Digital Whisper
Technologic Papers

אוטומציה וסקריפטינג ב-WinDbg

מאת סשה גולדשטיין

הקדמה

כל מפתחי הדרייברים ב-Windows מכירים את WinDbg, ה-Windows Debugger. זהו כלי ניפוי השגיאות שצמח מתוך הצוות שמפתח את מערכת ההפעלה Windows, ומשמש מפתחים רבים עד היום. הכלי הוא ורסטילי, ומשתמשים בו הן לניפוי שגיאות קלאסי, הן למטרות הנדסה לאחור, והן לפענוח וניתוח דאמפים. בין יתרונותיו של הכלי ניתן לציין את העובדה שהוא מופץ בחינם (כחלק מחבילת ה-Windows SDK), וניתן "להתקין" אותו על ידי העתקה של מספר קבצים.

אלא שגם משתמשי WinDbg וותיקים אינם מנצלים לעתים קרובות את כל יכולותיו של הכלי. בפרט, WinDbg ניחן ביכולות אוטומציה מרשימות, המאפשרות לתת מענה למגוון תסריטים מעניינים. הנה כמה דוגמאות:

- ניתן לבקש מ-WinDbg לעבור על רשימת קבצי dmp. ולבצע ניתוח אוטומטי של כל אחד מהם כדי להבין איזה רכיב תוכנה אחראי לבעיה שהתעוררה.
- ניתן לבקש מ-WinDbg לחפש בזיכרון של התהליך ערך מסוים, וברגע שהוא נמצא - להדפיס את הזיכרון מסביבו.
- ניתן לבקש מ-WinDbg לעבור על רשימה מקושרת של ערכים, ולהציג כל ערך הגדול ממספר מסוים.
- ניתן לבקש מ-WinDbg לאתר בערימה (heap) את כל האובייקטים מסוג מסוים, ואז להדפיס שדה מסוים שיש לכל אחד מהאובייקטים האלה.

במאמר זה נבחן את הדרכים השונות לבצע אוטומציה של WinDbg, ונראה כיצד לכתוב סקריפטים ואף להרחיב את הכלי בעזרת ספריות נטענות. מטבע הדברים, תקצר היריעה מכדי לכסות את כל הנושאים הנ"ל, ולכן אני מפנה אתכם לתיעוד של WinDbg ב-MSDN.

נתחיל מתסריט פשוט יחסית. נניח שעומדת לפנינו ספריה מלאה בקבצי dmp. שהבאנו ממחשבים שונים שבהם האפליקציה שלנו קרסה. כעת אנו רוצים לבצע אבחון אוטומטי על הקבצים הנ"ל, כדי להבין האם יש אולי חולשת אבטחה באפליקציה שלנו הניתנת לניצול על ידי תוקף (או שאולי סתם יש לנו באג מטופש). אחת מקבוצות מחקר האבטחה במיקרוסופט הוציאה לפני מספר שנים ספריית הרחבה ל-WinDbg המכילה פקודה בשם exploitable!, המנסה להבין האם הבאג בתוכנית מהווה חולשת אבטחה. כך למשל, אם התוכנית קרסה בגלל ניסיון הרצת קוד מהמחסנית, exploitable! ידווח על כך שהתוכנית מכילה כנראה חריגה מגבולות מערך במחסנית, המווה חולשת אבטחה פוטנציאלית.

יש ל-WinDbg אפשרות להריץ פקודה עבורנו מיד עם פתיחת קובץ ה-dmp., באופן שיאפשר אוטומציה של התהליך. אנו גם נרצה לשפוך את הפלט לקובץ כדי שנוכל לבצע עליו ניתוח אוטומטי בהמשך או לשלוח אותו לארכיון לאגירה ארוכת-טווח. עבור קובץ dmp. בודד, הפקודה שנשתמש בה תהיה:

```
windbg -z crash.dmp -c ".load C:\msr\exploitable; .logopen /t
crash_exploitable.log; !exploitable; .logclose; q"
```

בשורת הפקודה לעיל, אנו מנחים את WinDbg לפתוח לניתוח את הקובץ crash.dmp ולאחר מכן להריץ מספר פקודות, הפקודות הנמצאות בין logopen. ל-logclose. תכתבנה לקובץ לוג שנוכל לנתח בהמשך. אך כאמור, ברשותנו הרבה קבצי dmp. - ולכן נוכל להשתמש, למשל, בפקודה המובנית FOR כדי לבצע את הניתוח הנ"ל עבור כל הקבצים:

```
FOR %D IN (*.dmp) DO windbg -z %d -c ".load C:\msr\exploitable; .logopen
/t %d_.log; !exploitable; .logclose; q"
```

לולאות ותנאים

בדוגמה הקודמת הרצנו פחות או יותר אוסף קבוע של פקודות ב-WinDbg. אלא שבמקרים רבים אנו רוצים לבצע לוגיקה מורכבת או להריץ מספר פקודות שאינו ידוע מראש. לשם כך WinDbg מציע פקודות ייעודיות המאפשרות בקרת זרימה - .if, .for, .foreach - שבהן נשתמש כעת.

למשל, נניח שלפנינו מערך של ידיות (handles) לאובייקטים של מערכת ההפעלה, ואנו מעוניינים להבין מהם האובייקטים שאליהם הידיות מתייחסות. בהינתן ידית אחת, הפקודה handle! תעשה את העבודה בשבילנו, אבל אנו לא יודעים מראש את גודל המערך ולא רוצים לבזבז זמן על העתקה ידנית ומסורבלת של ערכים. יתר על כן, לפעמים אנו רוצים שפקודה מסוימת תרוץ באופן אוטומטי (למשל, בכל פעם שמגיעים לנקודת עצירה - breakpoint), וממש לא נרצה לעצור ולהעתיק ערכים מהזיכרון באופן ידני בכל פעם שזה קורה.

הפקודה הבאה מתייחסת לפונקציה WaitForMultipleObjects, שהפרמטר השני שלה הוא מערך של ידיות, והפרמטר הראשון הוא מספר הידיות במערך. נוכל להשתמש בשתי עובדות אלה כדי להציג את הפרטים על כל הידיות בכל פעם שהפונקציה תיקרא:

```
bp KernelBase!WaitForMultipleObjects ".echo WaitForMultipleObjects
called; .for (r $t0 = 0; @$t0 < @rcx; r $t0 = @$t0 + 1) { !handle
poi(@rdx + 8*@$t0) f }"
```

ובכן, מה קורה כאן? בכל פעם שנגיע לנקודת העצירה, WinDbg יפעיל בשבילנו את הפקודה for., שמריצה לולאה. כמו בלולאת for במרבית שפות התכנות, יש כאן שלושה חלקים - אתחול של מונה הלולאה, $t0, בדיקה של גבולות הלולאה מול האוגר RCX, וקידום מונה הלולאה ב-1 כדי לעבור לאיטרציה הבאה. בתוך גוף הלולאה אנו מוצאים את האיבר במערך שמעניין אותנו על ידי חישוב היסט מתחילת המערך, הנמצא באוגר RDX. האופרטור poi מחזיר את הערך שנמצא בזיכרון בכתובת הנתונה (כלומר הוא שקול לאופרטור * של C/C++).

אגב, הפקודה הנ"ל מניחה שאנו במערכת הפעלה 64-ביט, שם שני הפרמטרים הראשונים מועברים באוגרים RCX, RDX בהתאמה (לפרטים נוספים על מוסכמת הקריאה לפונקציות במערכות 64-ביט תוכלו לעיין במאמר שלי בגיליון ה-43 של Digital Whisper). הערה אחרונה לפני שממשיכים: השתמשנו במשתנה הזמני $t0, ועומדים לרשותנו 20 כאלה - $t0, $t1, ..., $t19. אם אתם צריכים יותר מ-20 משתנים זמניים, חבל מאוד. או יותר נכון: צריך להקצות זיכרון ולהשתמש בו. אני מקווה שלא נגיע למצב הזה (לפחות במאמר הנוכחי).

במקרה זה, המידע שרצינו לעבוד עליו היה קיים ישירות בזיכרון ובאוגרים, ולכן יכולנו לגשת אליהם ישירות. במקרים מסוימים אחרים, יש לנו פקודה שימושית שפולטת כמות גדולה של טקסט, ועלינו לפרסר את הפלט שלה ולהפעיל עליו פקודות נוספות. למשל, בתהליכי NET., נוכל להשתמש בפקודה DumpHeap! המציגה רשימה של אובייקטים מסוג מסוים ומאפשרת להתבונן בפרטים עליהם. אבל מה אם אנו רוצים לקבל רשימה של כל האובייקטים מסוג מסוים ואז להריץ פקודה נוספת עבור כל אובייקט? כאן יכולות הסקריפטינג של WinDbg נכנסות לפעולה.

הפקודה הבאה עוברת על כל האובייקטים מסוג System.String בזיכרון, ומציגה עבור כל אחד את תוכן המחרוזת אותה הוא מייצג. בדרך אנו משתמשים בפקודה foreach., שהיא הדרך לפרסר פלט של פקודה אחרת שורה-שורה:

```
.foreach /pS 3 /ps 2 (string {!dumpheap -type System.String}) { !dumpobj
-nofields string }
```

כדי להבין את הפקודה הזאת, עלינו להתחיל מלהבין איך נראה הפלט של הפקודה הפנימית, המציגה את רשימת כל המחרוזות. הפלט שלה נראה בערך כך[1]:

```
0:018> !dumpheap -type System.String
 Address       MT     Size
02d71228 735bacc0       14
02d71254 735bacc0      128
02d712d4 735bacc0      196
02d71408 735bacc0       22
02d71420 735bacc0       78
02d714b4 735bacc0       28
02d714d0 735bacc0       68
02d71514 7356ab98       88
02d7156c 735bacc0       28
...
```

מכיוון שמעניינת אותנו למעשה רק העמודה הראשונה מבין השלוש, אנו משתמשים באפשרויות pS/ ו-ps/ של הפקודה foreach. - אלה גורמות ללולאה לדלג על ערכים מסוימים (שלושה בהתחלה עבור הכותרות של הטבלה, ולאחר מכן שני ערכים בכל פעם כדי להתייחס רק לעמודה השמאלית).

לבסוף, נתבונן בדוגמה שבה נשתמש בתנאים. נניח שאנו רוצים לעצור (או להדפיס הודעה) בכל פעם שהתוכנית נכשלת בניסיון הקצאת זיכרון בפונקציה malloc. יתר על כן, כאשר כישלון כזה מתרחש, נרצה להדפיס את גודל ההקצאה שהתבקש (שהרי בקשות הקצאה גדולות במיוחד עלולות לגרום ל-malloc להיכשל).

הפקודה הבאה מגדירה נקודת עצירה שמבצעת את מה שאנו רוצים, בשני שלבים. בשלב הראשון, כאשר אנו נכנסים לתוך הפונקציה malloc, אנו שומרים במשתנה זמני t0 את גודל ההקצאה המבוקש. בשלב השני, כאשר הפונקציה חוזרת (ואנו מגיעים לנקודה זו בעזרת הפקודה gu), אנו בודקים את ערך החזרה (באוגר RAX - שם הוא יהיה במערכת 64-ביט) ואם הוא 0, אנו יודעים שההקצאה נכשלה ומדפיסים את המידע הדרוש:

```
bp msvcrt!malloc "r $t0 = poi(@esp+4); gu; .if (@eax == 0) { .printf
\"malloc failed, requested alloc size: %d\n\", @$t0 }"
```

שימו לב שבפקודה הנ"ל השתמשנו גם בפקודה printf., העוזרת מאוד כשאנו רוצים להדפיס ערכים מורכבים מתוך הסקריפטים שלנו. יש ל-printf. פחות או יותר את אותה התנהגות כמו ל-printf הרגילה, אבל תמיד כדאי להביט בתיעוד.

[1] למען האמת, אני משתמש כאן בפקודה DumpHeap! בצורתה זו רק לשם ההמחשה של שימושיות הפקודה foreach. - בפועל, ניתן לבקש מ- DumpHeap! לייצר פלט נקי יותר (עמודה אחת בלבד בכל שורה) על ידי הוספת - short לפקודה. במקרה כזה, לא היינו צריכים להשתמש בדגלים ps/ ו- pS/.

סקריפטים מורכבים

כבר בדוגמאות הקצרות שראינו קודם, לכתוב את כל התוכנית בשורה אחת נראה לא נוח במיוחד, ובטח לא מועיל מבחינת תחזוקה. לכן WinDbg מאפשר גם לטעון סקריפט שלם (בעל מספר שורות) מתוך קובץ נפרד, ואפילו להעביר פרמטרים בשורת הפקודה של הסקריפט כדי לגרום לו להתנהג כך או אחרת.

ראשית, אם פשוט נעביר את התוכן של אחת מהתוכניות הנ"ל לקובץ טקסט, נוכל להריץ אותה משם כסקריפט (ובמקרה כזה נוכל גם להוסיף לתוכנית הערות, בשורות המתחילות ב-$$). למשל, ניצור קובץ בשם stackwalk.wds ונשים בו את התוכנית הבאה, המבצעת מעבר ידני על מבנה המחסנית בעזרת הרשימה המקושרת שמתחילה באוגר EBP (זהו מבנה המחסנית במערכות 32-ביט):

```
$$ Walk the stack from EBP downwards until we reach 0
.for (r $t0 = @ebp; poi(@$t0) != 0; r $t0 = poi(@$t0)) {
  $$ Display the symbols closest to the potential return address on
stack
  ln poi(@ebp+4)
}
```

כעת נוכל להריץ את התוכנית מכל נקודה ב-WinDbg (כולל מנקודת עצירה באופן אוטומטי), באמצעות הפקודה הבאה:

```
$$>< stackwalk.wds
```

ובכן, מה אם הסקריפט שלנו זקוק לפרמטרים? לדוגמא, נניח שאנו רוצים לכתוב סקריפט העובר על רשימה מקושרת של מספרים ומציג אותם. כל חוליה ברשימה המקושרת מוגדרת כך:

```
typedef struct _LIST_NODE {
  struct _LIST_NODE *Next;
  int Value;
} LIST_NODE;
```

במקרה כזה, יהיה הגיוני לצפות שהסקריפט שלנו יקבל כפרמטר מצביע לתחילת הרשימה - או - אפילו יותר טוב - שם של משתנה מקומי או פרמטר של פונקציה שמצביע לתחילת הרשימה. כדי לקבל את ערכו של הפרמטר, הסקריפט שלנו ישתמש במשתנה המיוחד ${$arg0}:

```
$$ Assume this is in a file called walklist.wds
$$ Checks to see if there *is* a first argument at all
.if (${/d:$arg0} == 0) {
  .echo This script requires at least one argument
} .else {
  .for (r? $t0 = (_LIST_NODE*)${$arg0}; @$t0 != 0; r? $t0 = @$t0->Next)
{
    .printf "value = %d\n", @@(@$t0->Value)
  }
}
```

בסקריפט הנ"ל, השתמשנו ביכולת חשובה נוספת של WinDbg, והיא היכולת לבצע ולחשב ביטויים בתחביר C/C++. הפקודה המיוחדת r? (בניגוד ל-r) מבצעת השמה למשתנה זמני תוך שמירה על הטיפוס

המקורי, וכך בדוגמה שלנו $t0 הוא מסוג *LIST_NODE_. בהמשך, האופרטור המיוחד @@ גורם ל-WinDbg להתייחס לביטוי באמצעות תחביר C/C++, שאינו ברירת המחדל. לפרטים נוספים על שני מצבי הביטויים (Expression Modes) הנתמכים ב-WinDbg, עיינו בתיעוד של המוצר.

כדי להפעיל את הסקריפט הנ"ל, אנו זקוקים רק למשתנה מקומי, פרמטר, או אפילו סתם כתובת בזיכרון המכילה מצביע לראש הרשימה המקושרת. נניח ש-head הוא משתנה כזה, מסוג *LIST_NODE - אז נוכל להפעיל את הסקריפט שלנו כך:

```
$$>a< walklist.wds head
```

לסיום נושא הסקריפטים, כדאי לציין שסקריפטים יכולים גם להיות רקורסיביים. יש מקרים בהם זה בלתי נמנע - למשל, כאשר צריך לעבור על מבנה נתונים שהוא בתורו רקורסיבי, כמו עץ בינארי. סקריפטים רקורסיביים חורגים מגבולותיו של מאמר זה ודורשים טריקים עדינים כדי לא לדרוס את המשתנים הזמניים ($t0 - $t19) במהלך הפעלה רקורסיבית. פרטים נוספים ודוגמה שלמה תוכלו למצוא בבלוג שלי.

ספריות הרחבה (Extension DLLs)

כל מאמר על WinDbg לא יכול להתעלם מחבילות ההרחבה הרבות והעשירות שכלי זה מציע. ספריות הרחבה (Extension DLLs) ל-WinDbg מכילות אוסף של פונקציות גלובאליות שיכולות לקבל קלט, לייצר פלט, ולגשת לזיכרון של התהליך ולפקודות של WinDbg לצורך פעולתן. היום, WinDbg מציע שלושה מודלים לפיתוח ספריות הרחבה (כולל אפשרות לפתח חבילות הרחבה ב-C++), אבל כאן אנו נבחן רק את המודל המקורי, ה-WDbgExts, שהוא פשוט יחסית לשימוש אך קצת מוגבל ביכולותיו. לפרטים על האפשרויות הנוספות, אני מפנה אתכם כרגיל לתיעוד של WinDbg.

להלן שלד בסיסי של חבילת הרחבה טיפוסית בעלת פקודת הרחבה אחת בשם allwaits!. פקודה זו מנסה לאתר במחסנית של החוט (thread) הנוכחי את הפונקציה WaitForMultipleObjects, ולהדפיס פרטים על ידיות אובייקטי הסנכרון שהחוט ממתין לשחרורם (הפקודה תעבוד במערכות 32-ביט בלבד, שכן במערכות 64-ביט הפרמטרים אינם מועברים במחסנית).

```
#include <windows.h>
#include <wdbgexts.h>
EXT_API_VERSION g_ExtApiVersion = {
      5 ,
      5 ,
      EXT_API_VERSION_NUMBER ,
      0
} ;
WINDBG_EXTENSION_APIS ExtensionApis = {0};
USHORT SavedMajorVersion = 0xF;
USHORT SavedMinorVersion = 0;
```

```
LPEXT_API_VERSION __declspec(dllexport) ExtensionApiVersion (void)
{
      return &g_ExtApiVersion;
}

VOID __declspec(dllexport) WinDbgExtensionDllInit (
      PWINDBG_EXTENSION_APIS lpExtensionApis,
      USHORT usMajorVersion,
      USHORT usMinorVersion)
{
      ExtensionApis = *lpExtensionApis;
}
__declspec(dllexport) DECLARE_API (allwaits)
{
      BOOL detail = FALSE;
      EXTSTACKTRACE stackFrames[20];
      ULONG frames;
      CHAR symbol[1024];
      ULONG i;

      if (strlen(args) > 0 && 0 == strcmp(args, "-detail"))
      {
            detail = TRUE;
      }

      frames = StackTrace(0, 0, 0, stackFrames, ARRAYSIZE(stackFrames));
      if (frames == 0)
      {
            dprintf("Failed to obtain stack trace\n");
      }

      for (i = 0; i < frames; ++i)
      {
            ULONG_PTR offset;
            GetSymbol((PVOID)stackFrames[i].ProgramCounter, symbol,
&offset);
            if (strstr(symbol, "WaitForMultipleObjects") != NULL)
            {
                  ULONG read;
                  ULONG count;
                  ULONG PTR handles;
                  ULONG j;

                  ReadMemory(stackFrames[i].FramePointer+8, &count,
                              sizeof(count), &read);
                  ReadMemory(stackFrames[i].FramePointer+12, &handles,
                              sizeof(handles), &read);
                  dprintf("Waiting on %d handles at 0x%08x\n",
                          count, handles);

                  if (FALSE == detail)
                        return;

                  for (j = 0; j < count; ++j)
                  {
                        HANDLE handle;
```

```
                        ReadMemory(handles + j*sizeof(HANDLE), &handle,
                                   sizeof(handle), &read);
                        dprintf("\tHandle #%d: 0x%08x\n", j, handle);
                    }
                    return;
                }
            }
}
```

על בסיס הפונקציות ReadMemory, WriteMemory, GetSymbol, GetExpression, dprintf וכמה אחרות ניתן לבנות חבילות הרחבה המחפשות ערכים בזיכרון, מממשות לולאות או תוכניות רקורסיביות מורכבות, ומבצעות דברים רבים אחרים שנוח יותר לכתוב בשפה "אמיתית" כמו C/C++ מאשר בשפת הסקריפטינג ה"מיוחדת" של WinDbg.

סיכום

במאמר זה ראינו מספר דרכים לבצע אוטומציה ל-WinDbg, ואף ראינו כיצד להרחיב את המנוע המובנה באמצעות ספריות הרחבה. מגוון האפשרויות הנפרשות בפני מי ששולט ביכולות אלה עצום, והדוגמאות שראינו הן רק קצה הקרחון של הפוטנציאל הטמון בהן. אם אתם רגילים ל-gdb, ייתכן שכבר יצא לכם לכתוב סקריפטים מורכבים ששולטים על הדיבאגר שלכם. אבל אם בדרך כלל אתם משתמשים ב- Visual Studio, עולם שלם מחכה מעבר לפינה.

למידע נוסף על WinDbg ובפרט יכולות הסקריפטינג וההרחבה שלו אני ממליץ על הספרים Advanced Windows Debugging של Mario Hewardt ו-Inside Windows Debugging של Tareek Soulami. שני הספרים מביאים דוגמאות ותרחישים אמיתיים לשימוש בפקודות מתקדמות של WinDbg ובסקריפטים פשוטים ומורכבים.

על המחבר

סשה גולדשטיין הוא ה-CTO של קבוצת סלע, חברת ייעוץ, הדרכה ומיקור חוץ בינלאומית עם מטה בישראל. סשה אוהב לנבור בקרביים של Windows וה-CLR, ומתמחה בניפוי שגיאות ומערכות בעלות ביצועים גבוהים. סשה הוא מחבר הספר Pro .NET Performance, ובין היתר מלמד במכללת סלע קורסים בנושא .NET Debugging ו-Windows Internals. בזמנו הפנוי, סשה כותב בלוג על נושאי פיתוח שונים.