DUmps Binaries Logs INternals

School of Security

Reversing Disassembly Reconstruction Accelerated

Third Edition

Dmitry Vostokov
Software Diagnostics Services

Published by OpenTask, Republic of Ireland

OpenTask books and magazines are available through booksellers and distributors worldwide. For further information or comments, send requests to press@opentask.com.

A CIP catalog record for this book is available from the British Library.

ISBN-l3: 978-1-912636-67-9 (Paperback)

Revision 3.00 (September 2023)

Contents

Exercise R1

Goal: Review x64 assembly fundamentals; learn how to reconstruct stack trace manually.

ADDR Patterns: Universal Pointer, Symbolic Pointer S^2, Interpreted Pointer S^3, Context Pyramid.

Memory Cell Diagrams: Register, Pointer, Stack Frame.

1. Launch WinDbg.

2. Open \ADDR\MemoryDumps\Windows10\notepad.dmp.

3. We get the following output:

```
Microsoft (R) Windows Debugger Version 10.0.25921.1001 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.


Loading Dump File [C:\ADDR\MemoryDumps\Windows10\notepad.dmp]
User Mini Dump File with Full Memory: Only application data is available


************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*
Symbol search path is: srv*
Executable search path is:
Windows 10 Version 18362 MP (2 procs) Free x64
Product: WinNt, suite: SingleUserTS Personal
Edition build lab: 18362.1.amd64fre.19h1_release.190318-1202
Debug session time: Thu Sep  5 07:37:05.000 2019 (UTC + 1:00)
System Uptime: 0 days 0:03:43.584
Process Uptime: 0 days 0:02:07.000
.........................................
For analysis of this file, run !analyze -v
win32u!NtUserGetMessage+0x14:
00007ffe`dad31164 c3              ret
```

4. We open a log file:

```
0:000> .logopen C:\ADDR\MemoryDumps\R1.log
Opened log file 'C:\ADDR\MemoryDumps\R1.log'
```

5. We get this stack trace:

```
0:000> k
 # Child-SP          RetAddr           Call Site
00 000000a1`c2ccf988 00007ffe`dc19477d win32u!NtUserGetMessage+0x14
01 000000a1`c2ccf990 00007ff7`437da3d3 user32!GetMessageW+0x2d
02 000000a1`c2ccf9f0 00007ff7`437f02b7 notepad!WinMain+0x293
03 000000a1`c2ccfac0 00007ffe`dc557bd4 notepad!__mainCRTStartup+0x19f
04 000000a1`c2ccfb80 00007ffe`ddc6cee1 kernel32!BaseThreadInitThunk+0x14
05 000000a1`c2ccfbb0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

6. Let's check the main CPU registers:

```
0:000> r
rax=0000000000001009 rbx=000000a1c2ccfa40 rcx=000000a1c2ccfa40
rdx=0000000000000000 rsi=0000000000000000 rdi=00007ff7437d0000
rip=00007ffedad31164 rsp=000000a1c2ccf988 rbp=000000a1c2ccfa59
 r8=0000000000000080  r9=0000000000010412 r10=0000000000010412
r11=1151005044840000 r12=0000000000000000 r13=0000000000000000
r14=00007ff7437d0000 r15=0000000000000001
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
win32u!NtUserGetMessage+0x14:
00007ffe`dad31164 c3              ret
```

Note: The register parts and naming are illustrated in the MCD-R1.xlsx A section.

7. The current instruction registers (registers that are used and affected by the current instruction or semantically tied to it) can be checked by **r.** command:

```
0:000> r.
At return instr, rax = 1009
```

8. Any register value or its named parts can be checked with the **?** command:

```
0:000> ? r10
Evaluate expression: 66578 = 00000000`00010412

0:000> ? r10d
Evaluate expression: 66578 = 00000000`00010412

0:000> ? r10w
Evaluate expression: 1042 = 00000000`00000412

0:000> ? r10b
Evaluate expression: 18 = 00000000`00000012
```

Original x86 registry set can be accessed using mnemonics:

```
0:000> ? rbx
Evaluate expression: 694757947968 = 000000a1`c2ccfa40

0:000> ? ebx
Evaluate expression: 694757947968 = 000000a1`c2ccfa40

0:000> ? bx
Evaluate expression: 64064 = 00000000`0000fa40

0:000> ? bl
Evaluate expression: 64 = 00000000`00000040

0:000> ? bh
Evaluate expression: 250 = 00000000`000000fa

0:000> ? rbp
Evaluate expression: 694757947993 = 000000a1`c2ccfa59

0:000> ? ebp
Evaluate expression: 694757947993 = 000000a1`c2ccfa59
```

```
0:000> ? bp
Evaluate expression: 64089 = 00000000`0000fa59

0:000> ? bpl
Evaluate expression: 89 = 00000000`00000059
```

Note: It appears that the version of WinDbg that we used when writing this revision didn't differentiate between R and E register names, but the **r** command did:

```
0:000> r ebx
ebx=c2ccfa40

0:000> r ebp
ebp=c2ccfa59
```

9. Individual parts can also be interpreted using the typed **r** command (here, we format them as unsigned values, see WinDbg help for all other format types, for example, **ib** for signed bytes):

```
0:000> r r9
r9=0000000000010412

0:000> r r9:uq
r9=0000000000010412

0:000> r r9:ud
r9=00000000 00010412

0:000> r r9:uw
r9=0000 0000 0001 0412

0:000> r r9:ub
r9=00 00 00 00 00 01 04 12

0:000> r r9:ib
r9=0 0 0 0 0 1 4 18
```

10. Any registry value can be interpreted as a pointer to memory cells, a memory address (**Universal Pointer** pattern vs. a pointer that was originally designed to be such). However, memory contents at that address may be inaccessible or unknown, as in the case of RAX below.

```
0:000> dp rax
00000000`00001009  ????????`???????? ????????`????????
00000000`00001019  ????????`???????? ????????`????????
00000000`00001029  ????????`???????? ????????`????????
00000000`00001039  ????????`???????? ????????`????????
00000000`00001049  ????????`???????? ????????`????????
00000000`00001059  ????????`???????? ????????`????????
00000000`00001069  ????????`???????? ????????`????????
00000000`00001079  ????????`???????? ????????`????????
```

Note: The following output for RDI is illustrated in the MCD-R1.xlsx B section.

```
0:000> dp rdi
00007ff7`437d0000  00000003`00905a4d 0000ffff`00000004
00007ff7`437d0010  00000000`000000b8 00000000`00000040
00007ff7`437d0020  00000000`00000000 00000000`00000000
00007ff7`437d0030  00000000`00000000 000000f8`00000000
```

```
00007ff7`437d0040  cd09b400`0eba1f0e 685421cd`4c01b821
00007ff7`437d0050  72676f72`70207369 6f6e6e61`63206d61
00007ff7`437d0060  6e757220`65622074 20534f44`206e6920
00007ff7`437d0070  0a0d0d2e`65646f6d 00000000`00000024
```

11. We can also specify a range or limit to just one value and use finer granularity for memory dumping:

```
0:000> dp rdi L1
00007ff7`437d0000  00000003`00905a4d
```

Note: The similar output for RDI as below is illustrated in the MCD-R1.xlsx C section.

```
0:000> dd rdi
00007ff7`437d0000  00905a4d 00000003 00000004 0000ffff
00007ff7`437d0010  000000b8 00000000 00000040 00000000
00007ff7`437d0020  00000000 00000000 00000000 00000000
00007ff7`437d0030  00000000 00000000 00000000 000000f8
00007ff7`437d0040  0eba1f0e cd09b400 4c01b821 685421cd
00007ff7`437d0050  70207369 72676f72 63206d61 6f6e6e61
00007ff7`437d0060  65622074 6e757220 206e6920 20534f44
00007ff7`437d0070  65646f6d 0a0d0d2e 00000024 00000000
```

Note: Visible wwxxyyzz in the output of the **dp** command: ASCII string fragments, an example of **Regular Data** memory analysis pattern (UNICODE fragments have 00xx00yy pattern).

```
0:000> dw rdi
00007ff7`437d0000  5a4d 0090 0003 0000 0004 0000 ffff 0000
00007ff7`437d0010  00b8 0000 0000 0000 0040 0000 0000 0000
00007ff7`437d0020  0000 0000 0000 0000 0000 0000 0000 0000
00007ff7`437d0030  0000 0000 0000 0000 0000 0000 00f8 0000
00007ff7`437d0040  1f0e 0eba b400 cd09 b821 4c01 21cd 6854
00007ff7`437d0050  7369 7020 6f72 7267 6d61 6320 6e61 6f6e
00007ff7`437d0060  2074 6562 7220 6e75 6920 206e 4f44 2053
00007ff7`437d0070  6f6d 6564 0d2e 0a0d 0024 0000 0000 0000

0:000> db rdi
00007ff7`437d0000  4d 5a 90 00 03 00 00 00-04 00 00 00 ff ff 00 00  MZ..............
00007ff7`437d0010  b8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00  ........@.......
00007ff7`437d0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00007ff7`437d0030  00 00 00 00 00 00 00 00-00 00 00 00 f8 00 00 00  ................
00007ff7`437d0040  0e 1f ba 0e 00 b4 09 cd-21 b8 01 4c cd 21 54 68  ........!..L.!Th
00007ff7`437d0050  69 73 20 70 72 6f 67 72-61 6d 20 63 61 6e 6e 6f  is program canno
00007ff7`437d0060  74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20  t be run in DOS
00007ff7`437d0070  6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00  mode....$.......
```

Note: The **dc** command combines **dd** with ASCII output:

```
0:000> dc rdi
00007ff7`437d0000  00905a4d 00000003 00000004 0000ffff  MZ..............
00007ff7`437d0010  000000b8 00000000 00000040 00000000  ........@.......
00007ff7`437d0020  00000000 00000000 00000000 00000000  ................
00007ff7`437d0030  00000000 00000000 00000000 000000f8  ................
00007ff7`437d0040  0eba1f0e cd09b400 4c01b821 685421cd  ........!..L.!Th
00007ff7`437d0050  70207369 72676f72 63206d61 6f6e6e61  is program canno
00007ff7`437d0060  65622074 6e757220 206e6920 20534f44  t be run in DOS
00007ff7`437d0070  65646f6d 0a0d0d2e 00000024 00000000  mode....$.......
```

Note: If you have noticed a slight delay when dumping memory pointed by registers, then the faster equivalent approach is to use **@** prefix, for example, **@rax**:

```
0:000> db @rcx
000000a1`c2ccfa40  00 00 00 00 00 00 00 00-13 01 00 00 00 00 00 00  ................
000000a1`c2ccfa50  48 7f 00 00 00 00 00 00-d0 66 41 dc fe 7f 00 00  H........fA.....
000000a1`c2ccfa60  27 e3 01 00 61 04 00 00-6a 00 00 00 00 00 00 00  '...a...j.......
000000a1`c2ccfa70  7a b6 9e e2 4d 71 58 4d-a5 98 46 de e8 7e 62 0b  z...MqXM..F..~b.
000000a1`c2ccfa80  d5 3e b5 14 20 7b 00 00-43 a6 b2 dd fe 7f 00 00  .>.. {..C.......
000000a1`c2ccfa90  00 00 00 00 00 00 00 00-80 31 7f 43 f7 7f 00 00  .........1.C....
000000a1`c2ccfaa0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
000000a1`c2ccfab0  00 00 00 00 00 00 00 00-b7 02 7f 43 f7 7f 00 00  ...........C....
```

12. Notice a difference between a value and its organization in memory stemmed from the little-endian organization of the Intel x86-x64 platform (least significant parts are located at lower addresses):

```
0:000> dq @rbp L1
000000a1`c2ccfa59  2700007f`fedc4166

0:000> dd @rbp L2
000000a1`c2ccfa59  fedc4166 2700007f
```

Note: The similar double word output for RDI is illustrated in the MCD-R1.xlsx C section.

```
0:000> dq @r14 L1
00007ff7`437d0000  00000003`00905a4d

0:000> dw @r14 L4
00007ff7`437d0000  5a4d 0090 0003 0000

0:000> dq @r14 L1
00000000`ff130000  00000003`00905a4d

0:000> db @r14 L8
00000000`ff130000  4d 5a 90 00 03 00 00 00                         MZ......
```

13. Every value can be associated with a symbolic value from PDB symbols files or the binary (exported symbols) if available. We call this **Symbolic Pointer** or **S²**:

```
0:000> dps @rbx
000000a1`c2ccfa40  00000000`00000000
000000a1`c2ccfa48  00000000`00000113
000000a1`c2ccfa50  00000000`00007f48
000000a1`c2ccfa58  00007ffe`dc4166d0 msctf!CThreadInputMgr::TimerProc
000000a1`c2ccfa60  00000461`0001e327
000000a1`c2ccfa68  00000000`0000006a
000000a1`c2ccfa70  4d58714d`e29eb67a
000000a1`c2ccfa78  0b627ee8`de4698a5
000000a1`c2ccfa80  00007b20`14b53ed5
000000a1`c2ccfa88  00007ffe`ddb2a643 msvcrt!initterm+0x43
000000a1`c2ccfa90  00000000`00000000
000000a1`c2ccfa98  00007ff7`437f3180 notepad!_xi_z
000000a1`c2ccfaa0  00000000`00000000
000000a1`c2ccfaa8  00000000`00000000
000000a1`c2ccfab0  00000000`00000000
000000a1`c2ccfab8  00007ff7`437f02b7 notepad!__mainCRTStartup+0x19f
```

```
0:000> ln 00007ffe`dc4166d0
Browse module
Set bu breakpoint

(00007ffe`dc4166d0)   msctf!CThreadInputMgr::TimerProc    |  (00007ffe`dc416718)
msctf!CThreadInputMgr::OnTimerEvent
Exact matches:

0:000> dt 00007ffe`dc4166d0
CThreadInputMgr::TimerProc
Symbol  not found.
```

Note: The address 000000a1`c2ccfa58 that points to **00007ffe`dc4166d0** doesn't have an associated symbol:

```
0:000> dt 000000a1`c2ccfa58
Symbol not found at address 000000a1c2ccfa58.
```

Note: The next instruction pointer address contained in RIP should have an associated symbol of the current function in our example because we have symbols for *win32u.dll*:

```
0:000> ? @rip
Evaluate expression: 140732569686372 = 00007ffe`dad31164

0:000> dt @rip
NtUserGetMessage
Symbol  not found.

0:000> r
rax=0000000000001009 rbx=000000a1c2ccfa40 rcx=000000a1c2ccfa40
rdx=0000000000000000 rsi=0000000000000000 rdi=00007ff7437d0000
rip=00007ffedad31164 rsp=000000a1c2ccf988 rbp=000000a1c2ccfa59
 r8=0000000000000080  r9=0000000000010412 r10=0000000000010412
r11=1151005044840000 r12=0000000000000000 r13=0000000000000000
r14=00007ff7437d0000 r15=0000000000000001
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
win32u!NtUserGetMessage+0x14:
00007ffe`dad31164 c3              ret
```

14. Now we come to the next pointer level after its value and symbol: its interpretation. We call it an **Interpreted Pointer**, **S³**. Such interpretation is implemented either via typed structures (the **dt** command) or via various WinDbg extension commands (**!** commands) that format information for us. In our example, we would like to check the memory pointed to by the value of the RBX register. We suspect it might be MSG structure related to the "get message" loop:

```
typedef struct tagMSG {
  HWND   hwnd;
  UINT   message;
  WPARAM wParam;
  LPARAM lParam;
  DWORD  time;
  POINT  pt;
} MSG;
```

```
0:000> dp @rbx
000000a1`c2ccfa40  00000000`00000000 00000000`00000113
000000a1`c2ccfa50  00000000`00007f48 00007ffe`dc4166d0
000000a1`c2ccfa60  00000461`0001e327 00000000`0000006a
000000a1`c2ccfa70  4d58714d`e29eb67a 0b627ee8`de4698a5
000000a1`c2ccfa80  00007b20`14b53ed5 00007ffe`ddb2a643
000000a1`c2ccfa90  00000000`00000000 00007ff7`437f3180
000000a1`c2ccfaa0  00000000`00000000 00000000`00000000
000000a1`c2ccfab0  00000000`00000000 00007ff7`437f02b7
```

Note: The raw structure makes sense for WM_TIMER message (0x113) where wParam is a timer ID (**7f48**), and usually a callback function (lParam) is NULL (0x0) but in our case it is not (**00007ffe`dc4166d0**, as we saw previously msctf!CThreadInputMgr::TimerProc). Also, mouse pointer data makes sense. Unfortunately, MSG structure is not available in symbol files available for notepad memory dump. However, we can load a different unrelated module with better symbol files, for example, CPUx64.exe from C:\ADDR\MemoryDumps\ExtraSymbols, which was compiled as a Windows application with full symbols and so should have structures necessary for thread message loop processing (**Injected Symbols** memory analysis pattern).

15. We specify an additional symbol file path:

```
0:000> .sympath+ C:\ADDR\MemoryDumps\ExtraSymbols
Symbol search path is: srv*;C:\ADDR\MemoryDumps\ExtraSymbols
Expanded Symbol search path is:
cache*;SRV*https://msdl.microsoft.com/download/symbols;c:\addr\memorydumps\extrasymbols

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*
OK                                             C:\ADDR\MemoryDumps\ExtraSymbols
```

We need to find an address to “load” the CPUx64 module with its symbols. We choose a committed address **0`198d0000** the output of the **!address** command:

```
0:000> !address

Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...

        BaseAddress      EndAddress+1        RegionSize     Type       State                 Protect             Usage
----------------------------------------------------------------------------------------------------------------------
+        0`00000000        0`198d0000        0`198d0000             MEM_FREE    PAGE_NOACCESS                       Free
+        0`198d0000        0`198d1000        0`00001000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE                     <unknown>  [2.........J....]
+        0`198d1000        0`198e0000        0`0000f000             MEM_FREE    PAGE_NOACCESS                       Free
+        0`198e0000        0`198e1000        0`00001000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE                     <unknown>  [0.........J....]
+        0`198e1000        0`7ffe0000        0`666ff000             MEM_FREE    PAGE_NOACCESS                       Free
+        0`7ffe0000        0`7ffe1000        0`00001000 MEM_PRIVATE MEM_COMMIT  PAGE_READONLY                      Other      [User Shared Data]
+        0`7ffe1000        0`7ffef000        0`0000e000             MEM_FREE    PAGE_NOACCESS                       Free
+        0`7ffef000        0`7fff0000        0`00001000 MEM_PRIVATE MEM_COMMIT  PAGE_READONLY                     <unknown>  [.........Z...M6.]
+        0`7fff0000       a1`c2c50000       a1`42c60000             MEM_FREE    PAGE_NOACCESS                       Free
+       a1`c2c50000       a1`c2cbc000        0`0006c000 MEM_PRIVATE MEM_RESERVE                                 Stack      [~0; 740.750]
        a1`c2cbc000       a1`c2cbf000        0`00003000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE|PAGE_GUARD       Stack      [~0; 740.750]
        a1`c2cbf000       a1`c2cd0000        0`00011000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE                   Stack      [~0; 740.750]
+       a1`c2cd0000       a1`c2e00000        0`00130000             MEM_FREE    PAGE_NOACCESS                       Free
+       a1`c2e00000       a1`c2f93000        0`00193000 MEM_PRIVATE MEM_RESERVE                                 <unknown>
        a1`c2f93000       a1`c2f94000        0`00001000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE                   PEB        [740]
        a1`c2f94000       a1`c2f96000        0`00002000 MEM_PRIVATE MEM_COMMIT  PAGE_READWRITE                   TEB        [~0; 740.750]
[...]
```

```
0:000> .reload /f C:\ADDR\MemoryDumps\ExtraSymbols\CPUx64=0`198d0000
*** WARNING: Unable to verify timestamp for CPUx64

0:000> lm m CPU*
Browse full module list
start             end                 module name
00000000`198d0000 00000000`198d0000   CPUx64   T (private pdb symbols)
C:\ProgramData\dbg\sym\CPUx64.pdb\C9F083A312BD4F33801B4D0FDFF97DBA1\CPUx64.pdb
```

16. Now we can use MSG structure:

```
0:000> dt MSG
CPUx64!MSG
   +0x000 hwnd             : Ptr64 HWND__
   +0x008 message          : Uint4B
   +0x010 wParam           : Uint8B
   +0x018 lParam           : Int8B
   +0x020 time             : Uint4B
   +0x024 pt               : tagPOINT

0:000> dt -r MSG
CPUx64!MSG
   +0x000 hwnd             : Ptr64 HWND__
      +0x000 unused           : Int4B
   +0x008 message          : Uint4B
   +0x010 wParam           : Uint8B
   +0x018 lParam           : Int8B
   +0x020 time             : Uint4B
   +0x024 pt               : tagPOINT
      +0x000 x                : Int4B
      +0x004 y                : Int4B

0:000> dt -r MSG @rbx
CPUx64!MSG
   +0x000 hwnd             : (null)
   +0x008 message          : 0x113
   +0x010 wParam           : 0x7f48
   +0x018 lParam           : 0n140732593694416
   +0x020 time             : 0x1e327
   +0x024 pt               : tagPOINT
      +0x000 x                : 0n1121
      +0x004 y                : 0n106
```

17. When we have an exception such as a breakpoint or access violation, the values of the thread CPU registers are saved in the so-called exception context structure, and valid for the currently executing function and the instruction pointed to by the RIP register (the topmost frame). In other situations, such as a manual memory dump, we can only be sure about some registers such as RIP and RSP:

```
0:000> k
 # Child-SP          RetAddr           Call Site
00 000000a1`c2ccf988 00007ffe`dc19477d win32u!NtUserGetMessage+0x14
01 000000a1`c2ccf990 00007ff7`437da3d3 user32!GetMessageW+0x2d
02 000000a1`c2ccf9f0 00007ff7`437f02b7 notepad!WinMain+0x293
03 000000a1`c2ccfac0 00007ffe`dc557bd4 notepad!__mainCRTStartup+0x19f
04 000000a1`c2ccfb80 00007ffe`ddc6cee1 kernel32!BaseThreadInitThunk+0x14
05 000000a1`c2ccfbb0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

```
0:000> r
rax=0000000000001009 rbx=000000a1c2ccfa40 rcx=000000a1c2ccfa40
rdx=0000000000000000 rsi=0000000000000000 rdi=00007ff7437d0000
rip=00007ffedad31164 rsp=000000a1c2ccf988 rbp=000000a1c2ccfa59
 r8=0000000000000080  r9=0000000000010412 r10=0000000000010412
r11=1151005044840000 r12=0000000000000000 r13=0000000000000000
r14=00007ff7437d0000 r15=0000000000000001
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
win32u!NtUserGetMessage+0x14:
00007ffe`dad31164 c3              ret
```

18. In any situation when we move down to the next frame, for example, to *GetMessageW+0x2d* (which points to the next instruction after *NtUserGetMessage* was called) and to *WinMain+0x293*, we don't have most CPU registers' values saved previously (**r** command gives accurate values only for the topmost frame 0 except RIP and RSP and perhaps a few other registers):

```
0:000> k
 # Child-SP          RetAddr           Call Site
00 000000a1`c2ccf988 00007ffe`dc19477d win32u!NtUserGetMessage+0x14
01 000000a1`c2ccf990 00007ff7`437da3d3 user32!GetMessageW+0x2d
02 000000a1`c2ccf9f0 00007ff7`437f02b7 notepad!WinMain+0x293
03 000000a1`c2ccfac0 00007ffe`dc557bd4 notepad!__mainCRTStartup+0x19f
04 000000a1`c2ccfb80 00007ffe`ddc6cee1 kernel32!BaseThreadInitThunk+0x14
05 000000a1`c2ccfbb0 00000000`00000000 ntdll!RtlUserThreadStart+0x21

0:000> ub 00007ffe`dc19477d
user32!GetMessageW+0x9:
00007ffe`dc194759 488bd9          mov     rbx,rcx
00007ffe`dc19475c 458bc8          mov     r9d,r8d
00007ffe`dc19475f 440bc8          or      r9d,eax
00007ffe`dc194762 41f7c10000feff  test    r9d,0FFFE0000h
00007ffe`dc194769 0f8531db0100    jne     user32!GetMessageW+0x1db50 (00007ffe`dc1b22a0)
00007ffe`dc19476f 448bc8          mov     r9d,eax
00007ffe`dc194772 48897c2460      mov     qword ptr [rsp+60h],rdi
00007ffe`dc194777 ff158b320600    call    qword ptr [user32!_imp_NtUserGetMessage
(00007ffe`dc1f7a08)]

0:000> u 00007ffe`dc19477d
user32!GetMessageW+0x2d:
00007ffe`dc19477d 833d8821080005  cmp     dword ptr [user32!g_systemCallFilterId
(00007ffe`dc21690c)],5
00007ffe`dc194784 8bf8            mov     edi,eax
00007ffe`dc194786 741e            je      user32!GetMessageW+0x56 (00007ffe`dc1947a6)
00007ffe`dc194788 8b4308          mov     eax,dword ptr [rbx+8]
00007ffe`dc19478b 3d02010000      cmp     eax,102h
00007ffe`dc194790 742e            je      user32!GetMessageW+0x70 (00007ffe`dc1947c0)
00007ffe`dc194792 3dcc000000      cmp     eax,0CCh
00007ffe`dc194797 7427            je      user32!Ge

0:000> kn
 # Child-SP          RetAddr           Call Site
00 000000a1`c2ccf988 00007ffe`dc19477d win32u!NtUserGetMessage+0x14
01 000000a1`c2ccf990 00007ff7`437da3d3 user32!GetMessageW+0x2d
02 000000a1`c2ccf9f0 00007ff7`437f02b7 notepad!WinMain+0x293
03 000000a1`c2ccfac0 00007ffe`dc557bd4 notepad!__mainCRTStartup+0x19f
04 000000a1`c2ccfb80 00007ffe`ddc6cee1 kernel32!BaseThreadInitThunk+0x14
05 000000a1`c2ccfbb0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

```
0:000> r
rax=0000000000001009 rbx=000000a1c2ccfa40 rcx=000000a1c2ccfa40
rdx=0000000000000000 rsi=0000000000000000 rdi=00007ff7437d0000
rip=00007ffedad31164 rsp=000000a1c2ccf988 rbp=000000a1c2ccfa59
 r8=0000000000000080  r9=0000000000010412 r10=0000000000010412
r11=1151005044840000 r12=0000000000000000 r13=0000000000000000
r14=00007ff7437d0000 r15=0000000000000001
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
win32u!NtUserGetMessage+0x14:
00007ffe`dad31164 c3              ret

0:000> .frame /c 1
01 000000a1`c2ccf990 00007ff7`437da3d3 user32!GetMessageW+0x2d
rax=0000000000001009 rbx=000000a1c2ccfa40 rcx=000000a1c2ccfa40
rdx=0000000000000000 rsi=0000000000000000 rdi=00007ff7437d0000
rip=00007ffedc19477d rsp=000000a1c2ccf990 rbp=000000a1c2ccfa59
 r8=0000000000000080  r9=0000000000010412 r10=0000000000010412
r11=1151005044840000 r12=0000000000000000 r13=0000000000000000
r14=00007ff7437d0000 r15=0000000000000001
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
user32!GetMessageW+0x2d:
00007ffe`dc19477d 833d8821080005  cmp     dword ptr [user32!g_systemCallFilterId
(00007ffe`dc21690c)],5 ds:00007ffe`dc21690c=00000000

0:000> .frame /c 2
02 000000a1`c2ccf9f0 00007ff7`437f02b7 notepad!WinMain+0x293
rax=0000000000001009 rbx=000000000006038b rcx=000000a1c2ccfa40
rdx=0000000000000000 rsi=0000000000000000 rdi=00007ff7437d0000
rip=00007ff7437da3d3 rsp=000000a1c2ccf9f0 rbp=000000a1c2ccfa59
 r8=0000000000000080  r9=0000000000010412 r10=0000000000010412
r11=1151005044840000 r12=0000000000000000 r13=0000000000000000
r14=00007ff7437d0000 r15=0000000000000001
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
notepad!WinMain+0x293:
00007ff7`437da3d3 85c0            test    eax,eax

0:000> k
  *** Stack trace for last set context - .thread/.cxr resets it
 # Child-SP          RetAddr           Call Site
02 000000a1`c2ccf9f0 00007ff7`437f02b7 notepad!WinMain+0x293
03 000000a1`c2ccfac0 00007ffe`dc557bd4 notepad!__mainCRTStartup+0x19f
04 000000a1`c2ccfb80 00007ffe`ddc6cee1 kernel32!BaseThreadInitThunk+0x14
05 000000a1`c2ccfbb0 00000000`00000000 ntdll!RtlUserThreadStart+0x21

0:000> .cxr
Resetting default scope

0:000> k
 # Child-SP          RetAddr           Call Site
00 000000a1`c2ccf988 00007ffe`dc19477d win32u!NtUserGetMessage+0x14
01 000000a1`c2ccf990 00007ff7`437da3d3 user32!GetMessageW+0x2d
02 000000a1`c2ccf9f0 00007ff7`437f02b7 notepad!WinMain+0x293
03 000000a1`c2ccfac0 00007ffe`dc557bd4 notepad!__mainCRTStartup+0x19f
04 000000a1`c2ccfb80 00007ffe`ddc6cee1 kernel32!BaseThreadInitThunk+0x14
05 000000a1`c2ccfbb0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

19. Some CPU registers can be recovered manually, such as RIP (saved address when using *call* instruction) and RSP (the stack pointer value that was before saving that RIP address). Other register values can be recovered manually, too, if they were not used in called frames or were saved in temporary memory cells (such as on stack). Let's recover some registers for the first few frames.

```
0:000> r
rax=0000000000001009 rbx=000000a1c2ccfa40 rcx=000000a1c2ccfa40
rdx=0000000000000000 rsi=0000000000000000 rdi=00007ff7437d0000
rip=00007ffedad31164 rsp=000000a1c2ccf988 rbp=000000a1c2ccfa59
 r8=0000000000000080  r9=0000000000010412 r10=0000000000010412
r11=1151005044840000 r12=0000000000000000 r13=0000000000000000
r14=00007ff7437d0000 r15=0000000000000001
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
win32u!NtUserGetMessage+0x14:
00007ffe`dad31164 c3              ret
```

Let's disassemble the current function:

```
0:000> uf win32u!NtUserGetMessage
win32u!NtUserGetMessage:
00007ffe`dad31150 4c8bd1          mov     r10,rcx
00007ffe`dad31153 b809100000      mov     eax,1009h
00007ffe`dad31158 f604250803fe7f01 test    byte ptr [SharedUserData+0x308
(00000000`7ffe0308)],1
00007ffe`dad31160 7503            jne     win32u!NtUserGetMessage+0x15 (00007ffe`dad31165)
Branch

win32u!NtUserGetMessage+0x12:
00007ffe`dad31162 0f05            syscall
00007ffe`dad31164 c3              ret

win32u!NtUserGetMessage+0x15:
00007ffe`dad31165 cd2e            int     2Eh
00007ffe`dad31167 c3              ret
```

It is a short function. We see it overwrites R10 and EAX. Note that R10 and RCX values are different in the output of **r** command:

```
0:000> r @r10
r10=0000000000010412

0:000> r @rcx
rcx=000000a1c2ccfa40
```

We see that RSP is not used inside the *NtUserGetMessage* function, and its value should point to the return address of the caller, *GetMessageW* function during the execution of **call** instruction:

```
0:000> dp @rsp
000000a1`c2ccf988  00007ffe`dc19477d 00000000`00007f48
000000a1`c2ccf998  00000000`0001e327 000019ee`00000000
000000a1`c2ccf9a8  00007ff7`00000001 00000000`00000001
000000a1`c2ccf9b8  00007ff7`437d0000 00000000`00000001
000000a1`c2ccf9c8  00007ff7`437d0000 00000000`00000000
000000a1`c2ccf9d8  00000000`00000000 00000000`0006038b
000000a1`c2ccf9e8  00007ff7`437da3d3 00007ff7`437d0000
000000a1`c2ccf9f8  00000000`0006038b 00000000`00000000
```

```
0:000> ub 00007ffe`dc19477d
user32!GetMessageW+0x9:
00007ffe`dc194759 488bd9          mov     rbx,rcx
00007ffe`dc19475c 458bc8          mov     r9d,r8d
00007ffe`dc19475f 440bc8          or      r9d,eax
00007ffe`dc194762 41f7c10000feff  test    r9d,0FFFE0000h
00007ffe`dc194769 0f8531db0100    jne     user32!GetMessageW+0x1db50 (00007ffe`dc1b22a0)
00007ffe`dc19476f 448bc8          mov     r9d,eax
00007ffe`dc194772 48897c2460      mov     qword ptr [rsp+60h],rdi
00007ffe`dc194777 ff158b320600    call    qword ptr [user32!_imp_NtUserGetMessage
(00007ffe`dc1f7a08)]

0:000> u 00007ffe`dc19477d
user32!GetMessageW+0x2d:
00007ffe`dc19477d 833d8821080005  cmp     dword ptr [user32!g_systemCallFilterId
(00007ffe`dc21690c)],5
00007ffe`dc194784 8bf8            mov     edi,eax
00007ffe`dc194786 741e            je      user32!GetMessageW+0x56 (00007ffe`dc1947a6)
00007ffe`dc194788 8b4308          mov     eax,dword ptr [rbx+8]
00007ffe`dc19478b 3d02010000      cmp     eax,102h
00007ffe`dc194790 742e            je      user32!GetMessageW+0x70 (00007ffe`dc1947c0)
00007ffe`dc194792 3dcc000000      cmp     eax,0CCh
00007ffe`dc194797 7427            je      user32!GetMessageW+0x70 (00007ffe`dc1947c0)
```

This is the RIP value saved before the call, but RSP should be the value before the **call** instruction was executed. When a return value is saved, RSP is decremented by 8, so the value of RSP before the call should be the value of RSP pointing to the saved return address + 8:

```
0:000> ? @rsp + 8
Evaluate expression: 694757947792 = 000000a1`c2ccf990

0:000> k
# Child-SP          RetAddr           Call Site
00 000000a1`c2ccf988 00007ffe`dc19477d win32u!NtUserGetMessage+0x14
01 000000a1`c2ccf990 00007ff7`437da3d3 user32!GetMessageW+0x2d
02 000000a1`c2ccf9f0 00007ff7`437f02b7 notepad!WinMain+0x293
03 000000a1`c2ccfac0 00007ffe`dc557bd4 notepad!__mainCRTStartup+0x19f
04 000000a1`c2ccfb80 00007ffe`ddc6cee1 kernel32!BaseThreadInitThunk+0x14
05 000000a1`c2ccfbb0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

Let's now find out RIP and RSP for the next frame (the caller of *GetMessageW* function). To find out RSP we need to see how it was used in the callee, *GetMessageW* function before the callee called *NtUserGetMessage*. We disassemble *GetMessageW* function:

```
0:000> uf user32!GetMessageW
user32!GetMessageW:
00007ffe`dc194750 4053            push    rbx
00007ffe`dc194752 4883ec50        sub     rsp,50h
00007ffe`dc194756 418bc1          mov     eax,r9d
00007ffe`dc194759 488bd9          mov     rbx,rcx
00007ffe`dc19475c 458bc8          mov     r9d,r8d
00007ffe`dc19475f 440bc8          or      r9d,eax
00007ffe`dc194762 41f7c10000feff  test    r9d,0FFFE0000h
00007ffe`dc194769 0f8531db0100    jne     user32!GetMessageW+0x1db50 (00007ffe`dc1b22a0)
Branch

user32!GetMessageW+0x1f:
00007ffe`dc19476f 448bc8          mov     r9d,eax
```

```
00007ffe`dc194772 48897c2460      mov     qword ptr [rsp+60h],rdi
00007ffe`dc194777 ff158b320600    call    qword ptr [user32!_imp_NtUserGetMessage
(00007ffe`dc1f7a08)]
00007ffe`dc19477d 833d8821080005  cmp     dword ptr [user32!g_systemCallFilterId
(00007ffe`dc21690c)],5
00007ffe`dc194784 8bf8            mov     edi,eax
00007ffe`dc194786 741e            je      user32!GetMessageW+0x56 (00007ffe`dc1947a6)  Branch

user32!GetMessageW+0x38:
00007ffe`dc194788 8b4308          mov     eax,dword ptr [rbx+8]
00007ffe`dc19478b 3d02010000      cmp     eax,102h
00007ffe`dc194790 742e            je      user32!GetMessageW+0x70 (00007ffe`dc1947c0)  Branch

user32!GetMessageW+0x42:
00007ffe`dc194792 3dcc000000      cmp     eax,0CCh
00007ffe`dc194797 7427            je      user32!GetMessageW+0x70 (00007ffe`dc1947c0)  Branch

user32!GetMessageW+0x49:
00007ffe`dc194799 8bc7            mov     eax,edi
00007ffe`dc19479b 488b7c2460      mov     rdi,qword ptr [rsp+60h]

user32!GetMessageW+0x50:
00007ffe`dc1947a0 4883c450        add     rsp,50h
00007ffe`dc1947a4 5b              pop     rbx
00007ffe`dc1947a5 c3              ret

user32!GetMessageW+0x56:
00007ffe`dc1947a6 65488b042530000000 mov   rax,qword ptr gs:[30h]
00007ffe`dc1947af 488b88f8080000  mov     rcx,qword ptr [rax+8F8h]
00007ffe`dc1947b6 4885c9          test    rcx,rcx
00007ffe`dc1947b9 74cd            je      user32!GetMessageW+0x38 (00007ffe`dc194788)  Branch

user32!GetMessageW+0x6b:
00007ffe`dc1947bb e907db0100      jmp     user32!GetMessageW+0x1db77 (00007ffe`dc1b22c7)
Branch

user32!GetMessageW+0x70:
00007ffe`dc1947c0 48816310ffff0000 and     qword ptr [rbx+10h],0FFFFh
00007ffe`dc1947c8 ebcf            jmp     user32!GetMessageW+0x49 (00007ffe`dc194799)  Branch

user32!GetMessageW+0x1db50:
00007ffe`dc1b22a0 83f8ff          cmp     eax,0FFFFFFFFh
00007ffe`dc1b22a3 7510            jne     user32!GetMessageW+0x1db65 (00007ffe`dc1b22b5)
Branch

user32!GetMessageW+0x1db55:
00007ffe`dc1b22a5 41f7c00000feff  test    r8d,0FFFE0000h
00007ffe`dc1b22ac 7507            jne     user32!GetMessageW+0x1db65 (00007ffe`dc1b22b5)
Branch

user32!GetMessageW+0x1db5e:
00007ffe`dc1b22ae 33c0            xor     eax,eax
00007ffe`dc1b22b0 e9ba24feff      jmp     user32!GetMessageW+0x1f (00007ffe`dc19476f)  Branch

user32!GetMessageW+0x1db65:
00007ffe`dc1b22b5 b957000000      mov     ecx,57h
00007ffe`dc1b22ba ff1548530400    call    qword ptr [user32!_imp_RtlSetLastWin32Error
(00007ffe`dc1f7608)]
00007ffe`dc1b22c0 33c0            xor     eax,eax
00007ffe`dc1b22c2 e9d924feff      jmp     user32!GetMessageW+0x50 (00007ffe`dc1947a0)  Branch
```

```
user32!GetMessageW+0x1db77:
00007ffe`dc1b22c7 4883791000      cmp     qword ptr [rcx+10h],0
00007ffe`dc1b22cc 0f84b624feff    je      user32!GetMessageW+0x38 (00007ffe`dc194788)  Branch

user32!GetMessageW+0x1db82:
00007ffe`dc1b22d2 8b4b08          mov     ecx,dword ptr [rbx+8]
00007ffe`dc1b22d5 4c8bcb          mov     r9,rbx
00007ffe`dc1b22d8 8d81c0fdffff    lea     eax,[rcx-240h]
00007ffe`dc1b22de 85c0            test    eax,eax
00007ffe`dc1b22e0 7408            je      user32!GetMessageW+0x1db9a (00007ffe`dc1b22ea)
Branch

user32!GetMessageW+0x1db92:
00007ffe`dc1b22e2 81f919010000    cmp     ecx,119h
00007ffe`dc1b22e8 7505            jne     user32!GetMessageW+0x1db9f (00007ffe`dc1b22ef)
Branch

user32!GetMessageW+0x1db9a:
00007ffe`dc1b22ea 4c8d4c2420      lea     r9,[rsp+20h]

user32!GetMessageW+0x1db9f:
00007ffe`dc1b22ef 33d2            xor     edx,edx
00007ffe`dc1b22f1 8d4a03          lea     ecx,[rdx+3]
00007ffe`dc1b22f4 448d4201        lea     r8d,[rdx+1]
00007ffe`dc1b22f8 e81b75fdff      call    user32!CLocalHookManager::RunHookChain
(00007ffe`dc189818)
00007ffe`dc1b22fd 90              nop
00007ffe`dc1b22fe e98524feff      jmp     user32!GetMessageW+0x38 (00007ffe`dc194788)  Branch
```

We see that the stack pointer was decremented by 0x50 (*sub* instruction) and also by 8 (*push* instruction), and so we add these values to RSP we found out previously for the *NtUserGetMessage* call, **000000a1`c2ccf990**:

```
0:000> dps 000000a1`c2ccf990 + 50 + 8
000000a1`c2ccf9e8  00007ff7`437da3d3 notepad!WinMain+0x293
000000a1`c2ccf9f0  00007ff7`437d0000 notepad!TlgWrite <PERF> (notepad+0x0)
000000a1`c2ccf9f8  00000000`0006038b
000000a1`c2ccfa00  00000000`00000000
000000a1`c2ccfa08  00000000`00000000
000000a1`c2ccfa10  00000000`00000740
000000a1`c2ccfa18  00000221`00000000
000000a1`c2ccfa20  00000000`00000000
000000a1`c2ccfa28  00007ff7`437f0670 notepad!onexit+0x28
000000a1`c2ccfa30  00000000`00000000
000000a1`c2ccfa38  00000003`00000000
000000a1`c2ccfa40  00000000`00000000
000000a1`c2ccfa48  00000000`00000113
000000a1`c2ccfa50  00000000`00007f48
000000a1`c2ccfa58  00007ffe`dc4166d0 msctf!CThreadInputMgr::TimerProc
000000a1`c2ccfa60  00000461`0001e327
```

We see that *GetMessageW* was called from the *WinMain* function:

```
0:000> ub 00007ff7`437da3d3
notepad!WinMain+0x271:
00007ff7`437da3b1 ff1589850100    call    qword ptr [notepad!_imp_TranslateMessage
(00007ff7`437f2940)]
00007ff7`437da3b7 488d4de7        lea     rcx,[rbp-19h]
```

```
00007ff7`437da3bb ff1587850100    call    qword ptr [notepad!_imp_DispatchMessageW
(00007ff7`437f2948)]
00007ff7`437da3c1 4533c9          xor     r9d,r9d
00007ff7`437da3c4 488d4de7        lea     rcx,[rbp-19h]
00007ff7`437da3c8 4533c0          xor     r8d,r8d
00007ff7`437da3cb 33d2            xor     edx,edx
00007ff7`437da3cd ff1555850100    call    qword ptr [notepad!_imp_GetMessageW
(00007ff7`437f2928)]
```

The value of RSP before the call should be adjusted by 8 because of the saved return address:

```
0:000> ? 000000a1`c2ccf9e8 + 8
Evaluate expression: 694757947888 = 000000a1`c2ccf9f0

0:000> k
 # Child-SP          RetAddr           Call Site
00 000000a1`c2ccf988 00007ffe`dc19477d win32u!NtUserGetMessage+0x14
01 000000a1`c2ccf990 00007ff7`437da3d3 user32!GetMessageW+0x2d
02 000000a1`c2ccf9f0 00007ff7`437f02b7 notepad!WinMain+0x293
03 000000a1`c2ccfac0 00007ffe`dc557bd4 notepad!__mainCRTStartup+0x19f
04 000000a1`c2ccfb80 00007ffe`ddc6cee1 kernel32!BaseThreadInitThunk+0x14
05 000000a1`c2ccfbb0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

We can reconstruct the stack trace like a debugger. Note that we can correctly disassemble functions using the **uf** command because function boundaries are saved in PDB symbol files, or the start of the function is available from the image file as an exported function. If such information is not available, we would most likely have a truncated stack trace.

20. Other registers and memory values are reused and overwritten when we move down the frames so less and less information can be recovered. We call this ADDR pattern (Inverse) **Context Pyramid**.

21. We also introduce special **Stack Frame** memory cell diagrams. For example, the case of the stack frame for the *GetMessageW* function before calling *NtUserGetMessage* is illustrated in the MCD-R1.xlsx section D, where [RSP+58] corresponds to the stored return address of the *GetMessageW* caller.

22. We close logging before exiting WinDbg:

```
0:000> .logclose
Closing open log file C:\ADDR\MemoryDumps\R1.log
```

Note: To avoid possible confusion and glitches, we recommend exiting WinDbg after each exercise.