Linux Debugging Accelerated

Dmitry Vostokov
Software Diagnostics Services
Contents

About the Author ................................................................................................................................. 5
Presentation Slides and Transcript ........................................................................................................ 7
Review of x64 Disassembly (GDB, LLDB) .......................................................................................... 41
Review of ARM64 Disassembly ............................................................................................................ 51
Review of x64 Disassembly (WinDbg) .................................................................................................. 61
Practice Exercises ............................................................................................................................... 73
  Exercise UD0 ..................................................................................................................................... 79
  Exercise UD1 (WinDbg) ..................................................................................................................... 90
  Exercise UD1 (GDB) ......................................................................................................................... 101
  Exercise UD1 (LLDB) ....................................................................................................................... 107
  Exercise UD2 (GDB) ......................................................................................................................... 115
  Exercise UD2 (LLDB) ....................................................................................................................... 124
  Exercise UD3 (WinDbg) .................................................................................................................... 133
  Exercise UD3 (GDB) ......................................................................................................................... 152
  Exercise UD4 (WinDbg) .................................................................................................................... 164
  Exercise UD4 (GDB) ......................................................................................................................... 172
  Exercise UD4 (LLDB) ....................................................................................................................... 178
  Exercise UD5 (WinDbg) .................................................................................................................... 186
  Exercise UD5 (GDB) ......................................................................................................................... 203
  Exercise UD6 (GDB) ......................................................................................................................... 224
  Exercise UD7 (GDB) ......................................................................................................................... 229
  Exercise UD7 (LLDB) ....................................................................................................................... 233
  Exercise KD0 ..................................................................................................................................... 239
  Exercise KD8 ..................................................................................................................................... 254
  Exercise KD10 ................................................................................................................................. 282
  Exercise MD9 ..................................................................................................................................... 322
  Exercise TD5 ..................................................................................................................................... 333
  Exercise RD11 (WinDbg) ................................................................................................................... 341
  Exercise RD11 (GDB) ....................................................................................................................... 348
  Exercise RD11 (LLDB) ..................................................................................................................... 352
Exercise UD1 (WinDbg)

Goal: Learn how code generation parameters can influence process execution behavior.

Elementary Diagnostics Patterns: Crash.

Memory Analysis Patterns: Exception Stack Trace; NULL Pointer (Code); Constant Subtrace.

Debugging Implementation Patterns: Break-in; Scope; Variable Value; Type Structure; Code Breakpoint.

1. The source code and the Makefile to build executables and libraries can be found in the ud1 directory:
   
   $ git clone https://bitbucket.org/softwarediagnostics/ald4

2. Launch the ud1a executable under the gdbserver:
   
   /mnt/c/ALD4/ud1$ LD_LIBRARY_PATH=. gdbserver localhost:1234 ud1a
   Process /mnt/c/ALD4/ud1/ud1a created; pid = 3652
   Listening on port 1234

3. Connect WinDbg to the remote debugger:
4. We get ready for a debugging session:
From now on, we only show the output from the command window unless we need another view.

Microsoft (R) Windows Debugger Version 10.0.27553.1004 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

64-bit machine not using 64-bit API

************* Path validation summary **************
Response Time (ms) Location
Deferred srv*
Symbol search path is: srv*
Executable search path is:
Unknown
System Version 0 UP Free x64
System Uptime: not available
Process Uptime: not available
Reloading current modules
ModLoad: 00005555`55554000 00005555`55558048 /mnt/c/ALD4/ud1/ud1
ReadVirtual() failed in GetXStateConfiguration() first read attempt (error == 0.)
00007fff`f7fd6090 mov rdi,rsp

5. Open a log file (useful when the output doesn’t fit into the buffer and we need to search for something):

0:000> .logopen C:\ALD4\ud1a.log
Opened log file 'C:\ALD4\ud1a.log'

6. The lm command lists loaded modules and their addresses (it also shows whether symbols files are loaded):

0:000> lm

start end module name
00005555`55554000 00005555`55558048 ud1a (deferred)

7. We continue process execution using the g command until we get a segmentation fault:

0:000> g
ModLoad: 00007fff`f7fd3000 00007fff`f7fd3000 linux-vdso.so.1
ModLoad: 00007fff`f7fc8000 00007fff`f7fc8000 ./libwindows.so
ModLoad: 00007fff`f7fd0000 00007fff`f7fd0000 /lib/x86_64-linux-gnu/libc.so.6
ModLoad: 00007fff`f7fd5000 00007fff`f7fd5000 /lib64/ld-linux-x86-64.so.2
(e44.e44): Signal SIGSEGV (Segmentation fault) code SEGV_MAPERR (Address not mapped to object) at 0x5555 originating from PID 70fb
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
Unable to load image ./libwindows.so, Win32 error 0n2
*** WARNING: Unable to verify timestamp for ./libwindows.so
00000000 00005555 ???

0:000> lm

start end module name
00005555`55554000 00005555`55558048 ud1a T (service symbols: DWARF Private Symbols)
C:\Users\dmitr\AppData\Local\Temp\srcD37D.tmp
00007fff`f7fd0000 00007fff`f7fd0000 libc.so T (service symbols: ELF Export Symbols)
C:\Users\dmitr\AppData\Local\Temp\srcD880.tmp
00007fff`f7fd0000 00007fff`f7fd0000 libwindows T (service symbols: DWARF Private Symbols)
C:\Users\dmitr\AppData\Local\Temp\srcD12B.tmp
00007fff`f7fd3000 00007fff`f7fd3000 linux_vdso.so (deferred)
00007fff`f7fd5000 00007fff`f7fd5000 ld_linux_x86_64.so (deferred)
There, we see that the crash happens in the `libwindows` module with the following CPU state:

<table>
<thead>
<tr>
<th># Child-SP</th>
<th>RetAddr</th>
<th>Call Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0007fff<code>ffffe318 0007fff</code>f7fc926c</td>
<td>0x5555</td>
<td>libwindows!dispatch_message+0x28 [/mnt/c/ALD4/ud1/windows.c @ 81]</td>
</tr>
<tr>
<td>01 0007fff<code>ffffe320 00005555</code>555551ef</td>
<td>libwindows!dispatch_message+0x28 [/mnt/c/ALD4/ud1/windows.c @ 81]</td>
<td></td>
</tr>
<tr>
<td>02 0007fff<code>ffffe340 0007fff</code>f7e2109b</td>
<td>ud1!main+0x88 [/mnt/c/ALD4/ud1/ud1.c @ 36]</td>
<td></td>
</tr>
<tr>
<td>03 0007fff<code>ffffe3e0 00005555</code>5555509a</td>
<td>libc_so!_libc_start_main+0x2a</td>
<td>ud1!main+0x88 [/mnt/c/ALD4/ud1/windows.c @ 81]</td>
</tr>
<tr>
<td>04 0007fff<code>ffffe4a0 ffffffff</code>ffffffff</td>
<td>ud1!start+0x2a</td>
<td></td>
</tr>
<tr>
<td>05 0007fff<code>ffffe4a8 00000000</code>00000000</td>
<td>0xffffffff`ffffffff</td>
<td></td>
</tr>
</tbody>
</table>

We switch to the `libwindows` thread stack frame #1 and set the source code location:

<table>
<thead>
<tr>
<th># Child-SP</th>
<th>RetAddr</th>
<th>Call Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0007fff<code>ffffe318 0007fff</code>f7fc926c</td>
<td>0x5555</td>
<td>libwindows!dispatch_message+0x28 [/mnt/c/ALD4/ud1/windows.c @ 81]</td>
</tr>
<tr>
<td>01 0007fff<code>ffffe320 00005555</code>555551ef</td>
<td>libwindows!dispatch_message+0x28 [/mnt/c/ALD4/ud1/windows.c @ 81]</td>
<td></td>
</tr>
<tr>
<td>02 0007fff<code>ffffe340 0007fff</code>f7e2109b</td>
<td>ud1!main+0x88 [/mnt/c/ALD4/ud1/ud1.c @ 36]</td>
<td></td>
</tr>
<tr>
<td>03 0007fff<code>ffffe3e0 00005555</code>5555509a</td>
<td>libc_so!_libc_start_main+0x2a</td>
<td>ud1!main+0x88 [/mnt/c/ALD4/ud1/windows.c @ 81]</td>
</tr>
<tr>
<td>04 0007fff<code>ffffe4a0 ffffffff</code>ffffffff</td>
<td>ud1!start+0x2a</td>
<td></td>
</tr>
<tr>
<td>05 0007fff<code>ffffe4a8 00000000</code>00000000</td>
<td>0xffffffff`ffffffff</td>
<td></td>
</tr>
</tbody>
</table>

Source search path is: SRV*;C:\ALD4\ud1

************* Path validation summary *************

<table>
<thead>
<tr>
<th>Response</th>
<th>Time (ms)</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deferred</td>
<td></td>
<td>SRV*</td>
</tr>
<tr>
<td>OK</td>
<td></td>
<td>C:\ALD4\ud1</td>
</tr>
</tbody>
</table>
Note: We see a source code window immediately to the left of the command window:

9. We see that the `window_proc` pointer is invalid, so we need to investigate when it is set in the `register_class` function below. First, we set the next frame where the `dispatch_message` was called:

```
0:000> dp libwindows!window_proc L1
00007fff`f7fcc040 00000000`00005555

0:000> kn
# Child-SP    RetAddr               Call Site
00 00007fff`ffffe318 00007fff`f7fc926c          0x5555
01 00007fff`ffffe320 00005555`555551ef     libwindows!dispatch_message+0x28
[/mnt/c/ALD4/ud1/windows.c @ 81]
02 00007fff`ffffe340 00007fff`f7e2109b   ud1almain+0x88 [/mnt/c/ALD4/ud1/ud1.c @ 36]
03 00007fff`ffffe340 00005555`5555509a    libc_so!libc_start_main+0x2e
04 00007fff`ffffe4a0 00000000`00000000 0xffffffff`ffffffff
05 00007fff`ffffe4a8 00000000`00000000 0xffffffff`ffffffff

0:000> .frame 2
02 00007fff`ffffe340 00007fff`f7e2109b   ud1almain+0x88 [/mnt/c/ALD4/ud1/ud1.c @ 36]
```
10. We can now expand local structures in the Locals window (for example, \texttt{wc}):  

![Image of GDB Locals window]

We can also dump this variable using type information:

```
0:00> dt wc
Local var @ 0x7fffffffe350 Type wnd_class_t
  +0x00 style : 3
  +0x04 window_proc : 0x00005555'55555155 void ud1a!window_proc+0
  +0x0c class_extra : 0
  +0x10 window_extra : 0
  +0x14 instance : 0
  +0x1c icon : 1
  +0x24 cursor : 2
  +0x2c background : 3
  +0x34 menu_name : 0x00005555'55556004 "menu"
  +0x3c class_name : 0x00005555'55556009 "ud1"
```

11. We need to make sure that \textit{libwindows} is loaded before we put a breakpoint on the \texttt{register_class} function. To do that, we determine the main function address to set the breakpoint there first once we restart the debugged process. Then, on break-in we set our \texttt{register_class} breakpoint since the library is already loaded.
12. Now, we finish the process (the `g` command) and see WinDbg disconnected. Then we start the `gdbserver` again and reattach WinDbg to the remote debugger.

```
/mnt/c/ALD4/ud1 $ LD_LIBRARY_PATH=. gdbserver localhost:1234 ud1a
Process /mnt/c/ALD4/ud1/ud1a created; pid = 34
Listening on port 1234
```

Microsoft (R) Windows Debugger Version 10.0.27553.1004 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

64-bit machine not using 64-bit API

************* Path validation summary *************
Response Time (ms) Location
Deferred srv*
Symbol search path is: srv*
Executable search path is:
Unknown System Version 0 UP Free x64
System Uptime: not available
Process Uptime: not available
Reloading current modules
ModLoad: 00005555`55554000 00005555`55558048   /mnt/c/ALD4/ud1a
ReadVirtual() failed in GetXStateConfiguration() first read attempt (error == 0.)
00007fff`f7fd3000 mov     rdi,rsp

13. We put a breakpoint on the main function address we determined previously and resume execution until it is hit:

```
0:000> bp 00005555`55555167
0:000> g
```

```
ModLoad: 00007fff`f7fd3000 00007fff`f7fd3000   linux-vdso.so.1
ModLoad: 00007fff`f7fc8000 00007fff`f7fccc048   ./libwindows.so
ModLoad: 00007fff`f7df0000 00007fff`f7fbc800   /lib/x86_64-linux-gnu/libc.so.6
ModLoad: 00007fff`f7f50000 00007fff`f7ffe190   /lib64/ld-linux-x86-64.so.2
Breakpoint 0 hit
Unable to load image /lib/x86_64-linux-gnu/libc.so.6, Win32 error 0n2
*** WARNING: Unable to verify timestamp for /lib/x86_64-linux-gnu/libc.so.6
udialmain:
00005555`55555167 push     rbp
```
14. We now put a breakpoint on the call to the `register_class` function (F9) and resume execution:

```
0:000> g
Breakpoint 1 hit
ud1a!main+0x6e:
00005555`555551d5 lea     rax,[rbp-
0:000> dt ud1a!wc
Local var @ 0x7fffffffefe350 Type wnd_class_t
+0x000 style            : 3
+0x004 window_proc      : 0x00005555`55555155     void ud1a!window_proc+0
+0x00c class_extra      : 0
+0x010 window_extra     : 0
+0x014 instance         : 0
+0x01c icon             : 1
+0x024 cursor           : 2
+0x02c background       : 3
+0x034 menu_name        : 0x00005555`55556004  "menu"
+0x03c class_name       : 0x00005555`55556009  "ud1"
```
15. Then, we put a breakpoint inside the `register_class` function and resume execution:

```
0:000> bp libwindows!register_class
Unable to load image ./libwindows.so, Win32 error 0x2
*** WARNING: Unable to verify timestamp for ./libwindows.so

0:000> bp libwindows!register_class
breakpoint 2 redefined

0:000> g
Breakpoint 2 hit
libwindows!register_class:
00007fff`f7fc926f push rbp
```

16. Do one step over to have the parameter initialized and inspect it:

```
0:000> p
libwindows!register_class+0x8:
00007fff`f7fc9277 mov rax,qword ptr [rbp-8] ss:00007fff`ffffe328=00007fffffffe350
```
100

0:000> dt pwc
Local var @ 0x7fffffffe328 Type p_wnd_class_t
0x00007fff`ffffe350
+0x000 style : 3
+0x008 window_proc : 0x00000000`00005555 void +5555
+0x010 class_extra : 0\n+0x014 window_extra : 0\n+0x018 instance : 0x00000001`00000000
+0x020 icon : 0x00000002`00000000
+0x028 cursor : 0x00000003`00000000
+0x030 background : 0x55555555`00000000
+0x038 menu_name : 0x55555555`00005555 "--- memory read error at address 0x55555555`00005555 ---"
+0x040 class_name : 0x00000000`00005555 "--- memory read error at address 0x00000000`00005555 ---"

0:000> dt libwindows!p_wnd_class_t
Ptr64 +0x000 style : Uint4B
+0x008 window_proc : Ptr64 void
+0x010 class_extra : Int4B
+0x014 window_extra : Int4B
+0x018 instance : Uint8B
+0x020 icon : Uint8B
+0x028 cursor : Uint8B
+0x030 background : Uint8B
+0x038 menu_name : Ptr64 Char
+0x040 class_name : Ptr64 Char

17. But if we look at the ud1a structure variant, we see its members have different offsets:

0:000> dt ud1a!wnd_class_t
+0x000 style : Uint4B
+0x004 window_proc : Ptr64 void
+0x00c class_extra : Int4B
+0x010 window_extra : Int4B
+0x014 instance : Uint8B
+0x01c icon : Uint8B
+0x024 cursor : Uint8B
+0x02c background : Uint8B
+0x034 menu_name : Ptr64 Char
+0x03c class_name : Ptr64 Char

18. These discrepancies explain the crash. Looking at the Makefile, we can see that ud1a was compiled with the -fpack-struct setting. The ud1b executable was compiled without it and runs fine. Also, the problem was coincidentally fixed without changing alignment by using a different, bigger wnd_class2_t structure in the ud1c executable that adds another 32-bit field that makes both alignments identical.

19. We continue execution (g) to have the remote process finished and then close WinDbg.
Exercise UD1 (GDB)

Goal: Learn how code generation parameters can influence process execution behavior.

Elementary Diagnostics Patterns: Crash.

Memory Analysis Patterns: Exception Stack Trace; NULL Pointer (Code); Constant Subtrace.

Debugging Implementation Patterns: Break-in; Scope; Variable Value; Type Structure; Code Breakpoint.

1. The source code and the Makefile to build executables and libraries can be found in the udI directory:

   $ git clone https://bitbucket.org/softwareanalytics/ald4

2. When we launch the ud1a executable, it crashes:

   /mnt/c/ALD4/ud1$ LD_LIBRARY_PATH=. ./ud1a
   Segmentation fault

3. We run the executable under GDB until it shows a segmentation fault:

   /mnt/c/ALD4/ud1$ LD_LIBRARY_PATH=. gdb ./ud1a

   GNU gdb (Debian 8.2.1-2+b3) 8.2.1
   Copyright (C) 2018 Free Software Foundation, Inc.
   License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
   This is free software: you are free to change and redistribute it.
   There is NO WARRANTY, to the extent permitted by law.
   Type "show copying" and "show warranty" for details.
   This GDB was configured as "x86_64-linux-gnu".
   Type "show configuration" for configuration details.
   For bug reporting instructions, please see:
   Find the GDB manual and other documentation resources online at:
   
   For help, type "help".
   Type "apropos word" to search for commands related to "word"...
   Reading symbols from ./ud1a...done.

   (gdb) r
   Starting program: /mnt/c/ALD4/ud1/ud1a
   
   Program received signal SIGSEGV, Segmentation fault.
   0x000000000000005555 in ?? ()

4. The info proc mappings and info sharedlibrary commands list loaded modules and their addresses and show if symbols are available:

   (gdb) info proc mappings
   process 14581
   Mapped address spaces:

<table>
<thead>
<tr>
<th>Start Addr</th>
<th>End Addr</th>
<th>Size</th>
<th>Offset objfile</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x555555554000</td>
<td>0x555555555000</td>
<td>0x1000</td>
<td>/mnt/c/ALD4/ud1/ud1a</td>
</tr>
<tr>
<td>0x555555555000</td>
<td>0x555555556000</td>
<td>0x1000</td>
<td>/mnt/c/ALD4/ud1/ud1a</td>
</tr>
<tr>
<td>0x555555556000</td>
<td>0x555555557000</td>
<td>0x1000</td>
<td>/mnt/c/ALD4/ud1/ud1a</td>
</tr>
<tr>
<td>0x555555557000</td>
<td>0x555555558000</td>
<td>0x1000</td>
<td>/mnt/c/ALD4/ud1/ud1a</td>
</tr>
<tr>
<td>0x555555558000</td>
<td>0x555555559000</td>
<td>0x1000</td>
<td>/mnt/c/ALD4/ud1/ud1a</td>
</tr>
<tr>
<td>0x555555559000</td>
<td>0x55555555a000</td>
<td>0x1000</td>
<td>/mnt/c/ALD4/ud1/ud1a</td>
</tr>
<tr>
<td>0x55555555a000</td>
<td>0x55555555b000</td>
<td>0x1000</td>
<td>/mnt/c/ALD4/ud1/ud1a</td>
</tr>
<tr>
<td>0x55555555b000</td>
<td>0x55555555c000</td>
<td>0x1000</td>
<td>/mnt/c/ALD4/ud1/ud1a</td>
</tr>
<tr>
<td>0x55555555c000</td>
<td>0x55555555d000</td>
<td>0x1000</td>
<td>/mnt/c/ALD4/ud1/ud1a</td>
</tr>
<tr>
<td>0x55555555d000</td>
<td>0x55555555e000</td>
<td>0x1000</td>
<td>/mnt/c/ALD4/ud1/ud1a</td>
</tr>
<tr>
<td>0x55555555e000</td>
<td>0x55555555f000</td>
<td>0x1000</td>
<td>/mnt/c/ALD4/ud1/ud1a</td>
</tr>
<tr>
<td>0x55555555f000</td>
<td>0x555555560000</td>
<td>0x1000</td>
<td>/mnt/c/ALD4/ud1/ud1a</td>
</tr>
</tbody>
</table>
5. We see that the crash happens in the libwindows module with the following CPU state:

```
(gdb) bt
#0 0x0000000000005555 in ?? ()
#1 0x0000000000005555c0 in dispatch_message (pmessage=0x7fffffffe380) at windows.c:76
#2 0x0000000000005555551ef in main (argc=1, argv=0x7fffffffe498) at ud1.c:35
```

```
(gdb) info r
rax            0x5555              21845
rbx            0x0                 0
rcx            0x7ffffff7ec3594    140737352840596
rdx            0x7fffffff4e3b0     140737488348080
rsi            0x7fffffff4e2b0     140737488347824
rdi            0x7fffffff4e3b0     140737488348080
rbp            0x7fffffff4e340     0x7fffffff4e340
rsp            0x7fffffff4e328     0x7fffffff4e328
r8             0x7fffffff4f8d00    140737353846144
r9             0x7fffffff4f8d00    140737353846144
r10            0xffffffff4f8d00   -3031
r11            0x7fffffff4f9244    140737353912900
r12            0x7fffffff4f9244    140737353912900
r13            0x7fffffff4f9244    140737353912900
r14            0x0                 0
r15            0x0                 0
rip             0x5555              0x5555
elflags        0x10266            [ PF IF RF ]
```
6. We switch to stack frame #1 and check the source code:

```
(gdb) frame 1
#1 0x00007fffff7fc926c in dispatch_message (pmessage=0x7fffffffe3b0) at windows.c:76
6  window_proc(pmessage);
```

```
(gdb) list
71 void dispatch_message(p_msg_t pmessage)
72 {
73    if (window_proc)
74      {
75         window_proc(pmessage);
76      }
77   }
78 int register_class(pwnd_class_t pwc)
```

```
(gdb) p window_proc
$1 = (void (*)(p_msg_t)) 0x555
```

```
(gdb) list 83
78   {
79     window_proc(pmessage);
80   }
81 int register_class(pwnd_class_t pwc)
82 {
83     window_proc = pwc->window_proc;
84 }
```

7. We see that the `window_proc` pointer is invalid, so we need to investigate when it is set in the `register_class` function below. First, we set the next frame where the `dispatch_message` was called:

```
(gdb) bt
#0 0x0000000000005555 in ?? ()
#1 0x00007fffff7fc926c in dispatch_message (pmessage=0x7fffffffe3b0) at windows.c:76
#2 0x0000555555555ef in main (argc=1, argv=0x7fffffffe4c8) at ud1.c:35
```

```
(gdb) frame 2
#2 0x0000555555555ef in main (argc=1, argv=0x7fffffffe4c8) at ud1.c:35
35 dispatch_message(&msg);
```

```
(gdb) list
30
31 register_class(&wc);
32 while (get_message(&msg, 0, 0, 0))
33 {
34    dispatch_message(&msg);
35  }
36 return 0;
```
8. We can now check local variables and their structures (for example, wc):

```
(gdb) info locals
msg = {hwnd = 0, message = 275, param1 = 1, param2 = 140737353912581, time = 72844112, pt = {x = 156, y = 327},
      priv = 0}
wc = {style = 3, window_proc = 0x555555555155 <window_proc>, class_extra = 0, window_extra = 0,
      instance = 0, icon = 1, cursor = 2, background = 3, menu_name = 0x555555556004 "menu", class_name = 0x555555556009 "ud1"}
```

```
(gdb) ptype /o wc
type = struct {
    /* 0      |   4 */    uint32_t style;
    /* 4      |   8 */    void (*window_proc)(p_msg_t);
    /* 12     |   4 */    int32_t class_extra;
    /* 16     |   4 */    int32_t window_extra;
    /* 20     |   8 */    uint64_t instance;
    /* 28     |   8 */    uint64_t icon;
    /* 36     |   8 */    uint64_t cursor;
    /* 44     |   8 */    uint64_t background;
    /* 52     |   8 */    char *menu_name;
    /* 60     |   8 */    char *class_name;
    /* total size (bytes): 68 */
}
```

9. We put a breakpoint on the `main` function and resume execution until it is hit:

```
(gdb) c
Continuing.
Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
```

```
(gdb) break main
Breakpoint 1 at 0x55555555517f: file ud1.c, line 20.
```

```
(gdb) r
Starting program: /mnt/c/ALD4/ud1/ud1a
Breakpoint 1, main (argc=1, argv=0x7fffffffe4c8) at ud1.c:20
20        wc.style = 3;
```

10. We now put a breakpoint on the call to the `register_class` function and resume execution to inspect the passed value of the `wc` structure:

```
(gdb) list 27, 40
27        wc.background = 3;
28        wc.menu_name = "menu";
29        wc.class_name = "ud1";
30
31              register_class(&wc);
32
33              while (get_message(&msg, 0, 0, 0))
34              {
```
11. Then, we put a breakpoint inside the `register_class` function, resume execution, and inspect the parameter:

```
(gdb) break register_class
Breakpoint 3 at 0x7fffffc9277: file windows.c, line 82.
(gdb) c
Continuing.
Breakpoint 3, register_class (pwc=0x7fffffffde360) at windows.c:82
82     window_proc = pwc->window_proc;
(gdb) p pwc
$2 = (p_wnd_class_t) 0x7fffffffde360
```

```
(gdb) ptype /o pwc
```

```
type = struct {
/* 0 | 4 */     uint32_t style;
/* 4 | 8 */     void (*window_proc)(p_msg_t);
/* 12 | 4 */    int32_t class_extra;
/* 16 | 4 */    int32_t window_extra;
/* 20 | 8 */    uint64_t instance;
/* 28 | 8 */    uint64_t icon;
/* 36 | 8 */    uint64_t cursor;
/* 44 | 8 */    uint64_t background;
/* 52 | 8 */    char *menu_name;
/* 60 | 8 */    char *class_name;
    /* total size (bytes): 68 */
}
```
12. But if we compare the structure inside the function with the structure variant outside (see step #10), we see its members have different offsets:

```c
struct {
    /* 0 */ 4 */    uint32_t style;
    /* 4 */ 8 */    void (*window_proc)(p_msg_t);
    /* 12 */ 4 */    int32_t class_extra;
    /* 16 */ 4 */    int32_t window_extra;
    /* 20 */ 8 */    uint64_t instance;
    /* 28 */ 8 */    uint64_t icon;
    /* 36 */ 8 */    uint64_t cursor;
    /* 44 */ 8 */    uint64_t background;
    /* 52 */ 8 */    char *menu_name;
    /* 60 */ 8 */    char *class_name;

    /* total size (bytes): 68 */
}*
```

13. These discrepancies explain the crash. Looking at the Makefile, we can see that *ud1a* was compiled with the `-fpack-struct` setting. The *ud1b* executable was compiled without it and runs fine. Also, the problem was coincidentally fixed without changing alignment by using a different, bigger *wnd_class2_t* structure in the *ud1c* executable that adds another 32-bit field that makes both alignments identical.

14. We continue the execution and then quit GDB.

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x0000000000000555 in ?? ()
```

```
(gdb) c
Continuing.

Program terminated with signal SIGSEGV, Segmentation fault. 
The program no longer exists.
```

(gdb) q
Exercise UD1 (LLDB)

**Goal:** Learn how code generation parameters can influence process execution behavior.

**Elementary Diagnostics Patterns:** Crash.

**Memory Analysis Patterns:** Exception Stack Trace; NULL Pointer (Code); Constant Subtrace.

**Debugging Implementation Patterns:** Break-in; Scope; Variable Value; Type Structure; Code Breakpoint.

1. The source code and the `Makefile` to build executables and libraries can be found in the `ud1` directory:

   ```
   $ git clone https://bitbucket.org/softwarediagnostics/ald4
   ```

2. When we launch the `ud1a` executable, it crashes:

   ```
   /mnt/c/ALD4/ud1$ LD_LIBRARY_PATH=. ./ud1a
   Segmentation fault
   ```

3. We run the executable under LLDB until it shows a segmentation fault:

   ```
   /mnt/c/ALD4/ud1$ LD_LIBRARY_PATH=. lldb ./ud1a
   (lldb) target create './ud1a'
   Current executable set to './ud1a' (x86_64).
   ```

   ```
   (lldb) r
   Process 112 launched: '/mnt/c/ALD4/ud1/ud1a' (x86_64)
   Process 112 stopped
   * thread #1, name = 'ud1a', stop reason = signal SIGSEGV: invalid address (fault address: 0x5555)
     frame #0: 0x0000000000005555
     error: memory read failed for 0x5400
   ```

4. The `image list` command lists loaded modules and their addresses:

   ```
   (lldb) image list
   [ 0] 2C8F3A74-A0FC-977F-6362-C129A1E426DA-25DA9B99 /mnt/c/ALD4/ud1/ud1a
   /usr/lib/debug/.build-id/83/7433dd4258a7138a28c4f2032d17ad92a15db5.debug
   [ 3] 6F3490CE-A127-8C7-4A4F-D33AC3B6CAA-A2CE115B 0x00007fffffffe3b0 [vds] (0x00007fffffffe3b0)
   /usr/lib/debug/.build-id/c7/aa9a1e121fe2395f3840f3f0213146046d9fe3.debug
   ```

5. We see that the crash happens in the `libwindows` module with the following CPU state:

   ```
   (lldb) bt
   * thread #1, name = 'ud1a', stop reason = signal SIGSEGV: invalid address (fault address: 0x5555)
     * frame #0: 0x0000000000005555
       libwindows.so`dispatch_message(pmessa...)
     frame #1: 0x00007fffffffe3b0 at windows.c:76
     frame #2: 0x000055555555ef ud1a main(argc=1, argv=0x00007fffffffe4c8) at ud1.c:35
     frame #3: 0x00007fffffffe3b0 at libc.so.6`__libc_start_main(main=(ud1a`main at ud1.c:16),...)
     stack_end=0x00007fffffffe4b8 at libc-start.c:308
     frame #4: 0x0000555555555559a ud1a`_start + 42
   ```
General Purpose Registers:

rax = 0x0000000000000555
rbx = 0x0000000000000000
rcx = 0x00007fffffffe3594  libc.so.6`__GI___nanosleep + 20 at nanosleep.c:28
rdx = 0x00007fffffffe3b0
rdi = 0x00007fffffffe3b0
rsi = 0x00007fffffffe2b0
rbp = 0x00007fffffffe340
rsp = 0x00007fffffffe328
r8 = 0x00007fffffff7ec3594  libc.so.6`__GI___nanosleep + 20 at nanosleep.c:28
r9 = 0x00007fffffffe3b0
rdx = 0x00007fffffffe3b0
rdi = 0x00007fffffffe3b0
rsi = 0x00007fffffffe2b0
rbp = 0x00007fffffffe340
rsp = 0x00007fffffffe328
r8 = 0x00007fffffff7ec3594  libc.so.6`__GI___nanosleep + 20 at nanosleep.c:28
r9 = 0x00007fffffffe3b0
rdx = 0x00007fffffffe3b0
rdi = 0x00007fffffffe3b0
rsi = 0x00007fffffffe2b0
rbp = 0x00007fffffffe340
rsp = 0x00007fffffffe328

6. We switch to stack frame #1 and check the source code:

```
(lldb) frame select 1
frame #1: 0x00007fffffffe3b0

6
7

7
8
9

(lldb) p window_proc
(void (*)(p_msg_t)) $0 = 0x0000000000000555

(lldb) list 80
80  int register_class(p_wnd_class_t pwc)
81  {
82      window_proc = pwc->window_proc;
83  }
84
85  int register_class2(p_wnd_class2_t pwc)
86  {
87      window_proc = pwc->window_proc;
88  }
```

7. We see that the `window_proc` pointer is invalid, so we need to investigate when it is set in the `register_class` function below. First, we set the next frame where the `dispatch_message` was called:
109

(lldb) bt
* thread #1, name = 'ud1a', stop reason = signal SIGSEGV: invalid address (fault address: 0x5555)
  frame #0: 0x0000000000000000 (lldb)
  frame #1: 0x00007ffff7fc926c `dispatch_message` (pmessage=0x0000000000000000) at windows.c:76
  frame #2: 0x0000000000000000 `ud1a`main(argc=1, argv=0x0000000000000000) at ud1.c:35
  frame #3: 0x00007ffff7fc926c `dispatch_message` (pmessage=0x0000000000000000) at windows.c:76
  frame #4: 0x0000000000000000 (lldb)

(lldb) frame select 2
frame #2: 0x0000000000000000 `ud1a`main(argc=1, argv=0x0000000000000000) at ud1.c:35
  32    while (get_message(&msg, 0, 0, 0))
  33    {
  34        dispatch_message(&msg);
  35    }
  36    return 0;

(lldb) list 30
  30    register_class(&wc);
  31    while (get_message(&msg, 0, 0, 0))
  32    {
  33        dispatch_message(&msg);
  34    }
  35    return 0;
  36

8. We can now check local variables and their structures (for example, wc):

(lldb) frame variable
(int) argc = 1
(char **) argv = 0x0000000000000000
(msg_t) msg = {
    hwnd = 0
    message = 0
    param1 = 0
    param2 = 0
    time = 0
    pt = (x = 0, y = 0)
    priv = 0
}
(wnd_class_t) wc = {
    style = 3
    window_proc = 0x0000000000000000 `ud1a`window_proc at ud1.c:11
    class_extra = 0
    window_extra = 0
    instance = 0
    icon = 0
    cursor = 0
    background = 0
    menu_name = 0x0000000000000000 "menu"
    class_name = 0x0000000000000000 "ud1"
}
(lldb) p &wc
(wnd_class_t *) $1 = 0x0000000000000000

109
9. We put a breakpoint on the `main` function and resume execution until it is hit:

```
9. We put a breakpoint on the main function and resume execution until it is hit:

(lldb) c
Process 112 resuming
Process 112 exited with status = 11 (0x0000000b)
```

```
(lldb) breakpoint set -name main
Breakpoint 1: where = ud1a`main + 24 at ud1.c:2, address = 0x00005555555517f
```

```
(lldb) c
Process 112 resuming
Process 112 exited with status = 11 (0x0000000b)
```

```
(lldb) r
Process 176 launched: '/mnt/c/ALD4/ud1/ud1a' (x86_64)
Process 176 stopped
* thread #1, name = 'ud1a', stop reason = breakpoint 1.1
  frame #0: 0x00005555555517f ud1a`main(argc=1, argv=0x00007fffffffe4c8) at ud1.c:20
  20       msg_t msg;
  21       wnd_class_t wc;
  22
-> 23       wc.style = 3;
  24       wc.window_proc = window_proc;
  25       wc.class_extra = 0;
  26       wc.window_extra = 0;
```

10. We now put a breakpoint on the call to the `register_class` function and resume execution to inspect the passed value of the `wc` structure:

```
10. We now put a breakpoint on the call to the register_class function and resume execution to inspect the passed value of the wc structure:

(lldb) list 27
    wc.background = 3;
    wc.menu_name = "menu";
    wc.class_name = "ud1";
  31       register_class(&wc);
  32
  33       while (get_message(&msg, 0, 0, 0))
  34           {
  35               dispatch_message(&msg);
  36           }
```

```
(lldb) breakpoint set -line 31
Breakpoint 2: where = ud1a`main + 110 at ud1.c:31, address = 0x0000555555551d5
```

```
(lldb) c
Process 176 resuming
Process 176 stopped
* thread #1, name = 'ud1a', stop reason = breakpoint 2.1
  frame #0: 0x0000555555551d5 ud1a`main(argc=1, argv=0x00007fffffffe4c8) at ud1.c:31
  28       wc.menu_name = "menu";
  29       wc.class_name = "ud1";
  30
-> 31       register_class(&wc);
  32
  33       while (get_message(&msg, 0, 0, 0))
  34           {
```
11. Then, we put a breakpoint inside the `register_class` function, resume execution, and inspect the parameter:

```
(lldb) breakpoint set -name register_class
Breakpoint 3: where = libwindows.so`register_class + 8 at windows.c:82, address = 0x00007fffff7fc9277
```

```
(lldb) c
Process 176 resuming
Process 176 stopped
* thread #1, name = 'ud1a', stop reason = breakpoint 3.1
  frame #0: 0x00007fffff7fc9277 libwindows.so`register_class(pwc=0x00007fffffffe360) at windows.c:82
    79
    80   int register_class(p_wnd_class_t pwc)
    81   {
-> 82       window_proc = pwc->window_proc;
    83   }
    84
    85   int register_class2(p_wnd_class2_t pwc)
```

```
(lldb) p pwc
(p_wnd_class_t) $5 = 0x00007fffffffe360
```

```
(lldb) memory read pwc
0x7fffffffe360: 03 00 00 00 55 51 55 55 55 55 00 00 00 00 00 00 ....UQUUUU......
0x7fffffffe370: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 .................
```

```
(lldb) p &pwc->window_proc
(void (**)(p_msg_t)) $10 = 0x00007fffffffe368
```

```
(lldb) p (*pwc)
((anonymous struct)) $8 = {
    style = 3
    window_proc = 0x0000000000005555
    class_extra = 0
    window_extra = 0
    instance = 4294967296
    icon = 8589934592
```
12. So, if we compare the structure field address inside the function with the structure variant outside (see step #10), we see its members have different offsets:

\[
\text{(lldb) } p \&wc\text{.window_proc} \\
\text{(void (**)(p_msg_t)) $10 = 0x00007fffffff364}
\]

13. These discrepancies explain the crash. Looking at the Makefile, we can see that \textit{ud1a} was compiled with the \texttt{-fpack-struct} setting. The \textit{ud1b} executable was compiled without it and runs fine. Also, the problem was coincidentally fixed without changing alignment by using a different, bigger \textit{wnd\_class2\_t} structure in the \textit{ud1c} executable that adds another 32-bit field that makes both alignments identical.

14. We continue the execution and then quit LLDB.

\[
\text{(lldb) c} \\
\text{Process 176 resuming} \\
\text{Process 176 stopped} \\
* \text{thread #1, name = 'ud1a', stop reason = signal SIGSEGV: invalid address (fault address: 0x5555)} \\
\text{ frame #0: 0x00000000005555} \\
\text{ error: memory read failed for 0x5400}
\]

\[
\text{(lldb) q} \\
\text{Process 176 resuming} \\
\text{Process 176 exited with status = 11 (0x0000000b)}
\]