

Technologic Papers

Anti Anti-Debugging

מאת Zerith (אורי)

הקדמה

קוראים יקרים, במאמר זה נציג שיטות קלאסיות לבצע Anti-Debugging. במהלך המאמר נציג שיטות אלה משני צדדים: בתור מתכנת - נבין איך מבצעים פעולות Anti-Debugging שונות ובתור Reverser - נראה כיצד ניתן לעקוף אותן. הרחבה על השיטות המוצגות במאמר ניתן לקבל במאמרים הבאים:

- <http://www.codeproject.com/KB/security/AntiReverseEngineering.aspx>
- http://www.veracode.com/images/pdf/whitepaper_antidebugging.pdf

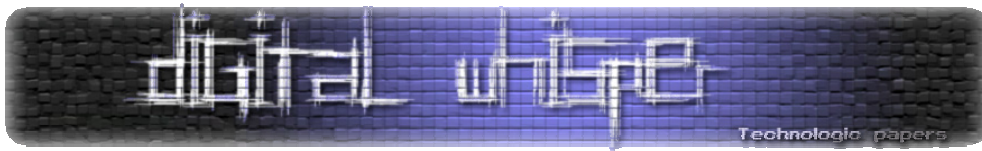
במאמר זה נעבור על השיטות שמוצגות במאמרים אלה, נסביר איך הן פועלות ונסביר איך ניתן לעקוף אותן במידה ואנו נתקלים בהן בזמן דיבוג אפליקציה. ניגע במאמר זה בכל מיני טכניקות הקשורות למגוון תחומים: זיהוי Debugger, מתקפות Debugger, מבני נתונים בקרנל ואחרות. מצד ה-Reverser נלמד שיטות איך לעקוף כל טכניקה בצורה יעילה ולהערים על אותן הטכניקות.

IsDebuggerPresent

אחת הטכניקות המוכרות והפשוטות ביותר היא שימוש בפונקציה IsDebuggerPresent. דוגמא לשימוש:

```
if (IsDebuggerPresent())
{
    //Debugger Found
    ExitProcess(0);
}
//Debugger wasn't found
```

השימוש הוא פשוט - קריאה לפונקציה IsDebuggerPresent על מנת לזהות Debugger שרץ.



איך הפונקציה פועלת?

- הפונקציה IsDebuggerPresent מחזירה לתוכנית את הבית השלישי של ה-PEB (PEB הוא קיצור של Process Environment Block) שנקבע על ידי המערכת ומייצג האם התהליך נמצא מתחת לדיבאגר או לא.
- ה-"דגל" הזה הוא לצורך ייצוגי בלבד ולא משפיע בשום אופן על ה-Debugger או על התהליך, לכן כשניגש לאפליקציה בכובע ה-Reverser נוכל לשנות אותו בחזרה למצבו המקורי (ערך 0) בלי לחשוש מהשלכות.

עקיפת הגנה זו נעשית על ידי פעולת איפוס הבית השלישי ב-PEB. בדרך כלל פעולת האיפוס נעשה על ידי ה-Debugger או ע"י Plug-in כלשהו, אך ניתן לעשות גם בקוד: (למרות שזה לא ממש פרקטי)

```
MOV EAX, DWORD PTR FS:[30] ;Offset to PEB
MOV DWORD PTR DS:[EAX+3], 0
```

למתעניינים, ניתן לקרוא על המבנה השלם של ה-PEB בכתובת הבאה:

<http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/NT%20Objects/Process/PEB.html>

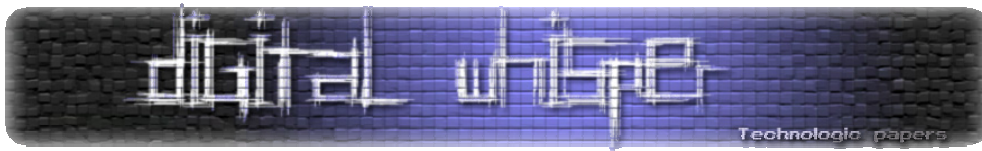
CheckRemoteDebuggerPresent

פונקציה זו היא פונקציה כמעט זהה ל-IsDebuggerPresent, היא בודקת את הדגל הנ"ל ב-PEB בתהליך הניתן כארגומנט. צורת העקיפה של הגנה זו זהה לצורת העקיפה של IsDebuggerPresent.

NtGlobalFlags

ב-PEB ישנו שדה נוסף הנקרא "NtGlobalFlag" (אופסט 0x68) שאחראי על התנהגות ה-Heaps שנוצרים בתכנית, כאשר תהליך כלשהו רץ ללא כל דיבאגר, השדה הנ"ל בדרך כלל מכיל את הערך 0. אך במקרים בהם התהליך רץ מתחת לדיבאגר, השדה יכול את הערך 0x70 שמקביל ל:

- FLG_HEAP_ENABLE_TAIL_CHECK
- FLG_HEAP_ENABLE_FREE_CHECK
- FLG_HEAP_VALIDATE_PARAMETERS



בדיקה יכולה להעשות בצורה הבאה (נשתמש בקוד אסמבלי לצורך הפשטות):

```
MOV EAX, FS:[30] ;Offset to PEB
CMP DWORD [EAX+0x68], 0x70
JNE DEBUGGER_FOUND

DEBUGGER FOUND:
התהליך רץ מתחת לדיבאגר;
```

המעקף דומה לזה של IsDebuggerPresent – ניתן גם במקרה הזה לשנות את הדגלים לערכים המקוריים שלהם כאשר התהליך לא רץ מתחת ל-Debugger – ועקפנו את הבדיקה. הדגל משפיע על יצירת Heaps באמצעות הפונקציה RtlCreateHeap כדי לעזור למתכנת המדבג את תוכניתו להבין יותר טוב מה הולך ב-Heap, אך אם נשנה אותו למצבו המקורי לא תיחיה השפעה ממשית עלינו.

INT3

רוב ה-Debuggers משתמשים בהוראה INT3 לביצוע ה-Breakpoint שלהם (למשל, ה-Debugger OLLYDBG). לכן במידה ונציב את ההוראה INT3 בקוד שלנו באופן אקראי ה-Debugger יתנהג כאילו זוהי אחת מנקודות העצירה שהמשתמש שם. בסביבה ללא Debugger, ההוראה INT3 מעבירה את השליטה אל ה-Exception Handler.

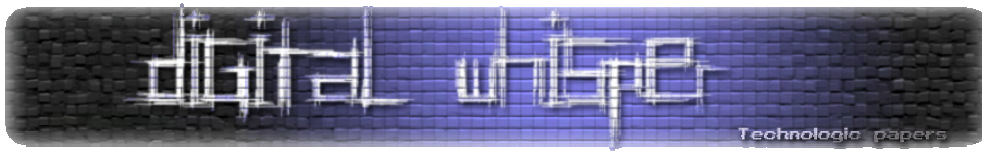
```
MOV ECX, ExceptionHandler
MOV DWORD PTR FS:[0], ExceptionHandler ; FS:[0]; נמצא ב-
INT3
//קוד זה מורץ בכלל, כנראה שאנחנו מתחת לדיבאגר. ; //CODE

ExceptionHandler: במקום אחר בקוד ;
//אין דיבאגר
//להמשיך בריצה הרגילה של התכנית
```

טכניקת Anti-debugging נוספת הכוללת את INT3 היא סריקה של הקוד להוראת INT3, יש לזכור כי ההוראה יכולה להופיע בצורה 0xCC או בצורה 0xCD 0x03.

```
for (int I = 0; I < SizeToScan; I++)
{
    if (Memory[i] == 0xCC)
        MessageBox(NULL, "DEBUGGER FOUND!", NULL, NULL);
}
```

אין דרך גורפת לעקוף את הטכניקה הנ"ל. עם זאת, אם מומשה הגנה נגד Software Breakpoints נוכל להשתמש ב-Hardware Breakpoints (שיוצגו בהמשך) במקומן כדי לעקוף את ההגנה.



Memory Breakpoint

להרבה Debuggers (כגון OLLYDBG) יש את האפשרות לשים Memory Breakpoints, הפועלים בדרך הבאה: ה-Debugger משנה את הגנת הדף בכתובת הרצויה ל-PAGE_GUARD, מצב שבו בגישה אל הדף נזרקת חריגה (STATUS_GUARD_PAGE_VIOLATION) וה-Debugger מתוודע אל גישה זו.

בכדי לנצל זאת לטובתנו (בתור מתכנתים), ניתן לשנות את הגנת הדף של פיסת הקוד ל-PAGE_GUARD. במידה ואנחנו ללא Debugger יקרא ה-Exception Handler המתאים, אך אם אנחנו רצים מתחת לדיבאגר הקוד ימשיך לרוץ כרגיל, הדיבאגר יחשוב שזו אחת מה-Memory Breakpoints שלו.

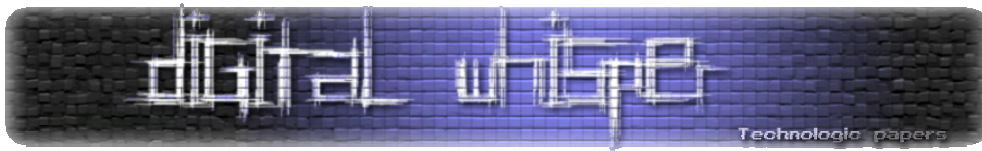
```
typedef void (*pToFunc)();
bool CheckForMemBP()
{ // הונח בפונקציה זאת כי אנחנו משתמשים בדפים של 4096 בתים, למרות שנתון
  // זה משתנה ממערכת למערכת
  DWORD Old = 0;
  void *p = VirtualAlloc(NULL, 4096, MEM_COMMIT | MEM_RESERVE,
    PAGE_EXECUTE_READWRITE);

  *(unsigned char *)p = 0xC3; //RETN
  pToFunc z = (pToFunc)p;
  VirtualProtect(p, 4096, PAGE_EXECUTE_READWRITE | PAGE_GUARD,
    &Old);

  try
  {
    z();
    return true;
  }
  __except(EXCEPTION_EXECUTE_HANDLER)
  {
    VirtualFree(p, NULL, MEM_RELEASE);
    return false;
  }
}
```

משום שנסיון גישה לדפים מסוג זה יגרום לחריגה, ה-Reverser יכול פשוט לדמות חריגה משלו בצורה ידנית במקום החריגה המקורית על מנת לעקוף את מנגנון הבדיקה הנ"ל, למשל, בעזרת ביצוע החלפה של ההוראה RETN בהוראה שתגרום לחריגה כגון WRMSR.

אם ישנה בדיקה לסוג החריגה ב-Exception Handler, ניתן גם את זאת לעקוף על-ידי שינוי של השדה ExceptionCode הקיים במבנה הנתונים ExceptionRecord אשר נשלח כארגומנט ל-Handler ל-STATUS_GUARD_PAGE_VIOLATION.



Hardware Breakpoint

Hardware Breakpoints הינם דרך נוספת לשים נקודות עצירה.

- Hardware Breakpoints מתבצעים על ידי המעבד ולא ע"י מערכת ההפעלה.
- נקודות העצירה האלו יכולות להיות ספציפיות לכתיבה, גישה, או הרצה של הכתובת.

בכל המעבדים מארכיטקטורת ia-32 קיימים שמונה אוגרים הנקראים Debug Registers (או DRs) האחראיים על נקודות עצירה אלו (Dr0-Dr7).

- Dr0-Dr3 - כל אחד מן האוגרים האלו יכול להכיל כתובת שבה יהיה נקודת העצירה, לכן מספר ה-Hardware Breakpoints מוגבל ל4 נקודות.
- Dr4-Dr5 - שמורים ע"י אינטל.
- Dr6 (Debug Status) - באמצעות אוגר זה ניתן לדעת איזה מהתנאים לעצירה קרו בעת העצירה. (כתיבה, גישה, הרצה וכו')
- Dr7 - מגדיר מתי כל נקודת עצירה עוצרת (כתיבה, גישה, הרצה), יש לאוגר זה עוד תפקיד שאינו רלוונטי לעניין שלנו.

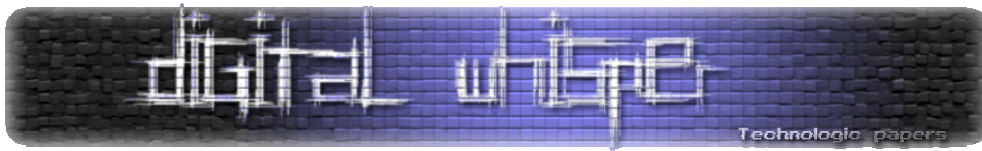
ניתן לזהות Hardware Breakpoints בכמה דרכים:

- **GetThreadContext** - הפונקציה הזאת מחזירה את מבנה הנתונים CONTEXT* של ה-Thread הנתון, מבנה זה מכיל בין שאר את האוגרים Dr0-Dr7 – ולכן ניתן לבדוק האם האוגרים Dr0-Dr3 מאופסים.

```
bool IsHWBP()
{
    CONTEXT ctx;
    ZeroMemory(&ctx, sizeof(CONTEXT));
    ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    // הפונקציה מחזירה חלקים מהמבנה לפי השדה הזה

    HANDLE hThread = GetCurrentThread();
    GetThreadContext(hThread, &ctx);
    if ((ctx.Dr0) || (ctx.Dr1) || (ctx.Dr2) || (ctx.Dr3))
        return false;
    else return true;
}
```

את השיטה הזאת ניתן בקלות לעקוף על ידי Hook לפונקציה GetThreadContext ושינוי השדות Dr0-Dr3 במבנה החוזר.



- **זיהוי באמצעות SEH (Structured Exception Handling)** – הטכניקה הזאת היא בעצם יצירת חריגה מכוונת, כשנוצרת חריגה אחד מהארגומנטים הניתנים הוא ה-CONTEXT Structure (שכאמור מכיל את האוגרים Dr0-Dr3).

איך ניתן לעקוף פה את הטכניקה? אין דרך קלה לעקוף אותה, אפשר למצוא ידנית את ה-Handle ולבצע Patching.

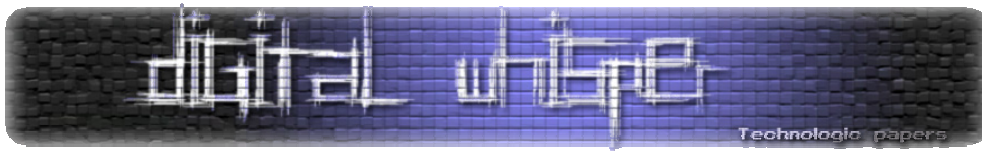
ניתן למצוא תיעוד על ה-Context Structure ב-winnt.h. נקודה למתעמקים: חשוב לשים לב שמבנה זה משתנה בין מערכת למערכת.

RDTSC

Read Time Stamp Counter – כשמה כן היא, מדובר בהוראה הקוראת את ה--Timestamp counter (שהוא 64 ביט) אל תוך EDX:EAX. ה-counter הזה הוא בעצם CPU Counter שגדל בכל CPU TICK, באמצעותו אפשר בין היתר למדוד את משך הזמן של הרצת קוד. כשהתוכנית רצה ללא Debugger הרצת פיסת קוד מסוים תארך זמן מסוים, אך מתחת ל-Debugger הרצת אותו פיסת קוד תארך הרבה יותר זמן. על מנת לזהות Debugger נמדוד זמן ריצה של פיסת קוד ובבדוק אם זמן הריצה הוא ארוך מהמצופה.

```
DWORD GenerateSerial(TCHAR* pName)
{
    DWORD LocalSerial = 0;
    DWORD RdtscLow = 0;
    __asm
    {
        rdtsc
        mov RdtscLow, eax
    }

    size_t strlen = _tcslen(pName);
    // חישוב טריאל כלשהו
    for(unsigned int i = 0; i < strlen; i++)
    {
        LocalSerial += (DWORD) pName[i];
        LocalSerial ^= 0xDEADBEEF;
    }
    __asm
    {
        rdtsc
        sub eax, RdtscLow
        cmp eax, SERIAL_THRESHOLD
        jbe NotDebugged
        push 0
    }
}
```



```
call ExitProcess  
NotDebugged:  
}  
return LocalSerial;  
}
```

דרך אחת לעקוף את הבדיקה הזאת היא לזהות את הטכניקה ופשוט לדלג עליה בתהליך ה-Debugger. דרך הרבה יותר מסובכת ממומשת בפלאגין Olly Advanced ל-OLLYDBG באופן הבא:

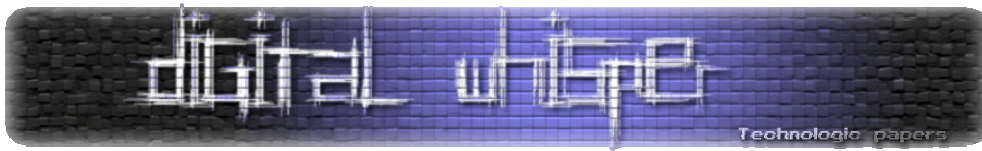
- OLLYDBG מתקין דרייבר ב-ring0 שמשנה את ה-Time Disable Bit ב-CR4 (Control Register 4), כשהביט הזה הוא 1 וההוראה RDTSC מתבצעת בטבעת הגבוהה מ-ring0, חריגת GP (General Protection Fault) תיזרק.
- הדרייבר עושה HOOK ל-Handler של חריגת ה-GP ב-IDT (Interrupt Descriptor Table), אם החריגה נזרקה בגלל ביצוע הוראת RDTSC - נגדיל את הערך שחזר מביצוע קודם של ההוראה ב-1 ונחזיר אותו.
- יש לזכור כי התקנת דרייבר כזה יכול לגרום לאי יציבות המערכת ויש להיזהר כשמשתמשים בטכניקה זאת.

ישנן מספר פונקציות מוכנות בהן ניתן להשתמש למדידת זמני ריצה כגון: `GetTickCount`, `timeGetTime`, וכן הפונקציה `QueryPerformanceCounter`. ניתן להשתמש בהן באותה הדרך שבה השתמשנו ב-RDTSC לזיהוי Debuggers.

SeDebugPrivilege

כאשר תהליך מסוים נפתח או רץ מתחת ל-Debugger, המערכת באופן אוטומטי מעניקה לו את הרשאת ה-`SeDebugPrivilege`, הרשאה זו מאפשרת לתהליך לפתוח (באמצעות `OpenProcess`) תהליכי מערכת חשובים (כגון `csrss.exe`). בשל מאפיין זה ניתן לבדוק בזמן הריצה האם התהליך הרץ יכול לפתוח `handle` לתהליך מערכת כגון `csrss.exe` וכך נדע האם אנחנו רצים מתחת ל-Debugger.

```
bool CanOpenCsrss()  
{  
    HANDLE Csrss = 0;  
    Csrss = OpenProcess(PROCESS_ALL_ACCESS,  
                        FALSE, GetCsrssProcessId());  
    // אם קריאה זו הצליחה, כנראה שאנחנו רצים מתחת לדיבאגר  
    if (Csrss != NULL)  
    {  
        CloseHandle(Csrss);  
        return true;  
    }  
    else
```



```
return false;  
}
```

מצד ה-Reverser, זו לא בעיה לעקוף את טכניקה זאת: פשוט משנים את ה"זכויות" (Privileges) הניתנות לתהליך המדובג ומורידים את ה-SeDebugPrivilege כך שלא יוכל לפתוח handle תהליכי מערכת.

Self-Debugging

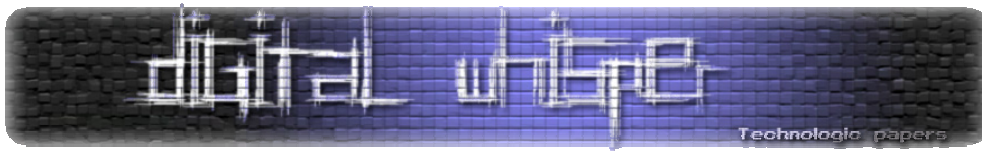
טכניקה זו הוצגה בפעם הראשונה ע"י Packer בשם Armadillo. בטכניקה זו התהליך הראשי יוצר Child-Process של עצמו (ב-Packer עצמו הוא כותב את החלק ה-Unpacked ל-Child process באמצעות WriteProcessMemory) ומדבג אותו.

בגלל שה-Child Process כבר רץ מתחת לדיבאגר (ה-Parent Process עצמו) לא יהיה ניתן לדבג אותו באמצעות שום כלי דיבאגר חיצוני נוסף, משום שיישום הפונקציה DebugActiveProcess תכשל ותחזיר STATUS_PORT_ALREADY_SET.

(הכישלון הוא בשל ששדה ה-DebugPort ב-EPROCESS (שהוא מבנה נתונים בקרנל) הוא כבר 1).

```
void DebugSelf()  
{  
    HANDLE hProcess = NULL;  
    DEBUG_EVENT de;  
    PROCESS_INFORMATION pi;  
    STARTUPINFO si;  
    ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));  
    ZeroMemory(&si, sizeof(STARTUPINFO));  
    ZeroMemory(&de, sizeof(DEBUG_EVENT));  
  
    GetStartupInfo(&si);  
  
    CreateProcess(NULL, GetCommandLine(), NULL, NULL, FALSE,  
        DEBUG_PROCESS, NULL, NULL, &si, &pi);  
  
    ContinueDebugEvent(pi.dwProcessId, pi.dwThreadId, DBG_CONTINUE);  
  
    WaitForDebugEvent(&de, INFINITE);  
}
```

פיתרון אפשרי הקיים לעקיפת טכניקה זו הוא דווקא לדבג את ה-Parent process, לבצע Hook לפונקציה WaitForDebugEvent – ובזמן שהתהליך יקרא לפונקציה בפעם הבאה, הקוד כבר יכיל הוראת קריאה ל-DebugActiveProcessStop ש"ינתק" את ה-Parent process מה-Child process שאותו אנחנו רוצים לדבג.



TLS Callbacks

טכניקה נוספת בה משתמשים פעמים רבות בכדי לבצע אריזה לקבצים היא שימוש בפונקציה הנקראת TLS Callbacks המשומשת ב-Thread Local Storage (שעליו לא ארחיב). מה שמיוחד בפונקציה הזאת, זה שה-TLS Callbacks מורצים לפני נקודת הכניסה (Entry Point) של התכנית, ככה שב-Debugger כמו OLLYDBG, כשפתחנו את התהליך, אנחנו כבר הרבה אחרי ההרצה של ה-TLS Callbacks. ניתן להשתמש בפונקציה זו על מנת להחביא קוד שאיננו רוצים שה-Debugger יראה או ידבג, כגון קוד Anti-debugging שיוּרץ לפני נקודת הכניסה.

מערכת ההפעלה צריכה לדעת מתי עליה להריץ TLS CALLBACKS, כמה כאלה יש ואיפה הם ממוקמים. מידע זה נשמר ב-PE Header של הקובץ. ה-PE header מכיל ארבעה "איזורים": DOS Header, COFF Header, Optional Header ו-Data Directories. המצביע ל-TLS Table ממוקם ב-Data Directories, ואחריו גודל ה-Table. באמצעות המצביע, נוכל למצוא את ה-TLS Table בקלות. מבנה ה-TLS Table מורכב מ-12 DWORDS:

```
DataBlockStartVa, DataBlockEndVa, IndexVariableVa, CallBackTableVa,
SizeOfZeroFill, Characteristics, NULL, NULL, NULL, TlsCallBack,
NULL, NULL.
```

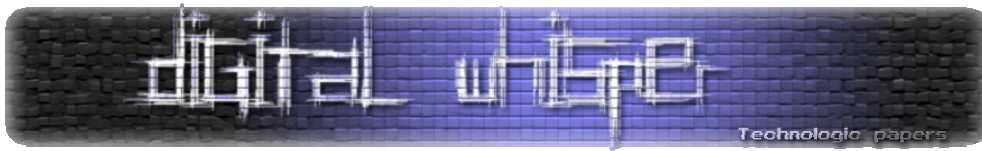
- שלושת ה-DWORDS הראשונים מצביעים ל-DWORDS אחרים המאותחלים ל-0.
- CallBackTableVa הוא מצביע לשדה ה-TlsCallBack שהזכרתי קודם.
- TlsCallBack מכיל כתובת של פונקציה שתרוץ בתוכנית.
- כל ה-NULL משמעם ריקים ומאותחלים ל-0.

בהינתן התכנית הבאה:

```
int MyFunction();
int tlsdone = 0;

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow)
{
    ExitProcess(0);
    return 0;
}

int MyFunction () {
    if (tlsdone == 0) {
        MessageBox( NULL,
                    "hello",
                    "hello",
                    MB_OK | MB_ICONINFORMATION);
        tlsdone = 1;
    }
    return 0;
}
```



הפונקציה MyFunction לא תרוץ לעולם, אך אם נכניס את כתובתה ב-TLS Table, היא תוכל לשמש לנו כ-tls callback.

יש לזכור כי TLS Callback מורצת פעמיים בתכנית – לפני נקודת הכניסה, ואחרי יציאת התכנית. לכן יש צורך במשתנה tldone בתכנית הנ"ל.

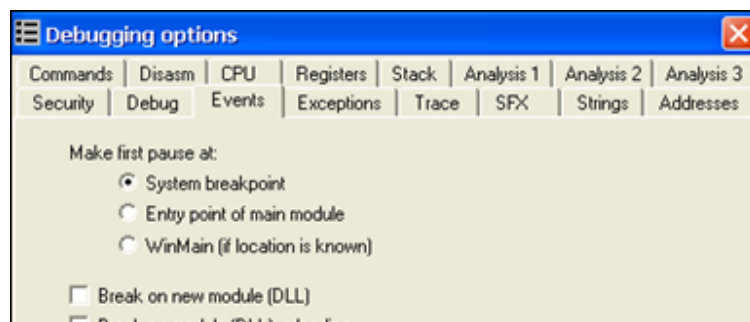
לשם השינויים שנעשה, נצטרך PE Editor פשוט כמו למשל PETools או LordPE. הדבר הראשון שיש לעשות הוא למצוא מקום פנוי מספיק בכדי לאכסן את ה-TlsTable שלנו, יש בדרכ מספיק מקום אחרי הגדרת המחרוזת האחרונה – למשל מחרוזת ה-"hello" שלנו. בדוגמא שלנו, המחרוזת ממוקמת בכתובת 0x403000. נוכל להתחיל את השינויים בכתובת 0x4030D3.

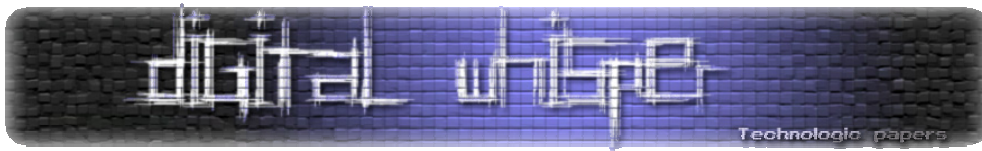
כתובת הפונקציה חייבת להיות בשדה ה-TlsCallback ב-TlsTable, השדה ממוקם ב-DWORD העשירי ומתחיל ב-0x4030F7 = 0x4030D3 + 9DWORDS. אחרי שמיקמנו את הכתובת בשדה הנכון, עלינו לשנות את שדה ה-CallbackTableVa שיצביע לשדה הנכון.

זהו הכל למעשה, כל מה שנותר לעשות הוא לשנות את ה-Data Directory שיכיל מצביע ל-TlsTable שלנו, אז נכניס שם 0x0030D3 = 0x4030D3 - 0x40000. (עלינו להחסיר את ה-ImageBase). בסוף נשנה את גודל ה-TlsTable (שהוא DWORD אחרי הפוינטר) לערך מתאים, למשל: 0x30.

עכשיו יש לנו Tls Callback שירוץ לפני נקודת הכניסה.

פיתרון לעקיפת הטכניקה במקרה ואתם משתמשים ב-OLLYDBG הוא לשנות באפשרויות של OLLYDBG שיעצור בנקודת הכניסה של המערכת במקום ב-WinMain או במקום כל נקודת כניסה אחרת.

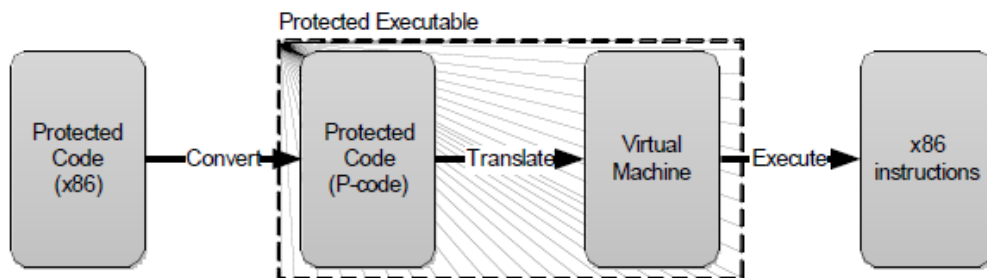




Virtual Machines

נושא המכונה הוירטואלית הוא עניין די פשוט. חלקים ספציפיים בקוד שאין ברצוננו שיהיו גלויים לגורם זר (Reverser-ל) מתורגמים לקוד מיוחד ונשמרים בזיכרון. קוד זה נקרא על ידי "מכונה וירטואלית" שהיא חלק מהתכנית שמתרגמת אותו לקוד רגיל של x86 ומורץ בתכנית.

כך הקוד האמיתי (הקוד המוגן) מוחלף בקוד מיוחד שרק המכונה וירטואלית יכולה לקרוא ולהבין. נוכל לתאר זאת בתרשים זרימה פשוט:



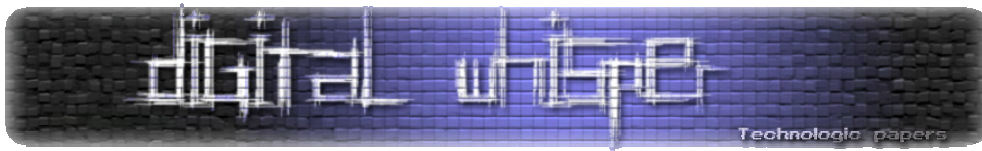
(התמונה המקורית מכאן: <https://www.blackhat.com/presentations/bh-usa-07/Yason/Presentation/bh-usa-07-yason.pdf>)

Packers חדשניים כגון Themida משתמשים בטכניקה הזאת, Themida יוצרת מכונה וירטואלית ייחודית לכל קובץ שעליו היא מגנה.

נוסיף על כך עוד דבר אשר יקשה על ה-Reverser בניתוח התוכנית עוד יותר הוא שימוש ב Obfuscation code- או "קוד הטעייה" בקוד הוירטואלי.

אין פיתרון טריוויאלי לטכניקה הזאת, מה שניתן לעשות הוא לנתח את מבנה הקוד הוירטואלי ולראות איך המכונה הוירטואלית מתרגמת אותו.

עם המידע הזה, נוכל לבנות disassembler ספציפי שידמה את פעולות המכונה הוירטואלית ויתרגם את הקוד הוירטואלי לקוד x86 ויצג לנו אותו כהוראות מפורשות. משימה לא פשוטה בכלל.



דוגמא למימוש disassembler לקוד וירטואלי ניתן למצוא בכתובת הבאה:

http://www.openrce.org/articles/full_view/28

אוי הייאוש

לסיום ברצוני לספר לכם על מקרה שקרה לי לפני מספר ימים לא רב, כשניסיתי את הדמו של Winlicense (Protector/Packer מפורסם).

דיבגתי לי בשקט וחקרתי קצת את ה-Packer, פתאום, מאמצע שום מקום – לא הצלחתי לעשות Step (F8 ב-OLLY), חשבתי לנסות לשים Breakpoint (F2) בכתובת ולחזור לאותה הנקודה, אך פתאום אני רואה שגם זה לא עובד! ברגע זה הבנתי שנפלתי לתחבולה – משהו חוסם אותי, איך יכול להיות שאני לא יכול לעשות Step, לא יכול לשים Breakpoint ולא יכול להריץ?! (F9)

לאחר בדיקה קצרה- צדקתי. התכנית אכן חסמה אותי מכל שימוש בכל מקשי ה-F1-F9. האמת – עד היום לא מצאתי באיזה טכניקה ספציפית התכנית השתמשה, אך אתן פה כמה דוגמאות.

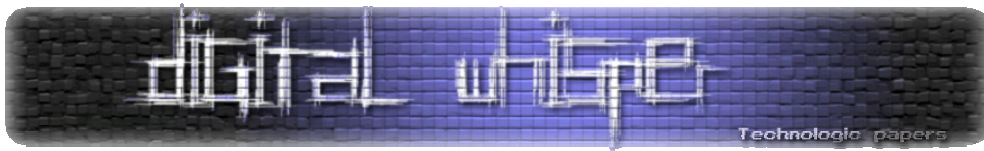
הטכניקה הזאת היא לא באמת אנטי-דיבאג, היא לא מונעת ממך, טכנית, לדבג את התכנית. הטכניקה פועלת יותר ברמה הפסיכולוגית, אך לדעתי לא פחות חזקה מכל טכניקה אחרת ואולי אף יותר.

אני חייב להגיד כי הטכניקה הזאת פשוט בלתי נסבלת, היא יכולה לעצבן ברמות שלא ניתן לתאר, היא מייאשת – ויכולה בהחלט למנוע מה-Reverser לחקור את התכנית. לדעתי, בייחוד בתחום הזה יש לזכור כי ה-Reverser הוא בן אדם ויש לנצל את המגבלות שלו, כמו סף העצבים...

דרכים למימוש:

- עושים Global Hook ל-WH_KEYBOARD_LL, ומונעים לחיצת מקשים כאלו או אחרים. (לא אפרט על כך במאמר, חפשו את הפונקציה SetWindowsHookEx).
- עוד טכניקה מהסוג הנ"ל היא הפונקציה BlockInput, שפועלת בדרך דומה – התכנית קוראת לפונקציה במצבים שאין צורך למשתמש להשתמש במקלדת או בעכבר – אך ל-Reverser זהו מצב קריטי. הפונקציה מונעת כל שימוש בעכבר או במקלדת, כל מה שצריך לעשות על מנת לממש את הטכניקה הוא פשוט לקרוא לפונקציה:

```
Blockinput(TRUE);
```



במקרה הזה ל-Reverser אין ברירה אלא לעשות ריסט למערכת, כי אין באפשרותו להמשיך את ריצת התכנית – וקריאה נוספת ל-BlockInput יכולה להתבצע אך ורק ע"י ה-Thread שקרא לה מלכתחילה.

סיכום

יש עוד מספר דרכים לא מבוטל של דרכים לבצע Anti-Debugging אך בחרתי לעצור מאמר זה כאן. כמו שזה נראה עכשיו המלחמה הזאת תמשך תמיד - אין פתרון "קסם" שיכול לעצור את ה-Reversers מלדבג את התוכניות שהמפתחים פיתחו ואין שום שיטה גנרית שבעזרתה ניתן לעקוף את כל ההגנות. מפתחים תמיד ימצאו עוד דרכים שונות ומשונות למנוע מ-Reversers לחקור את תוכניותם וה-Reversers תמיד יעלו על פתרונות חדשניים לעקוף את אותן ההגנות. יצירתיות, הכרות עם יותר שיטות, מחשבה עמוקה ותשומת לב לפרטים יתנו לכם את הכלים להתמודד עם אתגרים מורכבים יותר ויותר בתחום.