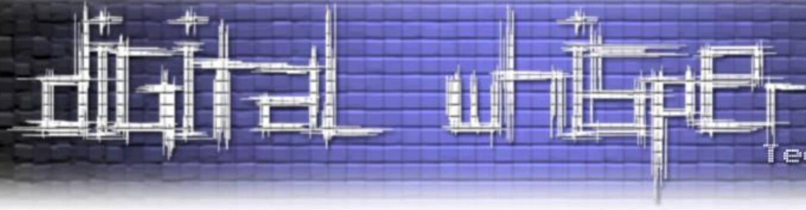


Reverse Engineering Automation - לקחת את החקירה צעד אחד קדימה

מאת תומר זית

הקדמה

המאמר יעסוק בצורך ההולך והגובר של אוטומציה בהנדסה הפוכה (Reverse Engineering Automation), כיצד שילוב אוטומציה בתהליך המחקר יכול לחסוך זמן יקר ולעזור לשאוב אינפורמציה שבלעדיו, או שהיינו מבצעים זאת בצורה פחות יסודית, או שהיינו עושים זאת בקנה מידה קטן משמעותית.

כמו כן, אראה במאמר דוגמאות של סקריפטים לאוטומציה ויהיה אפשר לגשת אליהם בחשבון ה-Github שלי, אשמח אם גם תוסיפו עוד דוגמאות מעבר לדוגמאות שנמצאות במאמר ותשגרו לי Pull Requests כדי שאוכל להוסיף אותם.

למה צריך Automation ב-Reverse Engineering?

קודם כל, חשוב לי לציין ש-**Reverse Engineering Automation** חוסך זמן בתהליך החקירה אך לא מיתר את התהליך, ושנית, בין היתר בגלל הסיבות הבאות:

1) פעולות שחוזרות על עצמן.
2) קטעי קוד דינמיים שמתגלים בהמשך התוכנית (Packers, Crypters).
3) התחמקות מהגנות כגון SSDT.
4) הקצאת זיכרון בתוך תוכנית שרצה ב-Ollydbg.

הבדלים בין Ollydbg2-Python ו-Ollydbg2-Playtime:

Ollydbg2-Playtime:

- כתיבת הסקריפטים מתבצעת בשפת **Lua** (אנו מכירים את זה קצת מ-Nmap ו-Nginx)
- מציע מגוון פתרונות כגון Event Listeners, Code Patch (אפשר לקרוא ב**autorun/detours.lua** כיצד הם משנים את **GetTickCount**) ופונקציות נוחות להוצאה מהזיכרון (למשל **ReadMemoryString**).
- מגיע עם דוקומנטציה מסודרת ונוחה.
- מאפשר שימוש בקוד שירוץ אוטומטית בהפעלת Ollydbg (AutoRun).
- מגיע עם סקריפטים לדוגמה.
- חצי Open source - כלומר ה-Api בשרת Lua שנקרא גם core הוא אופן סורס אך ה-Plugin Loader הוא קוד סגור, לאחר שיחה שלי עם יוצר ה-Plugin קיבלתי ממנו עזרה וגם הבנתי שהוא מוכר את קוד המוצר.

Ollydbg2-Python:

- כתיבת הסקריפטים מתבצעת בשפת **Python** (אנו מכירים את זה קצת מ-Immunity Debugger ו-Ida Python)
- ברגע זה אין Event Listeners, Code Patch או פונקציות נוחות להוצאת מהזיכרון.
- מגיע ללא דוקומנטציה.
- לא מאפשר שימוש בקוד שירוץ אוטומטית בהפעלת Ollydbg.
- מגיע עם סקריפטים לדוגמה.
- Ctypes - בכמה מילים שימוש עם C Structures, C Variables ו-C DLLs ישירות מקוד Python. מומלץ לקרוא על זה עוד בדוקומנטציה של Python.
- Open source מלא - כל הקוד פתוח ב-Github, מה שמאפשר עם אנשים כמונו לתרום קוד ולצמצם פערים מול Ollydbg-Playtime.

הדגמה ליכולות של Ollydbg2-Playtime

המקרה

ישנם מצבים בהם אנו צריכים לבצע פעולה בכל פעם בה נתקלנו בפונקציה מסוימת, במקרה הזה הפונקציה **IsDebuggerPresent** חוזרת מספר פעמים אדגים כיצד אפשר להשתמש באוטומציה כדי לשנות את הערך שחוזר מפונקציה זו (מי שלא מכיר את הפונקציה יכול לקרוא את המאמר **Anti Anti-Debugging** בגיליון **0x04**).

החקירה

הסתכלו על התמונה הבאה:

```
56                PUSH ESI
8B35 0080280     MOV ESI,DWORD PTR DS:[<&KERNEL32.IsDebu   Jump to KERNELBASE.IsDebuggerPresent
FFD6              CALL ESI                                  KERNEL32.IsDebuggerPresent
85C0              TEST EAX,EAX
74 07             JZ SHORT 002B1014
68 CC992B00       PUSH OFFSET 002B99CC                      ASCII "Debugger Present!!!"
EB 05             JMP SHORT 002B1019
68 E4992B00       PUSH OFFSET 002B99E4                      ASCII "Hello User"
E8 53000000       CALL 002B1071

Registers
EAX 00000001
ECX 00000001
EDX 0016DD08
EBX 7EFDE000
ESP 0028F9A0
EBP 0028F9E8
ESI 77844A25
```

אפשר לראות בתמונה שלאחר חזרה מהפונקציה **IsDebuggerPresent** ישנה בדיקה האם EAX שווה ל-0 אם כן הפונקציה תדחוף למחסנית את המחרוזת "Hello User", כרגע זהו לא המצב אז אולי נחזור כמה שלבים אחורה לדרך בה הגענו לפקודה הזו.

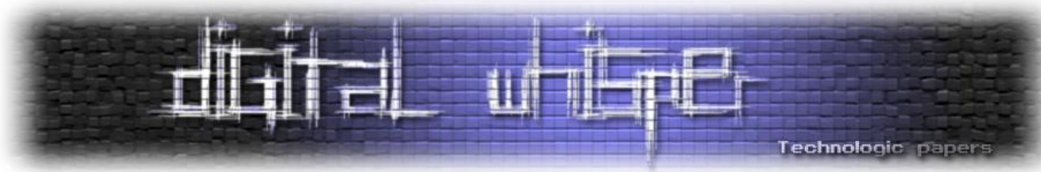
1) חיפשנו את הפונקציה **IsDebuggerPresent** בעזרת לחיצה על **CTRL + G**.

2) שמנו **Breakpoint** כדי לעצור כשהפונקציה נקראת.

3) לחצנו על **CTRL + F9** כדי להגיע לנקודה בה הפונקציה חוזרת (עוד אפשרות היא לראות את כתובת החזרה במחסנית).

4) לחצנו F8 כדי להגיע לשלב אחד אחרי הקריאה לפונקציה והגענו לבדיקה האם **EAX** שווה ל-0 (**TEST EAX, EAX** ולאחר מכן **JZ**)

5) ועכשיו נרצה לאפס את EAX כדי לדמות מצב ש-**IsDebuggerPresent** מחזיר **False**.



הסקריפט

```
1.  isDebuggerPresent = GPA("kernel32", "IsDebuggerPresent")
2.  isDebuggerPresentRet = nil
3.
4.  Event.Listen("Int3Breakpoint", function(info)
5.      if info.Address == isDebuggerPresent then
6.          if isDebuggerPresentRet == nil then
7.              isDebuggerPresentRet = Pop()
8.              Push(isDebuggerPresentRet)
9.              SetInt3Breakpoint(isDebuggerPresentRet)
10.         end
11.     elseif info.Address == isDebuggerPresentRet then
12.         EAX = 0
13.         RemoveInt3Breakpoint(isDebuggerPresentRet)
14.         isDebuggerPresentRet = nil
15.     end
16. end)
17.
18. SetInt3Breakpoint(isDebuggerPresent)
```

הסבר על הסקריפט

בסקריפט השתמשתי באחת היכולות המיוחדות של **Ollydbg2-Playtime** והיא ה-Event Listeners, בעזרת Event Listeners אנו יכולים לבצע פעולה בכל פעם שנקרא DLL, שיש Breakpoint, ש-Thread חדש נוצר וכו'.

- **GPA** - היא הפונקציה אשר מחפשת API Functions (**IsDebuggerPresent** ב-**kernel32**).
- **Event.Listen("Int3Breakpoint", function(info) end)** - כאן אנו מכריזים על פונקציה (Callback) הנקראת כאשר אירוע **Breakpoint** קרה.

אנו לוקחים את כתובת החזרה מהמחסנית בעזרת **Pop** מחזירים אותה למחסנית בעזרת **Push** (יכולנו לעשות זאת גם על ידי קריאת הזיכרון מכתובת ה-**ESP**), המשתנה **isDebuggerPresentRet** יעזור לנו בפעם הבאה לדעת שהאירוע Breakpointה מצביע על חזרה מהפונקציה **IsDebuggerPresent** בשביל שזה יקרה שמנו Breakpoint חדש בעזרת הפונקציה **SetInt3Breakpoint** לכתובת הנמצאת במשתנה **isDebuggerPresentRet**, לסוף כאשר נגיע למצב שהגענו אל היעד (אני בכתובת שנמצאת במשתנה **isDebuggerPresentRet**) נאפס את האוגר **EAX** ולאחר מכן, נמחק את ה-Breakpoint בעזרת **RemoveInt3Breakpoint**.

מטרת הסקריפט

מטרתו של הסקריפט היא למנוע מפונקציית ה-Anti-Debugging **IsDebuggerPresent** להפריע לנו בבדיקת מוצר, כמובן שיש דרכים יותר אלגנטיות לבצע זאת ויש Plugins שנועדו במיוחד בשביל זה, אך בדוגמה זו היה לי קל להראות שימוש ב-Event Listeners ב-Ollydbg2-Playtime.

הדגמה ליכולות של Ollydbg2-Python

המקרה

ישנם מצבים בהם אנו צריכים להוציא מידע מקטע זיכרון של תוכנה מסוימת, במקרה שלנו זה יהיה מערך גלובאלי של מבנים אשר מכילים מידע ששימושי לנו לחקירת התוכנה. הבעיה היא שהמערך גדול ומכיל מצביעים אז ייקח לנו זמן רב לקרוא אותו במלואו ללא פעולה אוטומטית.

החקירה

```
loc_401010:
mov     eax, dword_403018[edi]
push    eax
push    offset Format   ; "App Id: %d\n"
call    esi ; printf
mov     ecx, off_40301C[edi]
push    ecx
push    offset aAppNameS ; "App Name: %s\n"
call    esi ; printf
push    offset aCallbackOutput ; "Callback Output: \n"
call    esi ; printf
mov     edx, off_403020[edi]
call    edx
push    eax
push    offset aCallbackResult ; "Callback Result: %d\n"
call    esi ; printf
push    offset asc_40215C ; "\n"
call    esi ; printf
```

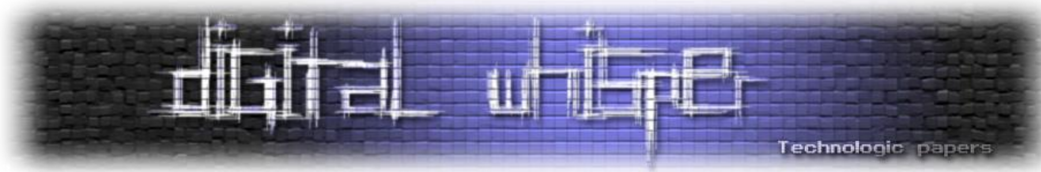
בתמונה למעלה אנו רואים חתיכת קוד מהפונקציה הראשית, בחתיכת הקוד הזו יש לולאה אשר רצה על המערך הגלובלי ומדפיסה את הנתונים בתוכו. במקרה מציאותי הריצה על המערך הגלובלי תהיה שקטה ולא תתבצע הדפסה של איברי המערך (מערך של תוכנות עם רשימות של Files, Registry ועוד.

```
.data:00403018 00 00 00 00 dword_403018    dd 0                    ; DATA XREF: _main:loc_401010↑r
.data:0040301C 0C 21 40 00 off_40301C      dd offset aTest1        ; DATA XREF: _main+1E↑r
.data:0040301C                                                     ; "Test1"
.data:00403020 80 10 40 00 off_403020      dd offset sub_401080    ; DATA XREF: _main+33↑r
.data:00403024 01 00 00 00                 dd 1
.data:00403028 04 21 40 00                 dd offset aTest2        ; "Test2"
.data:0040302C A0 10 40 00                 dd offset sub_4010A0
.data:00403030 02 00 00 00                 dd 2
.data:00403034 FC 20 40 00                 dd offset aTest3        ; "Test3"
.data:00403038 C0 10 40 00                 dd offset sub_4010C0
.data:0040303C 03 00 00 00                 dd 3
.data:00403040 F4 20 40 00                 dd offset aTest4        ; "Test4"
.data:00403044 E0 10 40 00                 dd offset sub_4010E0
```

כאן כבר נכנסנו למבנה הנתונים ואנו יכולים להבין כיצד הוא בנוי - (**int, char *, callback - void ***), כמו כן באיבר הראשון במערך ישנו מצביע למחרוזת **Test1**.

מה שנותר לנו עכשיו זה רק למצוא את כתובת תחילת המערך הגלובלי לרוץ עליו ולהדפיס את איברי המערך.

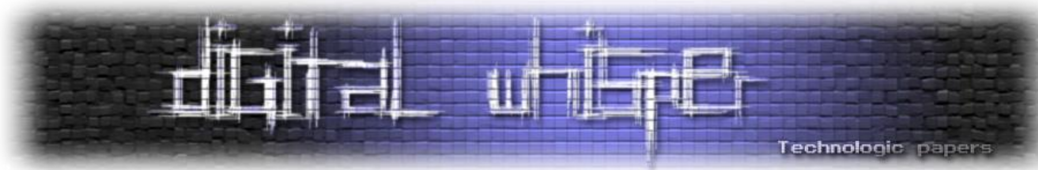
```
.rdata:004020F4 54 65 73 74+aTest4         db 'Test4',0            ; DATA XREF: .data:00403040↓o
.rdata:004020FA 00 00                      align 4
.rdata:004020FC 54 65 73 74+aTest3         db 'Test3',0            ; DATA XREF: .data:00403034↓o
.rdata:00402102 00 00                      align 4
.rdata:00402104 54 65 73 74+aTest2         db 'Test2',0            ; DATA XREF: .data:00403028↓o
.rdata:0040210A 00 00                      align 4
.rdata:0040210C 54 65 73 74+aTest1         db 'Test1',0            ; DATA XREF: .data:off_40301C↓o
.rdata:00402112 00 00                      align 4
```



כדי להגיע לכתובת של תחילת המערך, נצטרך קודם למצוא את המחרוזת הראשונה במערך Test1 ולאחר מכן להגיע לאיבר שמצביע עליו. אנחנו יכולים לראות ש-Test1 נמצא ב-rdata section והמערך עצמו נמצא ב-data section, מידע אשר יעזור לנו לבנות את הסקריפט.

הסקריפט

```
1.  import struct
2.  from ctypes import *
3.  from ollyapi import *
4.
5.
6.  class App(Structure):
7.      _fields_ = [
8.          ("id", c_int32),
9.          ("name", c_void_p),
10.         ("callback", c_void_p),
11.     ]
12.
13.
14. def bswap(val):
15.     return struct.unpack("<I", struct.pack(">I", val))[0]
16.
17. def get_section(section_name):
18.     sections = GetPESections()
19.     for section in sections:
20.         if section.sectname == section_name:
21.             return section.base
22.
23. def get_string(ea, max_length=1024):
24.     byte_array = bytearray()
25.     for offset in xrange(max_length + 1):
26.         read_chr = ReadMemory(1, ea + offset)
27.         if read_chr == '\0':
28.             break
29.         byte_array.append(read_chr)
30.
31.     return byte_array.decode("ascii")
32.
33.
34. if __name__ == '__main__':
35.     Test1_address = FindHexInPage("Test1".encode('hex'), get_section('.rdata'))
36.     Test1_pointer = FindHexInPage("%08X" % bswap(Test1_address), get_section('.data'))
37.     app_array_address = Test1_pointer - sizeof(c_int32)
38.
39.     app_size = sizeof(App)
40.     app_offset = 0
41.     while True:
42.         app = App.from_buffer_copy(ReadMemory(app_size, app_array_address + app_offset))
43.         if not app.name:
44.             break
45.
46.         print 'App Id: %d' % app.id
47.         print 'App Name: %s' % get_string(app.name)
48.         print 'App Callback Address: 0x%08X\n' % app.callback
49.
50.         app_offset += app_size
```



הסבר על הסקריפט

בתחילה נגדיר את המבנה App אשר ייצג את מבנה הנתונים שמכיל המערך הגלובאלי, לאחר מכן נגדיר פונקציות עזר:

- **bswap** - פונקציה אשר הופכת Big Endian ל-Little Endian (כתובת במעבדי Intel מיוצגות ב-Little Endian).
- **get_section** - פונקציה אשר מחזירה את כתובת ההתחלה של Section מסוים (למשל rdata, data או code).
- **get_string** - פונקציה אשר קוראת מחרוזת מהזיכרון - לא קיימת ב-Api של **Ollydbg2-Python**.

FindHexInPage - פונקציה שמחפשת Hex בקטע זיכרון מסוים ומחזירה את הכתובת שלו.

ReadMemory - כשמה כן היא, קוראת קטע זיכרון בגודל מסוים מהכתובת מסוימת.

מהלך הסקריפט

1. קבלת הכתובת של המחרוזת **Test1** על ידי השימוש בפונקציה **FindHexInPage** ו-**get_section** (כדי לחפש ב-section **rdata**).
2. קבלת הכתובת של המצביע למחרוזת **Test1** על ידי שימוש באותן הפונקציות ו-**bswap** כדי להפוך את הכתובת לLittle Endian, חיפוש הכתובת ב-section **data** כפי שגילינו בחקירה.
3. האיבר הראשון במבנה **App** מתחיל ב-**int** שהוא ה-Index במערך, לכן נצטרך להוריד את מהכתובת גודל של **Int** כדי להגיע לכתובת של תחילת המערך.
4. מה שנשאר היא הלולאה אשר תרוץ ותדפיס לנו כל איבר במערך, שם נשתמש **get_string** כדי להדפיס את המחרוזת במבנה שהיא השם של האפליקציה, **ReadMemory** בשביל לקרוא קטע זיכרון בגודל המערך ו-**from_buffer_copy** כדי להזין את המבנה בקטע הזיכרון שנלקח מהמערך.

מטרת הסקריפט

מטרת הסקריפט היא הדפסת כל איברי מערך גלובאלי כדי שנוכל לעבור עליו בצורה נוחה. עוד דרך לגלות את כתובת תחילת המערך היא למצוא Pattern ייחודי של פקודות ה-Assembly של הלולאה שרצה על המערך. (הקוד מורכב להסבר לכן אעלה אותו ל-Git ולא אסביר עליו במאמר).

הפלט של הסקריפט

```
[python-loader] Trying to execute the script located here:
App Id: 0
App Name: Test1
App Callback Address: 0x00F51080

App Id: 1
App Name: Test2
App Callback Address: 0x00F510A0

App Id: 2
App Name: Test3
App Callback Address: 0x00F510C0

App Id: 3
App Name: Test4
App Callback Address: 0x00F510E0

[python-loader] Execution is done!
```

לסיכום

שימוש ב-Reverse Engineering Automation יכול לחסוך זמן יקר בחקירה, Python היא בין השפות האולטימטיביות לכתיבת סקריפטים בגלל הקהילה הגדולה של המשתמשים, ספריות כמו Ctypes, Struct ועוד.

מומלץ לתרום קוד ל-Plugins כמו Ollydbg2-Python כדי להפוך אותם לשימושיים יותר.

תוכניות לעתיד

ברגע זה אני עמל על כתיבת Plugin ל-Open Source Debugger הראשון x64dbg **(x64dbg-Python)** כדי לאפשר Reverse Engineering Automation גם ל-x64dbg. אני הולך לשלב רעיונות מ-**Ollydbg2-Python** ו-**Ollydbg2-Playtime** ולהוסיף דברים משלי כמו פונקציית **Dump Process** שתעזור באוטומציה ל-Unpacking ועוד.

אתם מוזמנים לתרום לי קוד לכל אחד מהפרויקטים בחשבון ה-Git שלי, אשתדל לאשר קוד אשר נכתב בסטנדרטים שלי ושל היוצרים של x64dbg שאני איתם בקשר יום יומי כדי להתקדם עם הפרויקט הזה בצורה מהירה ונכונה.

בקרוב גם אעלה גם דוגמאות קוד לשימוש ב-x64dbg-Python כמו שהראיתי במאמר על-Ollydbg2-Python.

קישורים להמשך קריאה

- פרופיל ה-Github שלי:

  https://github.com/realgam3

- כל דוגמאות הקוד של הסקריפטים במאמר כולל תוכניות לדוגמה:

  https://github.com/realgam3/ReversingAutomation

- Ollydbg:

  http://www.ollydbg.de

- Ollydbg2-Python:

  https://github.com/0vercl0k/ollydbg2-python

- Ollydbg2-Playtime:

  https://code.google.com/p/ollydbg2-playtime

- X64dbg:

  http://x64dbg.com

- X64dbg-Python:

  https://github.com/realgam3/x64dbg-python

- Digital Whisper Anti Anti-Debugging:

  http://www.digitalwhisper.co.il/files/Zines/0x04/DW4-3-Anti-Anti-Debugging.pdf

- Ctypes Python Documentation:

  https://docs.python.org/2/library/ctypes.html

- SSDT:

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

- Little Endian Byte Order (Endianness):

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

- Bswap:

  http://web.itu.edu.tr/kesgin/mul06/intel/instr/bswap.html