Digital Whisper Technologic Papers

חילוץ ופענוח פרמטרים של פונקציות המשתמשות ב-x64 Calling Convention

מאת סשה גולדשטיין

הקדמה

במאמר זה נבחן את ה-x64 Calling Convention (מוסכמת קריאה) ונראה כיצד ניתן לחלץ פרמטרים של פונקציות המשתמשות במוסכמה זו (ב-Windows). המקרים שבהם יש צורך בחילוץ ידני כזה של פרמטרים רבים, וביניהם ניתוח דאמפים, הנדסה הפוכה, ופענוח שגיאות כאשר המחסנית נדרסה בטעות בגלל באג בתוכנית או הושחתה בכוונה על ידי תוקף. תחילה נתאר בקצרה את המצב ב-x86, שהוא בדרך כלל פשוט יותר למרות שקיים מגוון רחב יותר של מוסכמות קריאה, ואז נצלול אל הפרטים ב-x64.

אבל לפני הכל: מהי מוסכמת קריאה?

מוסכמת קריאה מתארת את האופן שבו פרמטרים מועברים לפונקציה, ובהרבה מקרים מגדירה גם את האופן שבו יש לשמור אוגרים לא-נדיפים (non-volatile registers) בכניסה לפונקציה לצורך שחזורם ביציאה ממנה. למשל, אנו יכולים להסכים על מוסכמת הקריאה הבאה (שלמען הסר ספק, היא דמיונית ולא נעשה בה שימוש בפועל):

> כאשר אתם קוראים לפונקציה שלי, אתם מעבירים את הפרמטר הראשון באוגר ECX, ואת יתר הפרמטרים מעבירים במחסנית מימין לשמאל. כאשר הפונקציה מסתיימת, היא אחראית לנקות מהמחסנית את הפרמטרים שלה. ערך ההחזרה של הפונקציה, אם יש כזה, מוחזר באוגר EDI. לבסוף, הפונקציה אחראית לשמור על ערכם של האוגרים EDX ו-ESI (כלומר לשחזרם ביציאה מהפונקציה לערך שהיה להם בכניסה לפונקציה), ויתר האוגרים מותרים לדריסה.

אם כן, מוסכמת הקריאה מתעדת את הפרטים הטכניים של העברת פרמטרים לפונקציות וכן שמירה על מצב האוגרים שהמהדר יכול להשתמש בהם במהלך ביצוע הפונקציה ולאחר החזרה ממנה (לאוגרים שערכם חייב להישמר נקרא אוגרים לא-נדיפים, ולאוגרים האחרים נקרא אוגרים נדיפים). הידור נכון של התוכנית אפשרי רק כאשר שני הצדדים מסכימים על מוסכמת קריאה באופן מדויק.

מוסכמות קריאה ב-x86

מהדרים המייצרים פקודות x86 במערכת ההפעלה חלונות משתמשים בארבע מוסכמות קריאה מרכזיות. למרות שעקרונית כל מהדר יכול לבחור במוסכמת קריאה פרטית ופנימית עבור פונקציות שאינן מוחצנות מעבר לגבולות יחידת הקישור (כמו DLL או EXE), בפועל יש מספר מוגבל של מוסכמות שמרבית המהדרים משתמשים בהם, והן מתוארות בקצרה בטבלה הבאה. יש לשים לב שבכל מוסכמות הקריאה להלן, ערך ההחזרה מוחזר באוגר EAX והאוגרים הלא-נדיפים הם EBX, EDI, ESI:

מוסכמת הקריאה	כיצד מועברים הפרמטרים?	מי מנקה את המחסנית?	היכן משתמשים בזה?
Cdecl	על המחסנית מימין לשמאל	הפונקציה הקוראת	ברירת המחדל של שפת C; פונקציות בעלות מספר משתנה של פרמטרים
Stdcall	על המחסנית מימין לשמאל	הפונקציה הנקראת	Windows API, COM
Thiscall	הפרמטר הראשון (this) באוגר ECX; יתר הפרמטרים על המחסנית מימין לשמאל	הפונקציה הנקראת	שיטות מחלקה ב-++C
Fastcall	שני הפרמטרים הראשונים באוגרים ECX ו-EDX; יתר הפרמטרים על המחסנית מימין לשמאל	הפונקציה הנקראת	המהדר-בזמן-ריצה של ה-CLR (.NET)

לדוגמא, נניח שנתונה הפונקציה הבאה שמשתמשת במוסכמת הקריאה cdecl:

```
int __cdecl Add(int x, int y)
{
  int temp = x+y;
  return temp;
}
```

כדי לקרוא לפונקציה הזאת ולהעביר לה את הפרמטרים 3 ו-5, הפונקציה הקוראת תצטרך לבצע את הקוד הבא:

```
push 5
push 3
call _Add
add esp, 8
```

לעומת זאת, נניח שנתונה הפונקציה הבאה המשתמשת במוסכמת הקריאה thiscall:

```
void __thiscall Offset(Point* point, int x, int y)
{
  point->x += x;
  point->y += y;
}
```

כעת כדי לקרוא לפונקציה הזאת הפונקציה הקוראת תצטרך לבצע את הקוד הבא:

```
lea ecx, [ebp-8]        ; address of Point local variable on the stack
push 5
push 3
call _Offset
```

לפני שאנו עוברים לדון במוסכמת הקריאה של x64, שהיא עיקרו של המאמר, נציין רק שרוב מוסכמות הקריאה של x86 כרוכות בהעברת פרמטרים על המחסנית, מה שמקל מאוד על שחזור הפרמטרים בהינתן תמונה של המחסנית. אפילו כאשר משתמשים ב-thiscall או fastcall, לעתים קרובות המהדרים שומרים עותקים של הפרמטרים על המחסנית כדי להשתמש באוגרים ECX ו-EDX למטרות אחרות (וזאת משום שב-x86 מספר האוגרים קטן מאוד יחסית לדרישות של מהדרים מודרניים). לדוגמא, כך נראית תמונת המחסנית במהלך ביצוע הפונקציה Add שהוצגה קודם:

```
EBP-4 = ESP       8
EBP+0             saved EBP
EBP+4             return address
EBP+8             3
EBP+C             5
```

מוסכמת הקריאה של x64

ייתכן שנוצר הרושם שמוסכמות הקריאה של x86 הן מגוחכות במורכבותן ואין סיבה למגוון כל כך רחב של אפשרויות כדי לעשות משהו פשוט כמו העברת פרמטרים לפונקציה. יש בכך הרבה מן האמת: למעשה, בלבול של מוסכמות קריאה בין הפונקציה הקוראת לפונקציה הנקראת יכול לגרום לבעיות חמורות כמו דליפת זיכרון מהמחסנית או קריסת התוכנית עקב חוסר איזון של המחסנית. מתוך התמונה הקלוקלת הזו של מגוון מוסכמות קריאה מוזרות עולה ב-x64 מוסכמה אחת ויחידה שכל המהדרים מסכימים עליה, אבל למרבה הצער היא מקשה על ניפוי שגיאות והנדסה הפוכה הרבה יותר מאשר חברותיה מ-x86.

במעבדי x64 מספר רב הרבה יותר של אוגרים העומדים לשירות המהדר. בנוסף על האוגרים RAX, RBX, RCX, RDX, RDI, RSI שמשקפים את חבריהם ב-x86, ישנם שמונה אוגרים נוספים, R8-R15, המאפשרים

שימוש רחב הרבה יותר באוגרים ומקטינים את הצורך בהעברת פרמטרים על המחסנית או שמירת משתנים מקומיים על המחסנית. לכן, מוסכמת הקריאה של x64 היא כדלקמן:

> ארבעת הפרמטרים הראשונים של הפונקציה מועברים באוגרים RCX, RDX, R8, R9. שאר הפרמטרים מועברים על המחסנית מימין לשמאל. ערך ההחזרה של הפונקציה מוחזר באוגר RAX. ערכם של האוגרים R12, R13, R14, R15, RDI, RSI, RBX, RBP חייב להישמר (כלומר, אם הפונקציה עושה בהם שימוש, עליה לשחזר אותם לערכיהם המקוריים שהיו בעת הכניסה לפונקציה).

מוסכמת הקריאה הזאת נשמעת פשוטה יחסית, והמפתח להבנתה הוא הרצון של המהדר להשתמש כמה שפחות במחסנית. העברת פרמטרים באוגרים בשילוב עם מספר לא קטן של אוגרים נוספים שעומדים לשירות הפונקציה מאפשרות יצירה של קוד קצר יותר ויעיל יותר, הנמנע מגישות אל הזיכרון. הקושי מתעורר כאשר מנסים להבין ממבנה המחסנית את הפרמטרים שהועברו לפונקציות: כאשר הפרמטרים מועברים לפונקציה באוגרים נדיפים, ייתכן שערכם נדרס על ידי הפונקציה עצמה או על ידי פונקציה אחרת שהיא קראה לה, ושחזור ערכם עשוי להיות קשה עד בלתי אפשרי.

חילוץ פרמטרים מהמחסנית

לעתים קרובות המהדר שומר את ערכיהם של הפרמטרים על המחסנית כיוון שהוא רוצה לעשות שימוש אחר באוגרים שבהם הפרמטרים הועברו, אך בד בבד זקוק לערכם של הפרמטרים בהמשך הריצה של הפונקציה. לשם כך מוסכמת הקריאה של x64 משריינת לכל פונקציה 32 בתים של מקום על המחסנית ("home space") המשמשים לשמירת הפרמטרים במקרה הצורך. למעשה, כדי להקל על ניפוי שגיאות, המהדר של Visual Studio מספק אפשרות הנקראת HOMEPARAMS/, הגורמת למהדר לכתוב את ערכם של ארבעת הפרמטרים למחסנית מיד בכניסה לפונקציה (אפילו אם אין בכך צורך).

כאשר פונקציה עושה שימוש ב-home space, ניתן בדרך כלל לשחזר את ערכם של הפרמטרים בצורה דטרמיניסטית לחלוטין ללא צורך בטריקים מוזרים. למשל, נתבונן בפונקציה הבאה שהודרה עם האפשרות HOMEPARAMS/:

```
__int64 AddNumbers(__int64 x, __int64 y)
{
      int temp = x + y;
      return temp;
}
```

לאחר ההידור נוכל לראות שהמהדר מייצר קוד לשמירת הפרמטרים במחסנית אפילו כאשר אין בכך צורך (ואגב, מי שבנה את ה-home space עבור הפונקציה AddNumbers היא הפונקציה הקוראת):

```
?AddNumbers@@YA_J_J0@Z proc near

mov     [rsp+10h], rdx
mov     [rsp+8], rcx
mov     eax, dword ptr [rsp+8]
add     eax, dword ptr [rsp+10h]
cdqe
retn

?AddNumbers@@YA_J_J0@Z endp
```

בזמן ריצה נוכל לחלץ את ערכם של הפרמטרים מן המחסנית ללא צורך להסתמך על ערכם הנוכחי וללא צורך להבין את מחסנית הקריאות במלואה:

```
0:000> k
Child-SP          Call Site
00000000`0014f878 CallingConventions!AddNumbers+0x12
00000000`0014f880 CallingConventions!wmain+0x42
00000000`0014f8d0 CallingConventions!__tmainCRTStartup+0x10f
00000000`0014f900 kernel32!BaseThreadInitThunk+0xd
00000000`0014f930 ntdll!RtlUserThreadStart+0x1d

0:000> dq 00000000`0014f878+10 L1
00000000`0014f888  00000000`00000040

0:000> dq 00000000`0014f878+8 L1
00000000`0014f880  00000000`00000020
```

לעיתים ניתן להשתמש בטכניקה זו גם כאשר המהדר לא משתמש ב-home space, אלא סתם במשתנה מקומי שקיבל את ערכו של אחד הפרמטרים ונשמר במחסנית. כמובן, במקרים כאלה יש לקרוא בעיון את הקוד של הפונקציה כדי להבין האם ניתן לסמוך על ערכו של המשתנה המקומי בנקודת הזמן הנוכחית.

חילוץ פרמטרים מאוגרים לא-נדיפים

לעיתים המהדר משתמש באוגרים לא-נדיפים ושומר בהם באופן זמני או קבוע את ערכיהם של הפרמטרים. כיוון שמדובר באוגרים לא-נדיפים, כל עוד הפונקציה מתבצעת (אפילו אם היא קראה

לפונקציות אחרות), ערכם של האוגרים הלא-נדיפים צריך להיות שמור במקום כלשהו - באוגר עצמו אם הוא עדיין לא נדרס, או במחסנית אם הוא נשמר על ידי פונקציה אחרת למטרת דריסה.

לדוגמא, נתבונן במחסנית הבאה, שבה תוכנית קראה לפונקציה SleepEx של מערכת ההפעלה, ואנו רוצים להבין את משך הזמן שהתוכנית מבקשת לישון:

```
0:000> k
Child-SP          Call Site
00000000`0030fc58 ntdll!NtDelayExecution+0xa
00000000`0030fc60 KERNELBASE!SleepEx+0xab
00000000`0030fd00 CallingConventions!wmain
00000000`0030fd50 CallingConventions!__tmainCRTStartup+0x10f
00000000`0030fd80 kernel32!BaseThreadInitThunk+0xd
00000000`0030fdb0 ntdll!RtlUserThreadStart+0x1d
```

עקרונית, הפונקציה SleepEx קיבלה את הפרמטר באוגר RCX, אבל ערכו של האוגר יכול היה להידרס מאז. אכן, אם נתבונן בערכו כרגע, נראה שהוא קטן מדי עבור הסיטואציה שאנחנו נמצאים בה:

```
0:000> r rcx
rcx=000000000000000c
```

כעת ניתן להסתכל על הקוד של הפונקציה SleepEx כדי להבין מה היא עשתה עם הפרמטר שלה. מדובר על קוד שעבר אופטימיזציה והוא מפורק למספר חלקים המקשים על הקריאה, ולכן הבאתי להלן רק חלק ממנו:

```
0:000> uf KERNELBASE!SleepEx
KERNELBASE!SleepEx:
000007fe`fdcc1150  mov     r11,rsp
000007fe`fdcc1153  mov     qword ptr [r11+8],rbx
000007fe`fdcc1157  mov     dword ptr [rsp+10h],edx
000007fe`fdcc115b  push    rsi
000007fe`fdcc115c  push    rdi
000007fe`fdcc115d  push    r12
000007fe`fdcc115f  sub     rsp,80h
000007fe`fdcc1166  mov     edi,edx
000007fe`fdcc1168  mov     esi,ecx
…
000007fe`fdcc11c3  cmp     esi,0FFFFFFFFh
000007fe`fdcc11c6  je      KERNELBASE!SleepEx+0xc1
…
000007fe`fdcc11cc  mov     rax,rsi
000007fe`fdcc11cf  imul    rax,rax,2710h
000007fe`fdcc11d6  mov     qword ptr [rsp+20h],rax
000007fe`fdcc11db  neg     rax
```

```
000007fe`fdcc11de  mov     qword ptr [rsp+20h],rax
000007fe`fdcc11e3  lea     r12,[rsp+20h]
000007fe`fdcc11e8  mov     qword ptr [rsp+28h],r12
…
000007fe`fdcc11ed  test    rdx,rdx
000007fe`fdcc11f0  jne     KERNELBASE!SleepEx+0xd9
…
000007fe`fdcc11f6  mov     rdx,r12
000007fe`fdcc11f9  movzx   ecx,dil
000007fe`fdcc11fd  call    qword ptr [KERNELBASE!_imp_NtDelayExecution]
000007fe`fdcc1203  mov     esi,eax
000007fe`fdcc1205  mov     dword ptr [rsp+0B0h],eax
000007fe`fdcc120c  test    edi,edi
000007fe`fdcc120e  jne     KERNELBASE!SleepEx+0xb8
…
000007fe`fdcc1240  add     rsp,80h
000007fe`fdcc1247  pop     r12
000007fe`fdcc1249  pop     rdi
000007fe`fdcc124a  pop     rsi
000007fe`fdcc124b  ret
```

כבדרך אגב, אנו רואים שהפרמטר שהועבר ב-RCX (למעשה, ב-ECX) נשמר באוגר ESI. בהמשך, ערכו של ECX נדרס, אבל ערכו של ESI נראה שנשמר עד הנקודה שבה אנו נמצאים, שהיא הקריאה ל-ntdll!NtDelayExecution. כעת השאלה היא רק מה עלה בגורלו של האוגר ESI בפונקציה הבאה, ואת זה אפשר לבדוק שוב על ידי קריאה של הקוד שלה:

```
0:000> uf ntdll!NtDelayExecution
ntdll!ZwDelayExecution:
00000000`77cd1650 4c8bd1          mov     r10,rcx
00000000`77cd1653 b831000000      mov     eax,31h
00000000`77cd1658 0f05            syscall
00000000`77cd165a c3              ret
```

למרבה המזל, נפלנו על פונקציה קצרה מאוד, שמבצעת קריאת מערכת, ואינה משנה את ערכו של ESI. לכן אנו יכולים לסמוך על כך שכרגע ESI מכיל את הערך שהועבר ב-ECX לפונקציה:

```
0:000> r esi
esi=ea60

0:000> ? esi
Evaluate expression: 60000 = 00000000`0000ea60
```

ואכן, בתוכנית זו הועבר הערך 60,000 (60 שניות) לפונקציה SleepEx.

חילוץ פרמטרים יוריסטי באמצעות מעקב אחרי ביצוע התוכנית

במקרים מסוימים קל יותר לעיין בקוד התוכנית ובמספר פקודות שפת סף סביב הקריאה לפונקציה כדי להבין מהם ערכי הפרמטרים שהועברו לה. למעשה, במקרים מסוימים זו האפשרות היחידה - אם הפונקציה דרסה את ערכי הפרמטרים, לא שמרה אותם למחסנית, ולא העתיקה אותם לאוגרים לא-נדיפים, אין אפשרות לשחזר את ערכי הפרמטרים ללא התחקות מתישה אחר ביצוע התוכנית.

לדוגמא, בהינתן מחסנית הקריאות הבאה, אנו מעוניינים לגלות מה הערכים שהועברו ל-WaitForMultipleObjects כדי להבין מדוע התוכנית שלנו תקועה:

```
0:000> k
Child-SP          Call Site
00000000`0022fad8 ntdll!ZwWaitForMultipleObjects+0xa
00000000`0022fae0 KERNELBASE!WaitForMultipleObjectsEx+0xe8
00000000`0022fbe0 kernel32!WaitForMultipleObjects+0xb0
00000000`0022fc70 CallingConventions!wmain+0x6a
00000000`0022fcd0 CallingConventions!__tmainCRTStartup+0x10f
00000000`0022fd00 kernel32!BaseThreadInitThunk+0xd
00000000`0022fd30 ntdll!RtlUserThreadStart+0x1d
```

אפשר להתחיל לקרוא את הקוד של WaitForMultipleObjects כדי להבין כיצד הוא משתמש בפרמטרים, אבל לפני כן אולי כדאי להסתכל על הקוד הקורא - ייתכן שהפרמטרים מועברים בצורה שתאפשר לגלות את ערכם. ואמנם, הנה הקוד לפני הקריאה ל-WaitForMultipleObjects:

```
mov     r8d,1
lea     rdx,[rsp+30h]
lea     ecx,[r8+1]
mov     r9d,1D4C0h
mov     qword ptr [rsp+30h],rbx
mov     qword ptr [rsp+38h],rax
call    qword ptr [CallingConventions!_imp_WaitForMultipleObjects]
```

אנו רואים שחלק מהפרמטרים קיבלו ערכים קבועים של ממש - למשל, הפרמטר R8D, שקובע עבור הפונקציה האם להמתין לכל אובייקטי הסנכרון או רק לחלק מהם, מכיל את הערך 1 (כלומר TRUE). באופן דומה, הפרמטר R9D, המכיל את משך הזמן המקסימלי שיש לחכות, מכיל את הערך 1D4C0 (כלומר 120 שניות). גם הפרמטר ECX, המונה את מספר אובייקטי הסנכרון המועברים לפונקציה, מאותחל באמצעות ערכו של R8+1, כלומר ערכו הוא 2. ולבסוף, הפרמטר המעניין ביותר, שהוא מערך של אובייקטי סנכרון, נמצא באוגר RDX, וערכו נלקח מהמחסנית. כיוון שאנו יכולים למצוא את ערכו של RSP כאשר הפונקציה

main התבצעה, אנו יכולים גם למצוא את ערכו של הפרמטר וכך להבין באופן מלא כיצד הפונקציה נקראה:

```
0:000> .frame /c 03
…
CallingConventions!wmain+0x6a:
00000001`3f41106a b960ea0000      mov     ecx,0EA60h

0:000> r rsp
Last set context:
rsp=000000000022fc70

0:000> dq rsp+30 L2
00000000`0022fca0  00000000`00000024 00000000`00000028

0:000> !handle 24 f
Handle 24
  Type            Event
  Attributes      0
  GrantedAccess   0x1f0003:
         Delete,ReadControl,WriteDac,WriteOwner,Synch
         QueryState,ModifyState
  HandleCount     2
  PointerCount    5
  Name            \Sessions\1\BaseNamedObjects\Foo
  Object Specific Information
    Event Type Auto Reset
    Event is Waiting

0:000> !handle 28 f
Handle 28
  Type            Mutant
  Attributes      0
  GrantedAccess   0x1f0001:
         Delete,ReadControl,WriteDac,WriteOwner,Synch
         QueryState
  HandleCount     2
  PointerCount    5
  Name            \Sessions\1\BaseNamedObjects\Bar
  Object Specific Information
    Mutex is Free
```

מטבע הדברים, "שיטה" זו לא תמיד עובדת, אבל כאשר ניתן להשתמש בה, היא יכולה לחסוך זמן רב לעומת חיטוט באוגרים לא-נדיפים או קריאה מפורשת של ה-home space.

סיכום

במאמר זה ראינו כיצד לחלץ פרמטרים של פונקציות המשתמשות במוסכמת הקריאה של x64. למוסכמת קריאה זו יתרונות רבים מבחינת גודל ויעילות הקוד, אבל היא מוסיפה קושי רב לניתוח אוטומטי (ואפילו ידני) של דאמפים, הנדסה הפוכה, וניפוי שגיאות בזמן ריצה. ראינו מספר שיטות יוריסטיות, וכדאי לציין - עבור המשתמשים ב-WinDbg - שקיימת הרחבה לכלי זה המאפשרת לבצע חלק מהעבודה לעיל באופן אוטומטי. הרחבה זו נקראת CMKD, וניתן להוריד אותה בחינם מכאן:

http://www.codemachine.com/tool_cmkd.html

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.