Contents

About the Author .......................................................................................................................... 5
Presentation Slides and Transcript .............................................................................................. 7
Review of x64 Disassembly ......................................................................................................... 35
Practice Exercises ......................................................................................................................... 47

Exercise 0: Download, setup, and verify your WinDbg or Debugging Tools for Windows installation, or
Docker Debugging Tools for Windows image ................................................................................... 52
Exercise P1: Analysis of a normal application process dump (64-bit wordpad) .................................. 66
Exercise P2: Analysis of a normal application process dump (32-bit wordpad) ................................. 77
Exercise P3: Analysis of a normal application process dump (64-bit Microsoft Edge) ....................... 80
Exercise P4: Analysis of an application process dump (64-bit AppK, no symbols) ......................... 91
Exercise P5: Analysis of an application process dump (64-bit AppK, with application symbols) ......... 101
Exercise P6: Analysis of an application process dump (AppL, 64-bit) ............................................. 106
Exercise P7: Analysis of an application process dump (AppL2, 64-bit) ........................................... 116
Exercise P8: Analysis of an application process dump (AppM, 64-bit) ............................................ 130
Exercise P9: Analysis of an application process dump (AppN, 64-bit) ........................................... 140
Exercise P10: Analysis of an application process dump (AppO, 64-bit) ........................................... 150
Exercise P11: Analysis of an application process dump (AppP, 64-bit) ........................................... 159
Exercise P12: Analysis of an application process dump (AppR2, 64-bit) ........................................ 171
Exercise P13: Analysis of an application process dump (AppA, WOW64) ........................................ 185
Exercise P14: Analysis of an application process dump (AppS, 64-bit) ........................................... 204
Exercise P15: Analysis of an application process dump (notepad, 32-bit) ......................................... 224
Exercise P16: Analysis of an application process dump (notepad, 64-bit) ......................................... 229
Exercise P17: Analysis of an application process dump (AppQ, 32-bit) ........................................... 237
Exercise P18: Analysis of an application process dump (AppQ, 64-bit) .......................................... 249
Exercise P19: Analysis of an application process dump (AppT, 64-bit) ........................................... 259
Exercise P20: Analysis of a service process dump (ServiceA, 64-bit) ............................................. 274
Exercise P21: Analysis of a Rust process dump (Rusty, 64-bit) ..................................................... 277

Application Source Code ............................................................................................................. 287
AppA ............................................................................................................................................ 289
AppK ............................................................................................................................................ 291
AppL ............................................................................................................................................ 292
AppL2 .......................................................................................................................................... 293
AppM ............................................................................................................................................ 294
AppN ............................................................................................................................................ 295
Exercise P1: Analysis of a normal application process dump (64-bit wordpad)

Goal: Learn how to see dump file type and version, get a stack trace, check its correctness, perform default analysis, list threads and modules, check module version information, dump module data, and check the process environment.

Patterns: Manual Dump (Process); Stack Trace; Not My Version (Software); Environment Hint; Unknown Component.

1. Launch WinDbg.
2. Open \AWMDA-Dumps\Process\x64\wordpad.DMP.
3. We get the dump file loaded:

Microsoft (R) Windows Debugger Version 10.0.25877.1004 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Loading Dump File [C:\AWMDA-Dumps\Process\x64\wordpad.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 22000 MP (2 procs) Free x64
Product: WinNt, suite: SingleUserTS Personal
Edition build lab: 22000.1.amd64fre.co_release.210604-1628
Debug session time: Sat Jul 15 18:04:32.000 2023 (UTC + 1:00)
System Uptime: 0 days 0:21:14.009
Process Uptime: 0 days 0:04:51.000

For analysis of this file, run !analyze -v
win32u!NtUserGetMessage+0x14:
00007ff9`13c31414 c3 ret

4. Open a log file to save all future output using the .logopen command:

@0:000> .logopen C:\AWMDA-Dumps\Process\x64\wordpad.log
Opened log file 'C:\AWMDA-Dumps\Process\x64\wordpad.log'
5. Type `k` command to verify the correctness of the stack trace:

<table>
<thead>
<tr>
<th>#</th>
<th>Child-SP</th>
<th>RetAddr</th>
<th>Call Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000037'1567fd48 00007ff9`15a6464e</td>
<td>win32u!NTUserGetMessage+0x14</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00000037'1567fd50 00007ff8`d3150813</td>
<td>user32!GetMessageW+0x2e</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00000037'1567fd60 00007ff9`15a6464e</td>
<td>mfc42u!CWinThread::PumpMessage+0x23</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00000037'1567fd70 00007ff8`d3150736</td>
<td>mfc42u!CWinThread::Run+0x96</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00000037'1567fe20 00007ff6`d9eebcfd</td>
<td>mfc42u!AfxWinMain+0xbc</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00000037'1567fe60 00007ff9`14a754e0</td>
<td>wordpad!_wmainCRTStartup+0x1dd</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00000037'1567ff20 00007ff9`1668485b</td>
<td>kernel32!BaseThreadInitThunk+0x1bb</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00000037'1567fe60 00007ff9`14a754e0</td>
<td>ntdll!RtlUserThreadStart+0x2b</td>
<td></td>
</tr>
</tbody>
</table>

6. Type the `version` command to get the OS version, system and process uptimes, the dump file timestamp, and its type:

```
0:000> version
Windows 10 Version 22000 MP (2 procs) Free x64
Product: WinNt, suite: SingleUserTS Personal
Edition build lab: 22000.1.amd64fre.co_release.210604-1628
Debug session time: Sat Jul 15 18:04:32.000 2023 (UTC + 1:00)
System Uptime: 0 days 0:21:14.009
Process Uptime: 0 days 0:04:51.000
Kernel time: 0 days 0:00:01.000
User time: 0 days 0:00:00.000
Full memory user mini dump: C:\AWMDA-Dumps\Process\x64\wordpad.DMP
```

Note: Debug session time is when the dump was generated. Although the dump is called a “mini dump,” it is a full memory user dump with all process memory included.
7. Type the default analysis command `!analyze -v`:

Note: This command may take some time initially as symbols may be downloaded from the symbol server:

```
0:000> !analyze -v
*******************************************************************************
*                                                                             *
*                        Exception Analysis                                 *
*                                                                             *
*******************************************************************************
KEY_VALUES_STRING: 1
   Key  : Analysis.CPU.mSec  Value: 782
   Key  : Analysis.Elapsed.mSec Value: 8353
   Key  : Analysis.IO.Other.Mb Value: 13
   Key  : Analysis.IO.Read.Mb Value: 0
   Key  : Analysis.IO.Write.Mb Value: 27
   Key  : Analysis.Init.CPU.mSec Value: 171
   Key  : Analysis.Init.Elapsed.mSec Value: 338539
   Key  : Analysis.Memory.CommitPeak.Mb Value: 145
   Key  : Failure.Hash Value: {3112b5eb-303b-e877-0655-90bdfa336126}
   Key  : Timeline.OS.Boot.DeltaSec Value: 1274
   Key  : WER.OS.Branch Value: co_release
   Key  : WER.OS.Version Value: 10.0.22000.1
   Key  : WER.Process.Version Value: 10.0.22000.1
FILE_IN_CAB:  wordpad.DMP
NTGLOBALFLAG:  400
APPLICATION_VERIFIER_FLAGS:  0
EXCEPTION_RECORD:  (.exr-1)
ExceptionAddress: 0000000000000000
ExceptionCode: 80000003 (Break instruction exception)
ExceptionFlags: 00000000
NumberParameters: 0
FAULTING_THREAD:  000002d8
PROCESS_NAME:  wordpad.exe
ERROR_CODE: (NTSTATUS) 0x80000003 - [EXCEPTION] Breakpoint A breakpoint has been reached.
EXCEPTION_CODE_STR:  80000003
STACK_TEXT:  00000037 1567f40 00007ff8 15a646ae 00007ff8 d32a460 00007ff8 d314b1b 00007ff8 d32a460 00007ff8 d314b4b 00007ff8 d315073c 00007ff6 e877-0655-90bdfa336126
STACK_COMMAND:  ~0s; .ecxr ; kb
SYMBOL_NAME:  win32u!NtUserGetMessage+14
MODULE_NAME: win32u
```
Note: "Break instruction exception" can be the sign of a Manual Dump pattern, but often WinDbg is not able to figure out an exception that may be on another thread or hidden. STACK_COMMAND shows the sequence commands that WinDbg executed to get STACK_TEXT.

8. Now we check how many threads there are by using the ~ command:

```
0:000> ~
  0: Id: 118c.2d8 Suspend: 0 Teb: 00000037`15400000 Unfrozen
     1 Id: 118c.1064 Suspend: 0 Teb: 00000037`15408000 Unfrozen
     2 Id: 118c.2108 Suspend: 0 Teb: 00000037`1540a000 Unfrozen
     3 Id: 118c.50c Suspend: 0 Teb: 00000037`1540c000 Unfrozen
```

Note: 118c is Process ID (PID), and 2d8 is Thread ID (TID). 118c.2d8 is called CID (Client ID).

9. Now we dump a stack trace using the kc command (only modules and symbols):

```
0:000> kc
  # Call Site
  00 win32u!NtUserGetMessage
  01 user32!GetMessageW
  02 mfc42u!CWinThread::PumpMessage
  03 mfc42u!CWinThread::Run
  04 mfc42u!AfxWinMain
  05 wordpad!__wmainCRTStartup
  06 kernel32!BaseThreadInitThunk
  07 ntdll!RtlUserThreadStart
```

10. Now we dump the stack trace of the current thread using the k command (with symbols, return addresses, and function offsets):

```
0:000> k
  # Child-SP RetAddr Call Site
  00 00000037`1567fd48 00007ff9`15a6464e win32u!NtUserGetMessage+0x14
  01 00000037`1567fd50 00007ff8`d3150813 user32!GetMessageW+0x2e
  02 00000037`1567fd80 00007ff8`d3150736 mfc42u!CWinThread::PumpMessage+0x23
  03 00000037`1567fe00 00007ff8`d314f2bc mfc42u!CWinThread::Run+0x96
  04 00000037`1567fe20 00007ff6`d9ebebcfd mfc42u!AfxWinMain+0xbc
  05 00000037`1567fe60 00007ff9`14a754e0 wordpad!__wmainCRTStartup+0x1dd
  06 00000037`1567ff20 00007ff9`1668485b kernel32!BaseThreadInitThunk+0x10
  07 00000037`1567ff50 00000000`00000000 ntdll!RtlUserThreadStart+0x2b
```

Hint: How to check that the stack trace is correct. Use the ub command (unassemble backward) to check if there is a call instruction. We check that the GetMessageW function was called from the CWinThread::PumpMessage function:
Then we check that the `NtUserGetMessage` function was called from the `GetMessageW` function:

```plaintext
0:000> k
# Child-SP RetAddr Call Site
00 00000037`1567fd48 00007ff8`15a6464e win32u!NtUserGetMessage+0x14
01 00000037`1567fd50 00007ff8`d3150813 user32!GetMessageW+0x2e
02 00000037`1567fd60 00007ff8`d3150736 mfc42u!CWinThread::PumpMessage+0x23
03 00000037`1567fe00 00007ff8`d314f2bc mfc42u!CWinThread::Run+0x96
04 00000037`1567fe20 00007ff6`d9eebcfd mfc42u!AfxWinMain+0xbc
05 00000037`1567fe60 00007ff9`14a754e0 wordpad!__wmainCRTStartup+0x1dd
06 00000037`1567ff20 00007ff9`1668485b kernel32!BaseThreadInitThunk+0x10
07 00000037`1567ff50 00000000`00000000 ntdll!RtlUserThreadStart+0x2b
```

Note: Remember the functions call each other from bottom to top. The topmost function from the stack trace is the last one that was called. `ExceptionAddress` may point to the last one. We will come to this in the real exception process dumps later.

11. Now we check the list of loaded modules using the `lm` command:

```plaintext
0:000> lm
start          end          module name          wdpad (pdb symbols)
00007ff8`d1508013 00007ff8`1567d6c0 mfc42u!CWinThread::PumpMessage+0x9:
00007ff8`d1508000 00007ff8`1567d6c0 mfc42u!CWinThread::PumpMessage+0x9:
```

Then we check that the `NtUserGetMessage` function was called from the `GetMessageW` function:

```plaintext
0:000> k
# Child-SP RetAddr Call Site
00 00000037`1567fd48 00007ff8`15a6464e win32u!NtUserGetMessage+0x14
01 00000037`1567fd50 00007ff8`d3150813 user32!GetMessageW+0x2e
02 00000037`1567fd60 00007ff8`d3150736 mfc42u!CWinThread::PumpMessage+0x23
03 00000037`1567fe00 00007ff8`d314f2bc mfc42u!CWinThread::Run+0x96
04 00000037`1567fe20 00007ff6`d9eebcfd mfc42u!AfxWinMain+0xbc
05 00000037`1567fe60 00007ff9`14a754e0 wordpad!__wmainCRTStartup+0x1dd
06 00000037`1567ff20 00007ff9`1668485b kernel32!BaseThreadInitThunk+0x10
07 00000037`1567ff50 00000000`00000000 ntdll!RtlUserThreadStart+0x2b
```

Note: Remember the functions call each other from bottom to top. The topmost function from the stack trace is the last one that was called. `ExceptionAddress` may point to the last one. We will come to this in the real exception process dumps later.

11. Now we check the list of loaded modules using the `lm` command:

```plaintext
0:000> lm
start          end          module name          wdpad (pdb symbols)
```
The image contains a memory dump with addresses and module names, indicating the use of debugging tools like WinDbg. The text is a representation of the process memory space, showing modules loaded into memory, their file names, and paths. The layout suggests a debugger's output, typically used in system debugging to understand the system's state at a specific point.

**Note:** The start and end addresses show where modules are loaded in process virtual memory. You can see the module contents by using the `dc` command (Unknown Component pattern).
12. We can check verbose module information using the `lmv` command or use `lmv m <module name>` to check an individual module (Not My Version pattern):

```
0:000> lmv m wordpad
Browse full module list
start    end    module name
00007ff6`d9ee0000 00007ff6`da1cd000 wordpad (pdb symbols)
```

```
C:\WinDbg.Docker.AWMDA6\mss\wordpad.pdb\B193BA11D609CB39E8D086A748A191651\wordpad.pdb
Loaded symbol image file: wordpad.exe
Image path: C:\Program Files\Windows NT\Accessories\wordpad.exe
Image name: wordpad.exe
Browse all global symbols functions data
Image was built with /Brepro flag.
Timestamp: B930DF5E (This is a reproducible build file hash, not a timestamp)
CheckSum: 002F073E
ImageSize: 002ED000
File version: 10.0.22000.1
Product version: 10.0.22000.1
File flags: 0 (Mask 3F)
File OS: 40004 NT Win32
File type: 1.0 App
File date: 00000000.00000000
Translations: 0409.04b0
Information from resource tables:
  CompanyName: Microsoft Corporation
  ProductName: Microsoft® Windows® Operating System
  InternalName: wordpad
  OriginalFilename: WORDPAD.EXE
  ProductVersion: 10.0.22000.1
  FileVersion: 10.0.22000.1 (WinBuild.160101.0800)
```
13. Sometimes `lmv` command doesn't show much and `llmi` command might give extra information:

```
0:000> !lmi wordpad
Loaded Module Info: [wordpad]
  Module: wordpad
  Base Address: 0007 ff6d9ee0000
  Image Name: wordpad.exe
  Machine Type: 34404 (X64)
  Time Stamp: b930df5e (This is a reproducible build hash, not a true timestamp)
  Size: 2ed000
  CheckSum: 2f073e
  Characteristics: 22
  Debug Data Dirs: Type Size VA Pointer
    CODEVIEW 24, e7b2c, e7b2c RSDS - GUID: {B193BA11-D609-CB39-E8D0-86A748A19165}
    Age: 1, Pdb: wordpad.pdb
    POGO 48c, e7b50, e7b50 [Data not mapped]
    REPRO 24, e7fdc, e7fdc Reproducible build
  Image Type: MEMORY - Image read successfully from loaded memory.
  Symbol Type: PDB - Symbols loaded successfully from symbol server.
C:\WinDbg.Docker.AWMDA6\mss\wordpad.pdb\B193BA11D609CB39E8D086A748A191651\wordpad.pdb
Load Report: public symbols , not source indexed
C:\WinDbg.Docker.AWMDA6\mss\wordpad.pdb\B193BA11D609CB39E8D086A748A191651\wordpad.pdb
```

Note: We can also use the `lmt` command variant if we are interested in timestamps only.

14. Sometimes `Environment Hint` pattern can give troubleshooting suggestions related to environment variables and DLL paths. `!peb` command (Process Environment Block):

```
0:000> !peb
PEB at 0000037155ff0000
InheritedAddressSpace: No
ReadImageFileExecutionOptions: No
BeingDebugged: No
ImageBaseAddress: 00007ff6d9ee0000
NtGlobalFlag: 400
NtGlobalFlag2: 0
Ldr 00007ff9167fa120
Ldr.Initialized: Yes
Ldr.InInitializationOrderModuleList: 000001649e3d1fd0 . 00000164a097d4b0
Ldr.InLoadOrderModuleList: 000001649e3d2150 . 00000164a097d4c0
Ldr.InMemoryOrderModuleList: 000001649e3d2160 . 00000164a097d4e0
Ldr.TimeStamp: Module
```

Note: We can also use the `lmt` command variant if we are interested in timestamps only.
To launch classic help from the WinDbg app, type the `.hh` command.

We close logging before exiting WinDbg:

```
0:000> .logclose
Closing open log file C:\AWMDA-Dumps\Process\x64\wordpad.log
```

**Note:** If you close a log and later reopen it using the `.logopen` command, its contents will be lost. To append new output to an already existing log please use `.logappend` WinDbg command.