Technologic Papers

-

Windows Kernel Fuzzing With Intel PT

מאת אריאל ז.

מבוא

במסגרת המאמר אציג שיטה חדשה לביצוע coverage guided fuzzing בקרנל של Windows תוך שימוש ב-WinAFL כמנוע ה-fuzzing וב-Intel PT כתשתית code trace שמחליפה את תשתית האינסטרומנטציה הסטנדרטית DynamoRIO.

ההנחה היא שהקורא מכיר ברמה סבירה את עולם ה-fuzzing ואת AFL בפרט, ואת עולם הרברסינג ודיבוג בקרנל של Windows. אך ראשית נתחיל בהגדרת מושגים בסיסיים והכרת הכלים איתם נעבוד.

Fuzzing, Coverage guided fuzzing

Fuzzing היא שיטה אוטומטית לגילוי באגים בתכנה במסגרתה מרעישים את הקלט לתכנה במטרה לגלות באגים, קריסות ומקרי קצה לא מטופלים.

ניתן לסווג כלי fuzzing ל-White, Grey, Black box:

- White box fuzzing - מכיר את קוד המקור של התכנית ומשתמש בשיטות ניתוח כמו symbolic execution ו-constraint solving במטרה לייצר קלטים שמגדילים כיסוי קוד או מגיעים לאזורי קוד קריטיים. כלים אלו יכולים להגיע לכיסוי קוד גדול אך דורשים מחיר חישובי גבוה.
- Black box fuzzing - משתמש בהכי פחות מידע על התכנית, מה שמאפשר לו ליצור קלטים באופן מהיר על חשבון העובדה שהם מייצרים הכי פחות כיסוי קוד.
- Grey box fuzzing או Coverage guided fuzzing - משתמש באינסטרומנטציה של ריצת התכנית על מנת למדוד כיסוי קוד שמייצר הקלט. כיסוי הקוד משמש לייצור קלטים שמטרתם להביא לכיסוי קוד מקסימלי. כלים אלו מהווים איזון טוב בין שני הסוגים הקודמים.

Intel PT

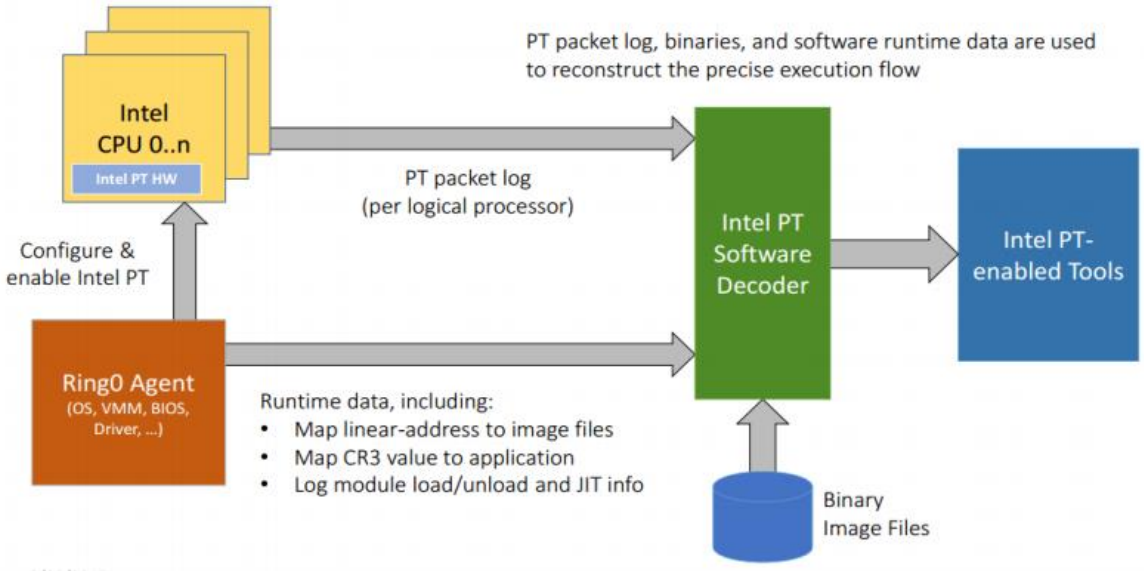
Intel PT (Processor Trace) הוא פיצ'ר חומרתי במעבדי אינטל המאפשר ביצוע tracing על ריצת קוד.

PT מייצר פאקטות מידע על הריצה ברמת ה-hardware thread (ליבה פיסית), מידע זה מאפשר בניה מחדש של רצף הפקודות של התכנית.

מה השימושים של PT?

- דיבוג - gdb ו-windbg משתמשים ב-PT על מנת לאפשר הקלטה והרצה קדימה ואחורה ב-trace התכנתי
- profiling
- performance analyzing - למשל linux perf
- fuzzing

איך PT עובד?



דרייבר שיושב במערכת ההפעלה (Ring0 Agent) מקנפג ומפעיל את PT באמצעות כתיבה וקריאה ל-MSR יעודיים. ה-trace נכתב בצורה מקודדת לזיכרון הפיסי.

ה-Decoder מפענח את ה-trace המקודד באמצעות תמונת זיכרון (image) של מודולים רלוונטים (כגון exe, dll, sys). נחזור לנקודה זו בהמשך המאמר.

אינטל מספקת ספרית decoding בשם libipt:

https://github.com/intel/libipt

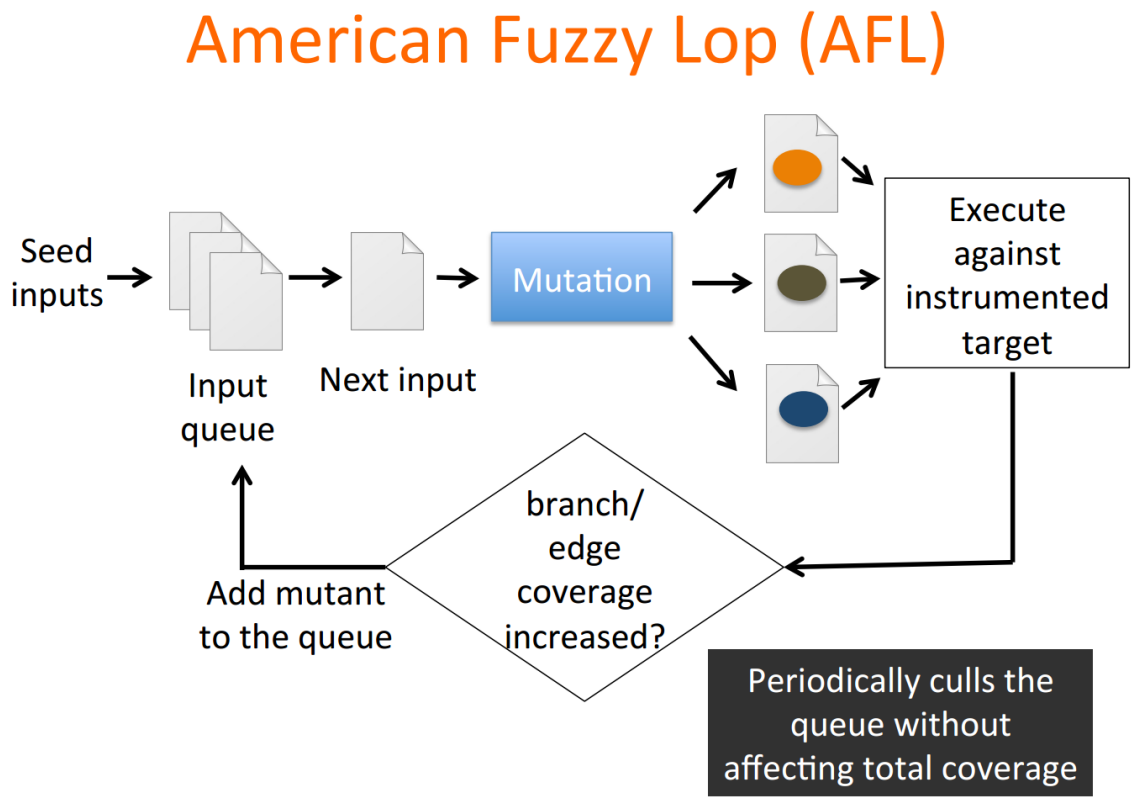
בקצה השרשרת נמצא רכיב היעד (Intel PT-enabled Tool), למשל debugger, profiler או fuzzer שמתממשק לרכיב המפענח ומשתמש ב-trace המפוענח לצרכיו.

WinAFL

WinAFL הנו פורט ל-windows של ה-fuzzer הלינוקסי AFL.

AFL הוא coverage guided fuzzer שמשתמש בהיוריסטיקות הרעשת קלט שהוכחו כיעילות מאד במציאת באגים.

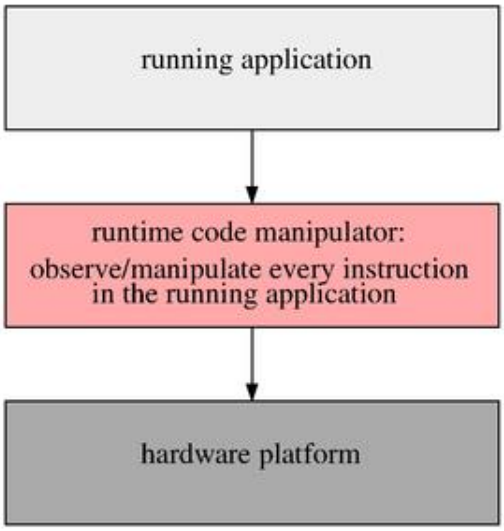
איך AFL עובד?



AFL מרעיש את הקלט של תכנית שרצה עם אינסטרומנטציה ומודד את כיסוי הקוד. קלט שמגדיל את הכיסוי (מביא למסלולים חדשים בקוד) מסווג כ"מעניין", נשמר בצד ומשמש בסיס להמשך הרעשה של הקלט.

איך AFL מבצע אינסטרומנטציה לקוד?

בלינוקס ניתן לקמפל את קוד התכנית עם אינסטרומנטציה מובנית במידה וקוד המקור זמין, או להשתמש ב-QEMU כשמדובר על fuzzing של אפליקציה closed source. ב Windows משתמשים בתשתית האינסטרומנטציה DynamoRIO. איך היא עובדת?



DynamoRIO רץ כשכבה חוצצת בין אפליקצית היעד למערכת ההפעלה, יש לה שליטה מלאה על ריצת האפליקציה וכזאת יודעת לנתח ולספק מידע על רצף הקוד של התכנית.

הבעיה באינסטרומנטציה היא הפגיעה בביצועים. לפי התיעוד QEMU ו-DynamoRIO מביאים לפגיעה של לפחות פי 2 בזמן הריצה של התכנית.

לאחרונה נוסף ל-WinAFL מוד trace חומרתי המבוסס על Intel PT שמחליף את השימוש ב-DynamoRIO.

יש לציין שמכיוון שה trace מבוצע בחומרה נצפה לקבל שיפור בביצועים אבל יש לקחת בחשבון ששלב הפענוח של ה-trace המקודד הנו כבד יחסית. כיום לפי הבנתי הביצועים פחות או יותר דומים ל-DynamoRIO אבל יש עוד מקום לאופטמיזיציה של שלב הפענוח.

שאלת המחקר

WinAFL הוא user mode fuzzer אבל PT מאפשר ביצוע trace של כל הקוד הרץ על המעבד[1] ובפרט הקוד של מערכת ההפעלה, לכן השאלה המתבקשת היא **האם ניתן להרחיב את WinAFL לתמיכה ב-kernel mode fuzzing ב-Windows**.

כלים קיימים

לפני שנתחיל לבחון את הבעיה, נבדוק אילו כלי coverage guided fuzzing קיימים כיום ל-Windows kernel והאם ניתן להשתמש בהם.

אז כיום ישנם שני כלים, שניהם משתמשים ב-AFL כמנוע ה-fuzzing וב-PT כתשתית code trace.

הכלים אמנם פומביים אבל לא ניתן להשתמש בהם כמוצר מדף מכיוון שהכותבים לא פרסמו גרסה מלאה שלהם.

1. WinAFL IntelPT (TALOS)

הוצג בכנס BH 2017. זהו ה-coverage guided fuzzer הפומבי הראשון ל-windows kernel. הכותב לא פרסם את רכיב הממשק בין PT ל-AFL לכן אין למעשה גרסה מלאה של הכלי. כמו כן הכלי משתמש בדרייבר custom ל-PT אבל הוא לא עובד בגרסאות חדשות של WIN10 בגלל עדכונים ב-patchguard.

https://github.com/intelpt/winafl-intelpt

2. kAFL

הוצג בכנס usenix 17. זהו כלי לינוקסי המשתמש ב-KVM Hypervisor על מנת לבצע fuzzing עם AFL בקרנל של מ"ה שרצה ב-VM. הכלי תומך ב-fuzzing של מ"ה Linux, Windows, MacOS אבל ה-Agents של Windows ו-MacOS לא שוחררו פומבית.

https://github.com/RUB-SysSec/kAFL

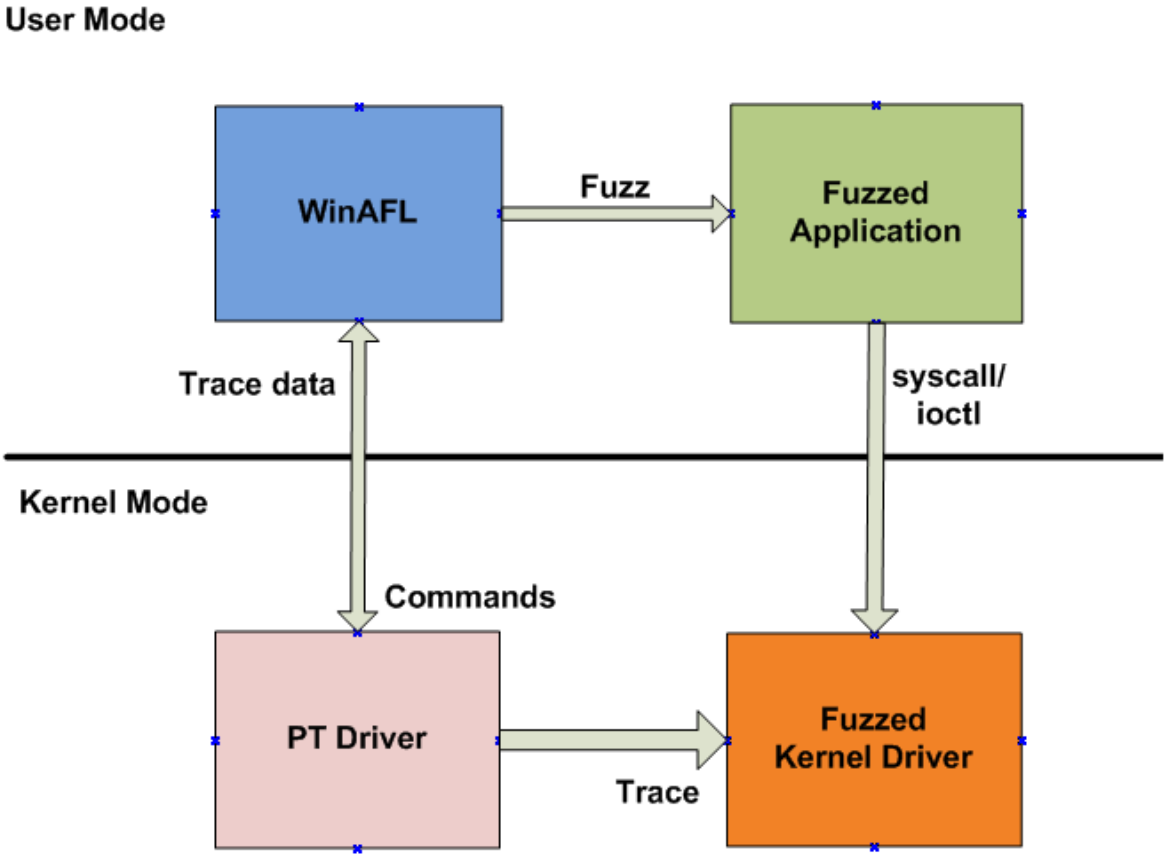
[1] למעשה ישנם איזורים עליהם לא ניתן לבצע trace: קוד SGX, וקוד SMM אלא אם עושים opt in ברמת ה-BIOS

מוטיבציה

לסיכום פרק המבוא, הרעיון הוא לכתוב אפליקציה שמדברת עם דרייבר ומטריגה בו פונקציונליות עליה רוצים לבצע fuzzing. WinAFL ירעיש את הקלטים לאפליקציה תוך מדידת כיסוי קוד קרנלי עם PT.

כלומר אנו למעשה מבצעים fuzzing על קוד הדרייבר כאשר האפליקציה משמשת כצינור להעברת הקלט.

הרעיון מתואר בדיאגרמה הבאה:



השמשה

השמשת WinAFL לעבודה עם PT

ראשית יש להוריד את הקוד של WinAFL ולקמפל אותו עם דגל INTELPT=1:

https://github.com/googleprojectzero/winafl

כדי לבצע fuzzing מבוסס PT במקום DynamoRIO יש להשתמש בפרמטר שורת פקודה -P במקום -D.

מומלץ גם לקרוא את ה-readme על השילוב של PT ב-WinAFL:

https://github.com/googleprojectzero/winafl/blob/master/readme_pt.md

PT ב-Windows

על מנת להבין את השינויים שנבצע בקוד יש להכיר את הדרייבר של PT ב-Windows.

הדרייבר Ipt.sys אחראי לממשק בין רכיבי user mode שזקוקים לשירותים של PT, כמו ETW או WinDbg, למנגנון PT החומרתי. הדרייבר חושף ממשק ioctl סטנדרטי לבקשות משתמש:

```
            return 0xC000000D;
          v38 = IptQueryCoreTrace(&v41);
        }
        else
        {
          v38 = IptQueryProcessTrace(*(a1 + 8), &v41);
        }
        v10 = v38;
        *v9 = v41;
        return v10;
      }
      v39 = IptResumeThreadTrace(*(a1 + 8), &v47, a3, v8);
    }
    else
    {
      v39 = IptPauseThreadTrace(*(a1 + 8), &v47);
    }
    v10 = v39;
    *v9 = v47;
    return v10;
  }
  v33 = IptStopProcessTrace(*(a1 + 8));
LABEL_62:
  v10 = v33;
  goto LABEL_63;
}
if ( v6 == 5 )
{
  v33 = IptStartProcessTrace(*(a1 + 8), *(a1 + 16));
  goto LABEL_62;
```

אפשור trace קרנלי

בקוד של WinAFL נבחן את הפונקציה run_target_pt שתפקידה להריץ בצורה מחזורית את ה- target function (הפונקציה עליה מבצעים את ה-fuzzing):

```
// start tracing
if ((!options.persistent_trace) || (fuzz_iterations_current == 0))
{
    IPT_OPTIONS ipt_options;

    memset(&ipt_options, 0, sizeof(IPT_OPTIONS));

    ipt_options.OptionVersion = 1;

    ConfigureBufferSize(options.trace_buffer_size, &ipt_options);

    ConfigureTraceFlags(0, &ipt_options);

    if (!StartProcessIptTracing(child_handle, ipt_options))
    {
        FATAL("ipt tracing error\n");
    }

    last_ring_buffer_offset = 0;
}
```

הפונקציה ConfigureTraceFlags מקנפגת את הדגלים השונים בבקשת ה-trace, למשל האם ה-trace הוא של קוד user או קרנל. הפרמטר הראשון שערכו 0 מסמן בקשת trace של קוד user.

נבחן דגלים נוספים:

```
#define IPT_TOOL_USE_MTC_TIMING_PACKETS     0x01
#define IPT_TOOL_USE_CYC_TIMING_PACKETS     0x02
#define IPT_TOOL_TRACE_KERNEL_MODE          0x04
#define IPT_TOOL_TRACE_ALL_MODE             0x08
```

שני הדגלים הראשונים מקנפגים אילו פאקטות מידע יש לכלול ב-trace.

הדגל IPT_TOOL_TRACE_KERNEL_MODE מסמן בקשת trace קוד קרנל בלבד.

הדגל IPT_TOOL_TRACE_ALL_MODE מסמן בקשת trace של קוד user וקרנל יחד.

אם כן, עלינו לשנות את בקשת הקנפוג ל-trace קרנלי:

```
// start tracing
if ((!options.persistent_trace) || (fuzz_iterations_current == 0))
{
    IPT_OPTIONS ipt_options;

    memset(&ipt_options, 0, sizeof(IPT_OPTIONS));

    ipt_options.OptionVersion = 1;

    ConfigureBufferSize(options.trace_buffer_size, &ipt_options);

    //ConfigureTraceFlags(0, &ipt_options);
    ConfigureTraceFlags(4, &ipt_options);

    if (!StartProcessIptTracing(child_handle, ipt_options))
    {
        FATAL("ipt tracing error\n");
    }

    last_ring_buffer_offset = 0;
}
```

לא השתמשנו בשם הדגל IPT_TOOL_TRACE_KERNEL_MODE מכיוון שהוא לא מוכר ב-scope באותה נקודה, כך WinAFL בנוי כרגע.

ננסה להריץ את WinAFL לאחר השינוי על תכנית לדוגמא:

```
Command Prompt
C:\Shared\winafl_intelpt\winafl-master\build64\bin\Release>afl-fuzz.exe -i in -o out -P -t 20000 -- -fuzz_i
terations 5000 -target_module harness.exe -target_offset 0x1390 -nargs 2 -nopersistent_trace -- harness.exe
 @@
WinAFL 1.16b by <ifratric@google.com>
Based on AFL 2.43b by <lcamtuf@google.com>
IPT service enebled
[+] You have 4 CPU cores and 0 runnable tasks (utilization: 0%).
[+] Try parallel jobs - see docs\parallel_fuzzing.txt.
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[+] Process affinity is set to 1.

[*] Setting up output directories...
[+] Output directory exists but deemed OK to reuse.
[*] Deleting old session data...
[+] Output dir cleanup successful.
[*] Scanning 'in'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Attempting dry run with 'id_000000'...

[-] PROGRAM ABORT : ipt tracing error

        Location : run_target_pt(), C:\Shared\winafl_intelpt\winafl-master\winaflpt.c:1503


C:\Shared\winafl_intelpt\winafl-master\build64\bin\Release>
```

קיבלנו שגיאה "ipt tracing error".

ניתן לראות בקוד שצירפנו לעיל שהשגיאה נובעת מכישלון בפונקציה StartProcessIptTracing שמבקשת מהדרייבר להתחיל להקליט trace.

נבחן את קוד הפונקציה:

```
BOOL
StartProcessIptTracing (
    _In_ HANDLE hProcess,
    _In_ IPT_OPTIONS ullOptions
    )
{
    BOOL bRes;
    HANDLE hIpt;
    IPT_INPUT_BUFFER inputBuffer;
    IPT_OUTPUT_BUFFER outputBuffer;

    //
    // Open the IPT Device
    //
    bRes = OpenIptDevice(&hIpt);
    if (bRes != FALSE)
    {
        //
        // Send a request to start tracing for this process
        //
        InitializeIptBuffer(&inputBuffer, IptStartProcessTrace);
        inputBuffer.StartProcessIptTrace.Options = ullOptions;
        inputBuffer.StartProcessIptTrace.ProcessHandle = hProcess;
        bRes = DeviceIoControl(hIpt,
                               IOCTL_IPT_REQUEST,
                               &inputBuffer,
                               sizeof(inputBuffer),
                               &outputBuffer,
                               sizeof(outputBuffer),
                               NULL,
                               NULL);
        CloseHandle(hIpt);
    }
    return bRes;
}
```

הפונקציה נכשלת בבקשת ה-DeviceIoControl לדרייבר. על מנת להבין את הסיבה לשגיאה עלינו לדבג את הדרייבר בקרנל.

בסופו של דבר בקשת ה-ioctl מגיעה לפונקציה IptStartProcessTrace בדרייבר:

```
__int64 __fastcall IptStartProcessTrace(__int64 a1, __int64 a2)
{
  void *process_handle; // r14@1
  __int64 v3; // rbx@1
  __int64 result; // rax@3
  unsigned int v5; // er15@4
  __int64 pid; // rsi@4
  signed __int64 v7; // rax@6
  __int64 _process_context; // rdi@6
  volatile signed __int32 *process_context; // rax@13
  PVOID eprocess; // [rsp+50h] [rbp+18h]@3

  process_handle = a1;
  v3 = a2;
  if ( !CheckIptOption(a2) || v3 & 0xF000000 )
    return 0xC000000Di64;
  result = GetProcessFromHandle(process_handle, &eprocess);
  if ( result < 0 )
    return result;
  v5 = 0xC0000001;
  pid = PsGetProcessId(eprocess);
  KeEnterCriticalRegion();
  ExAcquirePushLockSharedEx(&push_lock, 0i64);
```

ניתן לראות כי יש בדיקות שמפלטרות בקשות trace מסוימות.

מה בדיוק מפולטר?

```
Disassembly
Offset: ipt!IptStartProcessTrace
fffff805`4b59932c 48895c2408      mov     qword ptr [rsp+8],rbx
fffff805`4b599331 48896c2410      mov     qword ptr [rsp+10h],rbp
fffff805`4b599336 4889742420      mov     qword ptr [rsp+20h],rsi
fffff805`4b59933b 57              push    rdi
fffff805`4b59933c 4156            push    r14
fffff805`4b59933e 4157            push    r15
fffff805`4b599340 4883ec20        sub     rsp,20h
fffff805`4b599344 4c8bf1          mov     r14,rcx
fffff805`4b599347 488bda          mov     rbx,rdx
fffff805`4b59934a 488bca          mov     rcx,rdx
fffff805`4b59934d e8d67effff      call    ipt!CheckIptOption (fffff805`4b591228)
fffff805`4b599352 84c0            test    al,al
fffff805`4b599354 0f8455010000    je      ipt!IptStartProcessTrace+0x183 (fffff805`4b5994af)
fffff805`4b59935a f7c30000000f    test    ebx,0F000000h
fffff805`4b599360 0f8549010000    jne     ipt!IptStartProcessTrace+0x183 (fffff805`4b5994af)
fffff805`4b599366 488d542450      lea     rdx,[rsp+50h]
```

ניתן לראות כי את בדיקת CheckIptOption עוברים בהצלחה:

Registers

Customize...

Reg	Value
rax	1000001
rcx	15
rdx	0
rbx	1090001

הבדיקה השניה מוודאת שביטים 24-27 ב-ebx מאופסים, אבל ניתן לראות שביט 24 דולק לכן נכשלים בבדיקה והבקשה נדחית.

מהו אותו ערך 0x1090001 שמגיע לדרייבר? זהו מבנה IPT_OPTIONS של הדגלים המועברים לדרייבר בבקשת הקלטת trace:

```
typedef union _IPT_OPTIONS
{
    struct
    {
        DWORDLONG OptionVersion : 4;    // Must be set to 1
        DWORDLONG TimingSettings : 4;   // IPT_TIMING_SETTINGS

        DWORDLONG MtcFrequency : 4;     // Bits 14:17 in IA32_RTIT_CTL
        DWORDLONG CycThreshold : 4;     // Bits 19:22 in IA32_RTIT_CTL

        DWORDLONG TopaPagesPow2 : 4;    // Size of buffer in ToPA, as 4KB powers of 2 (4KB->128MB)
        DWORDLONG MatchSettings: 3;     // IPT_MATCH_SETTINGS
        DWORDLONG Inherit : 1;          // Children will be automatically added to the trace

        DWORDLONG ModeSettings : 4;     // IPT_MODE_SETTINGS
        DWORDLONG Reserved : 36;
    };
    DWORDLONG AsULonglong;
} IPT_OPTIONS, *PIPT_OPTIONS;
```

בפונקציה ConfigureTraceFlags שהצגנו לעיל מקנפגים את הדגלים:

```
//
// Choose the right mode setting
//
if (dwFlags & IPT_TOOL_TRACE_KERNEL_MODE)
{
    pOptions->ModeSettings = IptCtlKernelModeOnly;
}
else if (dwFlags & IPT_TOOL_TRACE_ALL_MODE)
{
    pOptions->ModeSettings = IptCtlUserAndKernelMode;
}
else
{
    pOptions->ModeSettings = IptCtlUserModeOnly;
}
```

(enum _IPT_MODE_SETTINGS)IptCtlKernelModeOnly = 1
Sets BranchEn[2000], ToPA[100], OS[4]

ניתן לראות שבבקשת trace קרנלי נכתב הערך 1 לשדה ModeSettings שמאכלס את ביטים 24-27 כלומר מדליק את ביט 24 וזה מה שמכשיל את הבדיקה שכן ראינו שביטים אלו חייבים להיות מאופסים. כזכור ראינו שבקשת user mode trace כותבת ערך 0 לשדה זה.

כלומר למעשה אנחנו רואים **שהדרייבר של מיקרוסופט לא מאפשר ביצוע trace קרנלי**. סיבה אפשרית להגבלה היא הגנה מפני מעקף KASLR. כידוע החל מ-WIN 8.1 תהליכים שרצים ב-sandbox/low il אינם יכולים לקרוא את כתובות הבסיס של מודולים בקרנל כדי להקשות על השמשת חולשות קרנל. Trace קרנלי עלול לשמש כמעקף.

מניסיוני בגרסת Win10 RS5 תהליך שרץ ב-low il אכן יכול לפנות לדרייבר PT ולהקליט ולקרוא trace.

על מנת לעקוף את ההגבלה הזו עלינו לפצ'פצ' את קוד הבדיקה בדרייבר:

```
fffff805`4b59934a 488bca          mov     rcx,rdx
fffff805`4b59934d e8d67effff      call    ipt!CheckIptOption (fffff805`4b591228)
fffff805`4b599352 84c0            test    al,al
fffff805`4b599354 0f8455010000    je      ipt!IptStartProcessTrace+0x183 (fffff805`4b5994af)
fffff805`4b59935a f7c30000000f    test    ebx,0F000000h
fffff805`4b599360 90              nop
fffff805`4b599361 90              nop
fffff805`4b599362 90              nop
fffff805`4b599363 90              nop
fffff805`4b599364 90              nop
fffff805`4b599365 90              nop
fffff805`4b599366 488d542450      lea     rdx,[rsp+50h]
fffff805`4b59936b 498bce          mov     rcx,r14
fffff805`4b59936e e84dffffff      call    ipt!GetProcessFromHandle (fffff805`4b5992c0)
```

יצוין כי אותה הגבלה קיימת גם בפונקציה המחזירה את ה-trace לקורא (IptGetProcessTrace) לכן יש לפצ'פצ' גם את קוד הפונקציה הזו:

```
v8 = 0xC0000001;
KeEnterCriticalRegion();
ExAcquirePushLockSharedEx(&push_lock, 0i64);
if ( !byte_1C0005381 && !byte_1C0005380 )
{
  v9 = PsGetProcessId(Object);
  v10 = IptLookupProcess(v9);
  v11 = v10;
  if ( v10 )
  {
    // !!! kernel trace flag check !!!
    if ( !(*(v10 + 48) & 0xF000000) && 2 == _InterlockedCompareExchange((v10 + 64), 3, 2) )
    {
      RunOnEachThread(*(v10 + 40), GetTrace, &v15);
      v12 = a5;
      v13 = v18;
      v14 = v17;
      *v5 = v18;
      if ( v12 )
        *v12 = v14;
      if ( v15 && v13 > v14 )
      {
        v8 = 0xC0000023;
        if ( v14 )
          v8 = 0x80000005;
```

נציין בנקודה זו שבאופן כללי פיצ'פוצ' קוד של מודולים קרנליים מנוטר ע"י Patchguard ועלול לגרום ל-BSOD, לכן ההמלצה היא לרוץ במוד Debug שבו Patchguard מנוטרל. מניסיוני VMWare/VBox עדיין לא מבצעים אמולציה של PT לכן כרגע יש לדבג מכונה פיסית.

לאחר פיצ'פוצ'י הקוד נוסיף הדפסות debug בפונקציה analyze_trace_full_reference שתפקידה לפענח את ה-trace ולעדכן את כיסוי הקוד, וננסה להריץ שוב את ה-fuzzer:

```
Command Prompt
[+] Try parallel jobs - see docs\parallel_fuzzing.txt.
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[+] Process affinity is set to 1.

[*] Setting up output directories...
[+] Output directory exists but deemed OK to reuse.
[*] Deleting old session data...
[+] Output dir cleanup successful.
[*] Scanning 'in'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Attempting dry run with 'id_000000'...
decoder->ip: FFFFF8060DCF8E5D status: -13
decoder->ip: FFFFF80601863214 status: -13
decoder->ip: FFFFF809DFC024F2 status: -13
decoder->ip: FFFFF80601900D05 status: -13
decoder->ip: FFFFF806018C0957 status: -13
decoder->ip: FFFFF809DFC0CA84 status: -13
decoder->ip: FFFFF8060DA5D220 status: -13
decoder->ip: FFFFF806019B9232 status: -13

[-] PROGRAM ABORT : No instrumentation detected
         Location : perform_dry_run(), C:\Shared\winafl_intelpt\winafl-master\afl-fuzz.c:3047

C:\Shared\winafl_intelpt\winafl-master\build64\bin\Release>
```

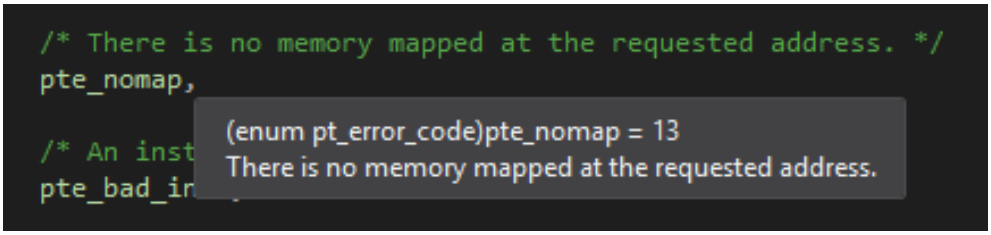
ניתן לראות שהצלחנו לקבל trace קרנלי - הכתובות ...0xFFFFF80.

מצוין, סיימנו את החלק הזה.

הוספת תמונת זיכרון ל-Decoder

ניתן לראות בצילום המסך לעיל שכעת מקבלים שגיאה אחרת "No instrumentation detected" והפונקציה pt_blk_next שתפקידה להחזיר את ה-basic block הבא ב-traceמחזירה שגיאה 13-.

נחפש את השגיאה 13- בקוד:



משמעות השגיאה pte_nomap היא שה decoder לא מצא תמונת זיכרון (image) שמתאימה לכתובות שהתקבלו ב-trace.

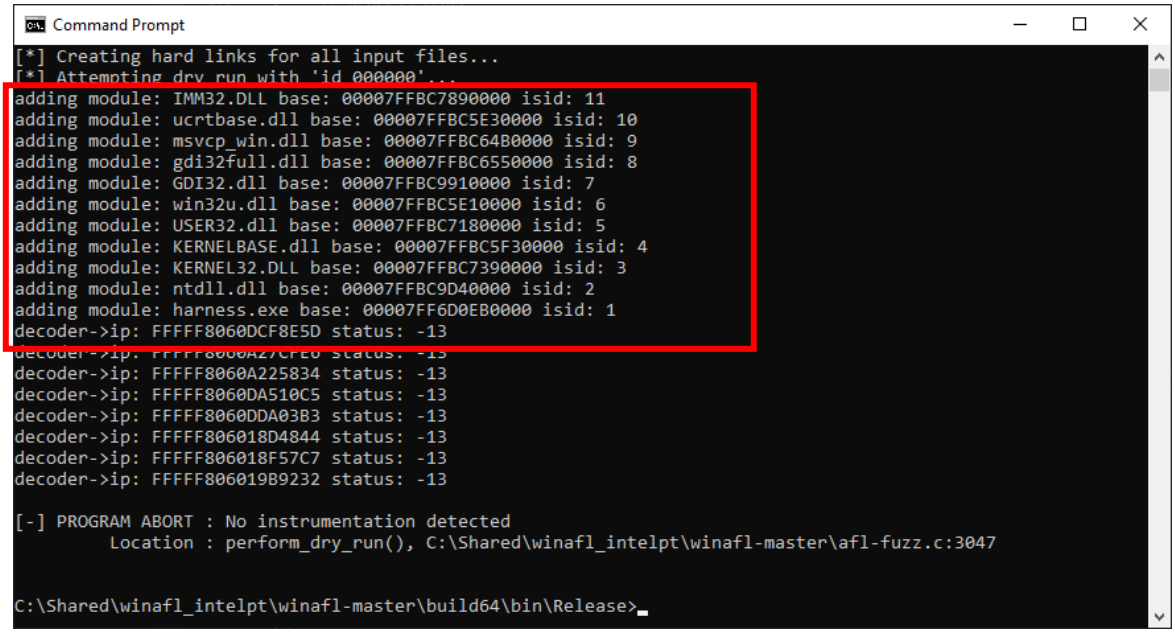
למה בעצם ה-decoder זקוק ל-image? מכיוון שה trace הגולמי אינו מכיל פקודות מעבד אלא רק "meta data" על ריצת הקוד כמו יעדי branch או האם branch נלקח או לא. ה-decoder משתמש ב-image על מנת לפענח את פקודות ה-assembly בכתובות שמתקבלות ב-trace, על מנת למשל להרכיב את כתובת הפקודה הבאה לביצוע או להרכיב basic blocks ברצף הקוד.

איך WinAFL מתמודד עם הדרישה הזו? הוא מפרסר ובונה לעצמו רשימה של המודולים הטעונים במרחב ה-user mode של התכנית.

בקטע הקוד הבא בפונקציה run_target_pt ניתן לראות שרצים על רשימת all_modules שמחזיקה descriptors של המודולים ב-user mode ומוסיפים אותם למבנה image של ה-decoder של אינטל. ה-descriptor מכיל את שם המודול, כתובת בסיס, גודל ואת ה-image של המודול הטעון.

```
struct pt_image *image = NULL;
if ((options.decoder == DECODER_FULL_FAST) || (options.decoder == DECODER_FULL_REFERENCE))
{
    image = pt_image_alloc("winafl_image");
    module_info_t *cur_module = all_modules;

    while (cur_module)
    {
        if (cur_module->isid > 0)
        {
            int ret = pt_image_add_cached(image, section_cache, cur_module->isid, NULL);
        }

        cur_module = cur_module->next;
    }
}
```

נוסיף הדפסות debug לקטע הקוד:



ניתן לראות שאכן כל ה-user mode modules נוספו ל-decoder.

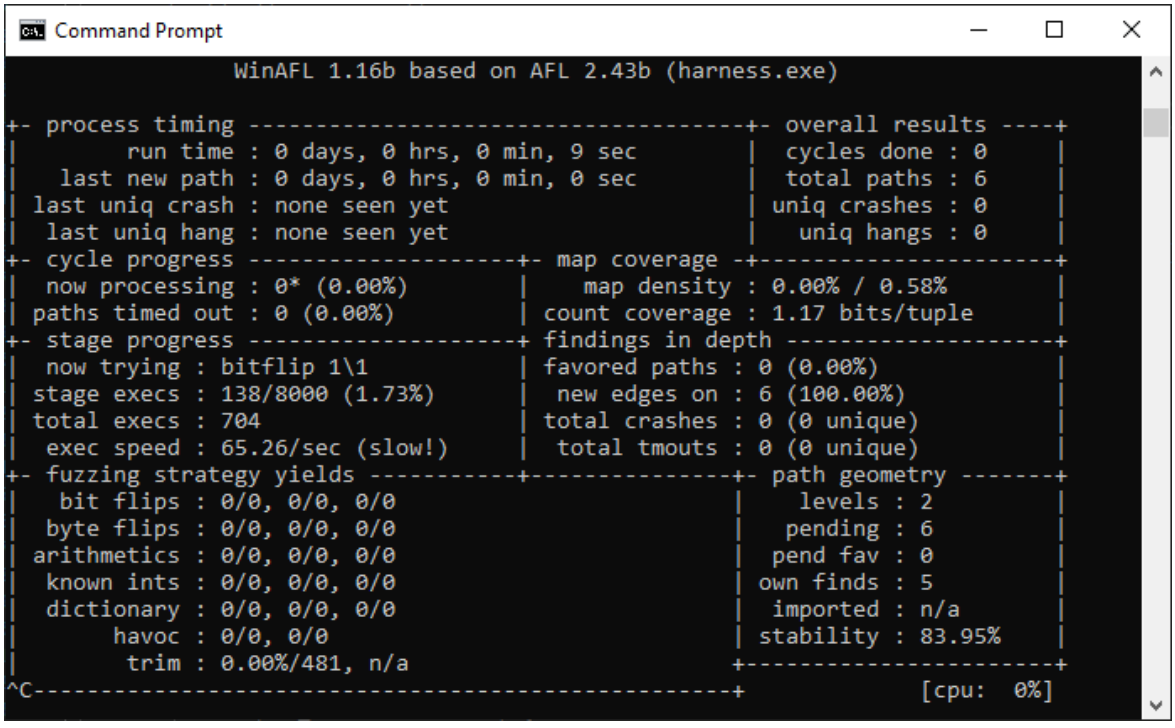
לכן על מנת לפתור את שגיאת pte_nomap עלינו להוסיף את ה-image-ים המתאימים **ממרחב הקרנל** לרשימת המודולים. לשם כך ניתן לשלוף את כתובת הבסיס, גודל ולהוריד dump של הדרייבר באמצעות windbg. לאחר מכן נוסיף פונקציה שמוסיפה את הדרייבר שהורדנו לרשימת המודולים:

```
void add_module_section_cache(char* modulename, void* base, size_t size)
{
    module_info_t *loaded_module;

    loaded_module = (module_info_t *)malloc(sizeof(module_info_t));

    strcpy(loaded_module->module_name, modulename);
    loaded_module->base = base; //module_info.lpBaseOfDll;
    loaded_module->size = size; //module_info.SizeOfImage;

    // todo put these files in a separate directory and clean it periodically
    char tmpfilename[MAX_PATH];
    sprintf(tmpfilename, "%s", modulename);

    loaded_module->isid = pt_iscache_add_file(section_cache, tmpfilename, 0, size, base);

    if (loaded_module->isid <= 0) {
        FATAL("Error adding file to pt cache.");
    }

    loaded_module->next = all_modules;
    all_modules = loaded_module;
}
```

ונקרא לה:

```
struct pt_image *image = NULL;
if ((options.decoder == DECODER_FULL_FAST) || (options.decoder == DECODER_FULL_REFERENCE)) {

    ////////////////////////////////////////////////////////////////////

    // 07/2019 pt kernel trace

    add_module_section_cache(options.driver_image, options.driver_base, options.driver_size);

    ////////////////////////////////////////////////////////////////////

    image = pt_image_alloc("winafl_image");
    module_info_t *cur_module = all_modules;

    while (cur_module) {
        if (cur_module->isid > 0) {
            int ret = pt_image_add_cached(image, section_cache, cur_module->isid, NULL);
        }
        cur_module = cur_module->next;
    }
}
```

נריץ שוב את ה-fuzzer ונבחין כי כעת איננו מקבלים שגיאות ו-WinAFL מתחיל לעבוד תוך כדי הצגת מסך הסטטוס המוכר שלו:



לסיכום שלב זה, נתקלנו בשגיאה "No instrumentation detected" שנבעה מכך שה decoder לא הצליח לפענח את ה-trace מכיוון שה image של הדרייבר היה חסר.

פתרנו את השגיאה באמצעות dump של הדרייבר והוספה של descriptor מתאים לרשימת המודולים.

בעית יציבות ה-Fuzzer

ניקח לדוגמא תרחיש שבו הדרייבר עליו מבצעים fuzzing יוצר כמה thread-ים בזמן טעינה. ה-trace שיוקלט יכיל גם את מסלולי הקוד של ה-thread-ים וגם את המסלולים שהתקבלו כתוצאה מביצוע ה-fuzzing.

מצב כזה יפגע בביצועי AFL. מדוע?

כדי להבין את משמעות הפגיעה בביצועים יש להכיר את מדד ה-Stability של AFL שאומר עד כמה ה-fuzzing "יציב". כאשר נאסף כיסוי קוד שלא נוצר מביצוע ה-fuzzing, ה-Stability קטן וביצועי ה-fuzzing נפגעים מכיוון שההחלטות ש-AFL מקבל לא בהכרח מובילות אותו להגדלת כיסוי הקוד.

בתרחיש שתיארנו מתווספים מסלולי קוד שאינם קשורים למסלולים שמתקבלים במסגרת ה-fuzzing, מסלולים אלו יטעו את AFL לחשוב שהקלטים שהוא חולל מגדילים את כיסוי הקוד.

פתרון אפשרי הוא מנגנון ב-PT שמאפשר פילטור ה-trace פר תהליך, כלומר איסוף כיסוי קוד רק בקונטקסט של תהליך ה-fuzzing.

PT מממש זאת באמצעות קינפוג ערך רגיסטר CR3 שכידוע הנו ייחודי לכל תהליך. ברגע שקובעים ערך CR3 של תהליך יעד, ה-trace יקליט רק קוד שרץ בקונטקסט של אותו תהליך. הדרייבר של windows לא תומך במנגנון זה אבל ניתן לממש אותו באמצעות הוספת shellcode בזמן ריצה.

נציין שפיתרון זה לא מטפל בכל התרחישים.

תרחיש אחר הוא דרייבר שמטפל בפסיקות ורושם callback שנקרא ע"י מערכת ההפעלה בקבלת פסיקה. סטטיסטית ייתכן שה-callback ירוץ בקונטקסט של תהליך ה-fuzzing מכיוון ש-DPC של פסיקה רץ ב-arbitrary user thread context, כלומר אנו שוב בבעיה של קוד שרץ על הדרייבר ואינו חלק מתהליך ה-fuzzing.

על מנת לפתור את הבעיה הזו מהשורש יש לפלטר את ה-trace ברמת ה-kthread או לשבת בצומת שמאפשר פילטור trace בקונטקסט של DPC. ישנם כמה כיוונים לפיתרון כזה אך הם חורגים מה scope של המאמר ולכן לא נתאר אותם.

לסיכום, השיטה שתיארנו עובדת כרגע על דרייברים שרץ עליהם קוד שמוטרג מתהליך ה-fuzzer בלבד. כמו כן תיארנו כיוונים נוספים להרחבת התמיכה גם לדרייברים אחרים, "מורכבים" יותר.

סיכום

במאמר זה הוצגה שיטה לביצוע coverage guided fuzzing בקרנל של Windows באמצעות יכולת tracing חומרתית של מעבדי אינטל. לשם כך בוצעה הרחבה לקוד של WinAFL ושינויי קוד דינמיים בדרייבר של PT ב-Windows. כמו כן הוצגו תרחישים בעייתיים שעלולים לפגוע בביצועי AFL ופתרונות אפשריים.

יובהר כי המתואר הנו POC בלבד וכי יש מקום להרחבות נוספות שיהפכו את התהליך לאוטומטי יותר.

על המחבר

אריאל ז. עובד בחברת רפאל כחוקר אבטחת מידע.

arielze[at]rafael[dot]co[dot]il

מקורות מידע נוספים

1. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3, Chapter 36, INTEL PROCESSOR TRACE:

https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf

2. Harnessing Intel Processor Trace on Windows for Vulnerability Discovery, HITB 2017

https://conference.hitb.org/hitbsecconf2017ams/materials/D1T1 - Richard Johnson - Harnessing Intel Processor Trace on Windows for Vulnerability Discovery.pdf

3. Evolutionary Kernel Fuzzing, Black Hat USA 2017

https://moflow.org/Presentations/Evolutionary Kernel Fuzzing-BH2017-rjohnson-FINAL.pdf

4. Intel Processor Trace on Linux, Tracing Summit 2015

https://halobates.de/pt-tracing-summit15.pdf

5. Hardware Tracing with Intel Processor Trace

https://hsdm.dorsal.polymtl.ca/system/files/10Dec2015_0.pdf