פקודת imul - ראו את הלולאה בתווית 1) ושימוש תלוי בתוצאתן.

כדי להימנע מאופטימיזציות וסידור מחדש של פקודות על ידי המעבד, נרצה שטעינה של ה-stack pointer לא תתבצע לפני תחילת הספקולציה: לכן נגרום לה להיות תלויה בערך eax שאינו ידוע עדיין ולכן המעבד פשוט יחכה עם הפקודה הנ"ל.

```
// The "inner" function: slow dependent instructions and stack pointer manipulation
static void noinline __attribute__((naked)) speculate_read_byte_inner(void)
{
    __asm__ __volatile__ (
        // Lots of slow dependent instructions using integer operations
        "xor %%r9, %%r9\n\t"
        "mov $10, %%rcx\n\t"
        "1:\n\t"
        "imul %%r9, %%r9\n\t"
        "add $1, %%r9\n\t"
        "dec %%rcx\n\t"
        "jnz 1b\n\t"
        // Using result of dependent instructions, adjust rsp to trick prediction of ret
        "mov %%r9d, %%eax\n\t"          // Use r9 result
        "and $0, %%eax\n\t"             // Always make it 0 for safety
        "lea 8(%%rsp, %%eax, 1), %%rsp\n\t"  // Add 0 to rsp (no change, just use result)
        // ret - Actually returns from speculate_read_byte, but predicted as returning from inner
        "ret\n\t"
        ::: "rax", "r9", "rcx", "memory"
    );
}
```

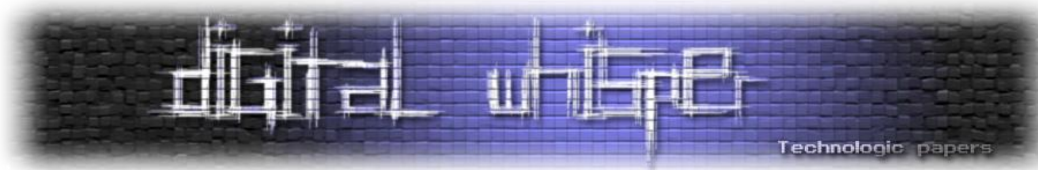
בגרסה זו, הרעיון המרכזי הוא לבלבל את ה-predictor של המעבד בפקודת ה-ret ‏(return): כך שהמעבד יחשוב שהוא חוזר מ-speculate_read_byte_inner אבל בפועל הוא חוזר מ-speculate_read_byte .

איך זה משרת אותנו

רגע לפני החזרה מהפונקציה אנו מזיזים את מצביע המחסנית, כך שהמעבד חוזה שנחזור לכתובת אחרת וימשיך להריץ ממנה בצורה ספקולטיבית. למרות שבפועל נחזור מהפונקציה speculate_read_byte, המעבד (בספקולציה בלבד) יריץ עוד שורות מתוך speculate_read_byte. שורות אלו יבצעו את קוד הקריאה שאנחנו רוצים לעשות בצורה חשאית ומוגנת.

אז מה בעצם נעשה בחלון ההרצה הזה? נקרא את כתובת העניין שלנו בצורה ספקולטיבית (לשם היציבות אציג מימוש שקורא רק בייט אחד בכל פעם).

היות ופקודות אלו לא יורצו בפועל במעבד, לאחר שהמעבד יבין שהספקולציה הייתה לא נכונה (שיגיע להריץ את פקודת ה-ret באמת), המעבד יזנח כל שינוי שנעשה ל-architectural state. עם זאת, שינויים שנעשו ב-cache יישמרו. לכן הדרך הקלה ביותר לשחזר את ערך כתובת העניין היא לבצע Flush + Reload. הרעיון הוא להחזיק מערך, לשם ההסבר, נקרא לו detector, והוא בגודל 4096 * 256. המערך יחזיק page עבור כל אפשרות לערך של הבייט הרלוונטי שכעת אנחנו קוראים.



- למען האמת מספיק להחזיק כמות בתים שמתחלקת בגודל ה-cache line עבור אופציה לכל תו. שזה כמעט תמיד 64 בתים במעבדי אינטל.

בעת האתחול נדאג שכל המערך אינו נמצא ב-cache, נעשה זאת על ידי פקודת האסמבלי clflush אשר דואגת שתוכן כתובת מסויימת לא יימצא באף cache level. הנ"ל ייראה ככה:

```
__attribute__((always_inline))
inline void flush(const void *addr)
{
  asm volatile ("clflush %0" : : "m" (*(volatile unsigned char *)addr));
}
```

נבצע flush לכל ה-buffer:

```
for (int i = 0; i < 256; i++) {
   flush(&detector[i * 4096]);
}
```

לאחר מכן, נבצע קריאה ספקולטיבית כמו שהוסבר קודם ל-[4096 *[offset] detector[target_addr. כעת נבחן את מצב המערך detector ב-cache:

1. עבור ערך אינדקס i שאינו byte_val: [4096 *detector[i אינו ב-cache ולכן גישה אליו תיקח זמן רב יחסית.
2. עבור ערך אינדקס byte_val הגישה ל-[4096 *detector[byte_val תהיה קצרה משמעותית (כי הוא נמצא ב-cache).

וככה נוכל לבדוק את זמני הגישה לאינדקסים שרלוונטיים לכל הערכים האפשריים ולהיון מה ערך הבייט האמיתי.

משהו קטן נוסף: כאשר אנחנו בודקים את זמני הגישה ל-[4096 *detector[i, אם ניגש בצורה סדרתית לערכים, אנו עלולים להיתקל בבעיה: המעבד חכם ויחזה את הערך הבא שנרצה לקרוא החל מאיזשהו שלב ולכן יטען מראש אינדקסים נוספים ב-detector ל-cache, מה שבגדול סוגר לנו את ערוץ ההזלגה. לכן, נרצה לגשת לאינדקסים בצורה מעורבלת.

נבצע פרמוטציה על ערכי האינדקס האפשריים בין 0 ל255 כך שהסדר יהיה אקראי (למעבד) :

```
// Order is lightly mixed up to prevent stride prediction
for (int i = 0; i < 256; i++) {
    int mix_i = ((i * 167) + 13) & 255; // Mix up the order to prevent stride
prediction
    uint64_t timing = timed_read(&dr7_detector[mix_i * 4096]);

    // If timing is below threshold, increment score for this byte value
    if (timing < threshold) {
        scores[mix_i]++;
    pr_info("Round %d: Index %d hit cache (timing = %llu < threshold %llu),
score now = %d\n",
   round, mix_i, timing, threshold, scores[mix_i]);
    }
}
```

- gcd(167,256)=1 ולכן זאת פרמוטציה.

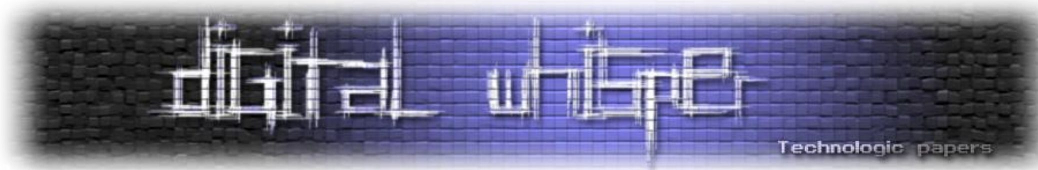
וככה זה נראה:

ראשית נטען את ה-rootkit שלנו (השתמשתי ב-subversive בגרסה שערכתי שמבצעת הוק לקריאה על הכניסה של __x64_sys_read ועורכת את האוגר אליו קוראים את ערך הכניסה ל-0xdeadbeef).

```
root@syzkaller:~# sudo insmod /subversive.ko
sudo: unable[  305.024673] subversive_init: Subversive rootkit initializing...
 to reso[  305.032672] subversive_init: Putting hardware breakpoint on syscall table at address: 0xffffffff82000260
l[  305.033672] subversive_init: Syscall table access protection enabled
v[  305.034672] subversive_init: Enabling debug register protection (DR7_GD)...
e[  305.035672] subversive_init: Subversive rootkit loaded successfully!
```

כעת, נקרא את הטבלה בצורה רגילה, ומייד לאחר מכן בעזרת ההרצה הספקולטבית:

```
root@syzkaller:~# sudo /ctrl read_regular
sudo: unable to[  275.554153] found 'sys_call_table' address: ffffffff82000260
 [  275.555153] handle_protected_address_breakpoint: Protected address accessed from ip=0xffffffffa0000490 (likely module)
resolve host syzkaller: Temporary failure in name resolution
DR-Detector Control Program
===========================

=== READ REGULAR (8 BYTES) ===
Step 1: Discovering syscall table address...
Syscall table discovered at: 0xffffffff82000260
Step 2: Reading 8 bytes from syscall table using regular (non-speculative) access...
Successfully read 8 bytes from syscall table!
Regular read result: 0x00000000deadbeef
root@syzkaller:~# sudo /ctrl read_speculative
sudo: unable t[  440.994002] found 'sys_call_table' address: ffffffff82000260
o[  440.995002] Reading 8 bytes from address ffffffff82000260 using speculative execution
 resolve host syzkaller: Temporary failure in name resolution
DR-Detector Control Program
===========================

=== READ SPECULATIVE (8 BYTES) ===
Step 1: Discovering syscall table address...
Syscall table discovered at: 0xffffffff82000260
Step 2: Reading 8 bytes from syscall table using speculative execution...
Attempting to read 8 bytes from address 0xffffffff82000260 using speculative execution...
IOCTL succeeded!
Result: 0xffffffff811c7ed0
=== READ SPECULATIVE COMPLETED ===
```



נוכל לראות שבקריאה הרגילה אנחנו אכן מקבלים 0xdeadbeef, ועם זאת בעזרת הקריאה הספקולטיבית אנחנו מצליחים לשחזר את הערך האמיתי של ה-syscall שדרסנו בעזרת subversive:

```
root@syzkaller:~# grep ffffffff811c7ed0 /proc/kallsyms
ffffffff811c7ed0 T __x64_sys_read
root@syzkaller:~# 
```

הערה: נציין שהשיטה נבדקה עד כה על שלל דגמי מעבדים וקרנלים (בסביבת לינוקס). מוזמנים לפנות לפירוט.

בעצם הצלחנו לקרוא איזורי עניין במערכת ההפעלה מבלי באמת לקרוא אותם ולהטריג את התוקף. נחמד. בשלב הזה, מי שקרא יפה בטח שואל את עצמו, אבל רגע: שימוש ב-Debug Registers זה רק ווקטור אחד, מה עוד מעניין באותו נושא?

מקרה בוחן נוסף – hypervisor זדוני

רקע

איום נוסף מאותה משפחה, ואף מתקדם יותר אם ממומש נכון, הוא ה-hypervisor הזדוני. בעוד שהטכניקות שסקרנו עד כה פעלו בתוך מערכת ההפעלה, ה-hypervisor פועל מתחתיה, ברמה נמוכה יותר, ומנהל את כלל החומרה הוירטואלית עבור מערכת ההפעלה "האורחת" (Guest). הדוגמה הקלאסית והמפורסמת ביותר ל-hypervisor זדוני היא "Blue Pill", שהוצגה על ידי החוקרת יואנה רטקובסקה בשנת 2006. הרעיון היה להשתמש ביכולות הוירטואליזציה של המעבד (Intel VT-x או AMD-V) כדי "להרים" את מערכת ההפעלה הפעילה ולהפוך אותה למכונה וירטואלית, כל זאת בזמן אמת ומבלי שהמערכת תהיה מודעת לכך. ה-hypervisor הזדוני משיג שליטה, עם יכולת לנטר, לשנות ולחסום כל פעולה בין מערכת ההפעלה לחומרה. קישור לקוד המקורי ניתן למצוא בפרויקט Bluepill.

השימוש בווירטואליזציה למטרות זדוניות פותח בפני התוקף אפשרויות כמעט בלתי מוגבלות:

1. הסתרה מוחלטת: כלי אבטחה הפועלים בתוך מערכת ההפעלה לא יכולים "לראות" את ה-hypervisor שמתחתיהם (אם ממומש נכון – נציין שיש הרבה דברים לתת עליהם את הדעת בתור התוקף)
2. יירוט וניטור פעולות נבחרות: כל גישה לדיסק, לרשת, לזיכרון או לכל רכיב חומרה אחר עוברת דרך ה-hypervisor ויכולה להיות מנוטרת או אף להיערך בזמן אמת.
3. עקיפת מנגנוני אבטחה: Hypervisors זדוניים יכולים להסתיר קריאה מכתובות זיכרון מסוימות או לשנות את התוצאות של פקודות מערכת כדי להתמודד עם מוצרי EDR ומנגנוני הגנה מתקדמים. הם יכולים, למשל, להציג תמונה "נקייה" של הזיכרון לכלי סריקה, בעוד שבפועל רכיבים זדוניים רצים בחופשיות.

שוב, במאמר זה לא ניתן את כל הרקע הנדרש לעיסוק בנושאים אלו, אפנה למאמרים במגזין וברשת בנושא. דוגמה מצוינת לשימוש בטכניקות כאלה ניתן למצוא במאמר הזה מהמגזין, שם אמיר מפרט כיצד ניתן להשתמש ב-hypervisor כדי להסתיר רכיבי חומרה ממערכת ההפעלה האורחת.

בואו נשתמש באותה טכנולוגיה ממקודם כדי לזהות hypervisors

לאחר שראינו את הכוח של הרצה ספקולטיבית, עולה שאלה מעניינת: מה יקרה אם ננסה להשתמש בטכניקה הזו מתוך סביבה שאולי כבר נמצאת תחת hypervisor? האם נוכל להשתמש ב-Spectre Gadgets כדי לזהות את נוכחותו של ה-hypervisor עצמו?

התשובה טמונה בהבנת האינטראקציה בין המכונה הווירטואלית ל-hypervisor. פעולות מסוימות, הנחשבות "רגישות", אינן יכולות להתבצע ישירות על ידי מערכת ההפעלה האורחת וגורמות ל"יציאה" מה-Guest הנקראת vmexit. במצב זה, השליטה עוברת מה-Guest ל- Hypervisor, אשר מטפל בפקודה הרגישה ומחזיר את השליטה.

ומה יקרה אם במהלך הרצה ספקולטיבית נגרום למעבד להריץ פקודה שגורמת ל-vmexit? כאן הדברים הופכים למעניינים. פקודה קלאסית כזו היא RDTSC (Read Time-Stamp Counter), אשר קוראת את מונה מחזורי השעון הפנימי של המעבד ומשמשת למדידת זמנים מדויקת ב-hypervisors רבים, הרצת RDTSC גורמת ל-vmexit כדי שה-hypervisor יוכל לספק ערך זמן עקבי למכונה הווירטואלית.

וכפי שניתן לראות במנואל של אינטל, ניתן לבצע אינטרספציה של הפקודה rdtsc על ידי שינוי הביט ה-12 ב-msr IA32_VMX_PROCBASED_CTLS:

Table 26-6. Definitions of Primary Processor-Based VM-Execution Controls (Contd.)

Bit Position(s)	Name	Description
12	RDTSC exiting	This control determines whether executions of RDTSC and RDTSCP cause VM exits.

תהליך המחשבה שלי היה כזה:

אין שום היגיון בלהריץ ספקולטיבית פקודות אחרי-vmexit כי הטיפול בו לוקח זמן רב (יחסית), וגם וסביר שה-context של ה-Guest ישתנה כתוצאה מה-vmexit לערך שלא ניתן לחזות מראש. לכן הנחתי שבמקרה של hypervisor הספקולציה תיעצר במקרה שלrdtsc, מה שלא יקרה על חומרה פיזית. נוכל לזהות באופן מהימן אם אנחנו רצים בתוך מכונה וירטואלית, ובכך לחשוף את נוכחותו של hypervisor, בין אם הוא לגיטימי או זדוני. בינתיים הנ"ל עובד בצורה מהימנה על כל מעבד אינטל שבדקתי. אציין שלא הצלחתי למצוא תיעוד רשמי של אינטל בנושא, אז אם מישהו מכיר אשמח לקבל הפנייה.

נייש את אותה הטכניקה במקרה הזה של הרצה ספקולטיבית, אבל כעת במקום להחזיק מערך ולנחש את הערכים נרצה רק לדעת אם rdtsc עצר את הספקולציה או לא (מה שיעיד על מצב הווירטואליזציה).

אלא לראותם בלבד: Mitigating (semi) new class of advanced threats

הדרך לעשות את זה היא פשוט לעשות flush + reload אחרי פקודת ה-rdtsc למשתנה בודד.

הערה: בגלל שrdtsc אינה פקודה מיוחסת (privilleged), נוכל להריץ אותה ברמת ה-userspace ואפילו לא נצטרך לעשות את זה מהקרנל, נחמד.

הערה נוספת: ניתן לבחור כל פקודה שאנו חושדים שתעשה vmexit ואינה serializing (עוצרת את הספקולציה בכל מקרה).

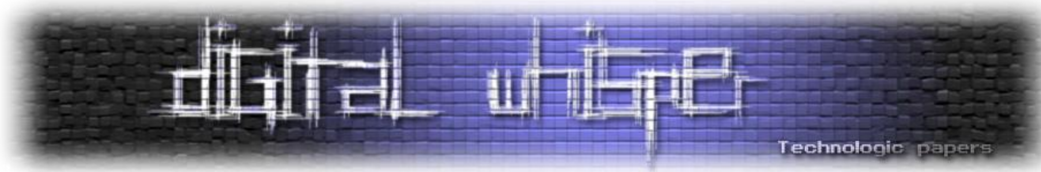
ובכן, למימוש בקוד (פה הלכתי על קובץ אסמבלר שלם שקמפלתי בעזרת nasm):

```
speculate:
  mfence
  call inner
  # Speculatively executed (because of inner's ret being mispredicted):
  rdtsc
  and eax, 0
  movzx eax, byte ptr [rdi+rax]

inner:
  # Lots of slow dependent instructions using integer operations:
  xor r9, r9
  mov rcx, 10
loop_start:
  imul r9, r9
  add r9, 1
  dec rcx
  jnz loop_start
  # Using result of dependent instructions, adjust rsp to trick prediction of
ret:
  mov eax, r9d
  and eax, 0
  lea rsp, [rsp+rax+8]
  ret # Actually returns from speculate, but predicted as returning from inner
```

כאן הפונקציה speculate תקבל פרמטר של כתובת הזיכרון detector שאינה נמצאת ב-cahce. אם מדובר בריצה בתוך hypervisor אזי הגישה ל-detector לא תתבצע וזמן הגישה אליו יהיה רב גם בשלב המדידה, אחרת הגישה תתבצע ושנבדוק שוב נשלוף אותו מה-cache.

ככה זה נראה במכונה פיזית:

```
research@ubuntu-playground:~$ ./checkvirt
Result: Not virtualized (more than 1% successful)
256 out of 256 rounds were successful (cache hit below threshold after speculation)
research@ubuntu-playground:~$
```



וככה זה נראה במכונה וירטואלית (במקרה הזה כזאת שרצה בתוך KVM):

```
root@syzkaller:~# sudo /checkvirt
sudo: unable to resolve host syzkaller: Temporary failure in name resolution
Result: Virtualized (1% or fewer successful)
0 out of 256 rounds were successful (cache hit below threshold after speculation)
root@syzkaller:~#
```

הוק במקרה של קריאת זיכרון

היינו מניחים כי ה-hypervisor הזדוני ירצה להזריק בשלב מסוים קוד לקונטקסט של מערכת ההפעלה, כי הרי משם משמעותית פשוט יותר לעשות דברים מעניינים בקונטקסט של תוקף (ריגול, איסוף מודיעין וכו'). כמו כן, הוא ירצה "להגן" על הקוד הזה מפני קריאה ומוצרי אבטחה הרצים ברמת מערכת ההפעלה, ולכן ירצה שמצד אחד מערכת ההפעלה של ה-Guest תוכל להריץ את אותו קוד אבל אם תנסה לקרוא אותו תיכשל.

לשמחתו הרבה, ניתן לממש את זה עם מנגנון EPT המוצע במעבדים מודרניים. לא ארחיב יותר מדי על EPT היות והוא הוסבר כבר במגזין זה בצורה מעולה במאמר הזה (למרות שהוא נכתב בסיבת ווינדוס – נסלח לו).

בווינדוס, מנגנון אחד שכדאי לזייף אל מולו אם לא רוצים להקריס את מערכת ההפעלה הוא ה-PatchGuard אשר בין היתר מאמת איזורי זיכרון קרנליים אל מול ה-image בדיסק ומתריע במקרה של שוני. במקרה שלנו, מדובר על כלי איתור ו-EDR אשר קוראים איזורים קרנליים על מנת לחפש אנומליות ושונות (בין אם על ידי השוואה עם ה-image של הקרנל מהדיסק או בין עם על ידי שיטות אחרות).

בקצרה, בטבלת EPT בניגוד לטבלת דפים רגילה, Page יכול להיות בר הרצה ואינו קריא. הרעיון הוא לגרום ל-EPT VIOLATION במקרה של קריאת אותה הכתובת והזרקת הערך "הנכון" ל-Guest, למרות שבפועל הערך המצוי בזיכרון הוא זדוני. הקוד עצמו יכול לעשות אחד משני דברים:

1. להתנהג כמו כל rootkit קרנלי ופשוט להיות מוסתר על ידי EPT.
2. לבצע hook מינימאלי (וקטן בהרבה) שפשוט יתריג EPT VIOLATION ויעביר שליטה ל-hypervisor שם תתבצע הלוגיקה הזדונית.

- למשל על ידי הצבת האופקוד 0xcc שיגרום ל-bp, והגדרה של Debug Exception ליצור Vm-Exit בעזרת הגדרה ב-Exception Bitmap של ה-hypervisor.

אז מה עושים כמגינים

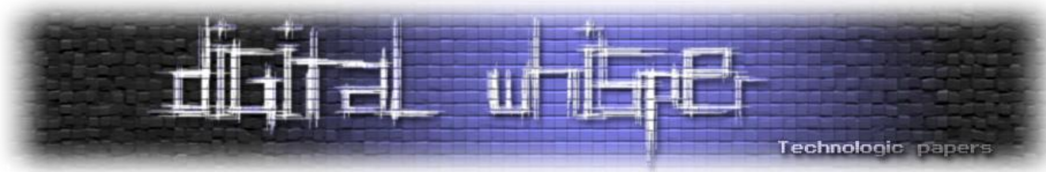
אני רק אפתח את התיאבון ולא אגלה הכל... נניח לרגע שיש לנו כתובת עניין שאנו חושדים בה שהיא מוסתרת (או שנחזיק סט בגודל לא גדול מדי של כתובות מעניינות במיוחד לתוקף ונבדוק את כולן... במקרה של כלי אוטומטי). מה יקרה ל-cache במקרה של vmexit, התערבות ולבסוף vmenter? ישנן כמה אופציות:

- ה-hypervisor יבצע איפוס ל-cache (או חלק מהרמות) בעת פקודת vmenter. הנ"ל קורה ב-hypervisors לאחר גילוי סט החולשות הנקרא Foreshadow ו-Foreshadow-ng בשנת 2018, אשר מאפשרות בין היתר קריאה של דפים מהקונטקסט של ה-hypervisor או של Guests אחרים. לא ארחיב יותר מדי אבל הפתרון (דאז – 2018) היה להוסיף אפשרות לאפס את ה-L1 cache בעת vmenter על ידי כתיבה ל-MSR בשם IA32_FLUSH_CMD. נציין שבמעבדים חדשים יש כבר הגנה נגד מתקפות מסוג ברמה החומרתית, ולכן כבר אין צורך ב-flush הנ"ל (שגם פוגם בביצועים בצורה לא זניחה). מבחינת איתור, hypervisor שמאפס את ה-L1D ניתן לאיתור בקלות רבה מאד – כל מה שצריך לעשות זה לבדוק זמני גישה לזיכרון שאמור להיות ב-cache אחרי פקודות חשודות. יצא לי לראות hypervisors זדוניים אשר מומשו על בסיס kvm ונופלים בזה כמעט כל פעם (לשם ההגינות אציין שלרוב הם "נופלים" באנומליות פשוטות אף יותר).
- לאחר החזרה, יהיו שינויים ב-cache של המעבד שיעידו על הרצה של קוד נוסף מעבר לנראה בקונטקסט של כלי האיתור ויעידו על קיום משהו חשוד. בימים אלה אני מקדם מחקר שמטרתו לפתח שיטות גנריות (שלא מתבססות על חולשות ספציפיות) כדי להבין לא רק שהורץ משהו זדוני כתוצאה מפקודה מסויימת אלא לנסות להתחקות אחרי שינויים ב-cache שיעזרו להבין גם מה הורץ. המטרה היא להכניס אותן לכלי האיתור שלנו בגרסאות הבאות.

לסיכום

כלי איתור המבצעים קריאת זיכרון קרנלי כחלק מתהליך האיתור נהיים נפוצים יותר ויותר (מי מאיתנו לא השתמש ב-Volatility?) אל מול כך, ניתן לצפות כי התוקפים יבצעו קפיצת מדרגה שתשמור על חשאיותם אל מול אותן שיטות הגנה. נוסף על כך AI, כנראה יוכל מתישהו לנתח בצורה חכמה את אותם Memory Dumps באופן המוני ולמצוא אנומליות במהירות שטרם הכרנו, מה שמגביר את הצורך העיצומי מצד התוקף, והפתרון הטוב והרובוסטי ביותר מעיניים של תוקף, הוא לא להופיע באותו Memory Dump מלכתחילה...

במאמר סקרנו שני ווקטורים מרכזיים אליהם תוקפים עלולים ללכת ומה אנחנו כמגינים יכולים לעשות על מנת להמשיך ולאתר אותם. וזה בדיוק מה שאנחנו עושים ב-CoreSights, בונים כלים עבור חברות אבטחה ו- EDR המסוגלים לרוץ בקונטקסט של מערכת ההפעלה בצורה המונית כחלק ממוצרים קיימים, אבל לזהות ולהתריע על איומים מתקדמים (גם אם הם נמצאים בשכבות נמוכות יותר ופחות נגישות למוצרי איתור מסחריים) .



המיזם עדיין בתחילת דרכו ואני מחפש שותף עם ראש יזמי ורקע רחב מאד במערכות הפעלה ועולמות התקיפה/איתור קוד עוין .

לפניות ושאלות על המאמר ניתן לפנות ב-X או במייל:

X - @omerhashalev

עמוד המיזם - https://coresights.co/

מייל - omer@coresights.co