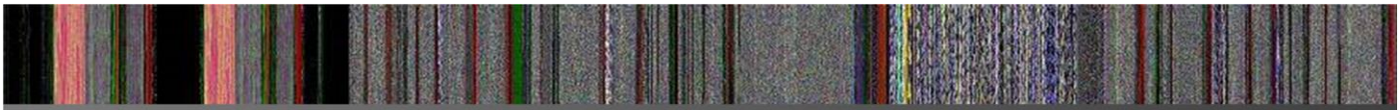

Linux Reversing Disassembly Reconstruction Accelerated

Second 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: 978-1-912636-74-7 (Paperback)

Revision 2.00 (March 2023)

Contents

Exercise R1 (x64, GDB)

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. Load a core dump *notepad.61* and *notepad* executable from x64/MemoryDumps directory:

```
~/ADDR-Linux/x64/MemoryDumps$ gdb -c notepad.61 -se notepad
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:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from notepad...(no debugging symbols found)...done.
[New LWP 61]
Core was generated by `./notepad'.
#0  0x000000000043c4e1 in nanosleep ()
```

2. We open a log file:

```
(gdb) set logging on R1.log
Copying output to R1.log.
```

3. We get this stack trace:

```
(gdb) bt
#0  0x000000000043c4e1 in nanosleep ()
#1  0x000000000043c49a in sleep ()
#2  0x0000000000401c17 in internal_get_message ()
#3  0x0000000000401c57 in get_message ()
#4  0x0000000000401cea in main ()
```

4. Let's check the main CPU registers:

```
(gdb) info registers
rax            0xfffffffffffffdfc  -516
rbx            0xffffffffffffffc0  -64
rcx            0x43c4e1            4441313
rdx            0x401b4d            4201293
rsi            0x7ffffe5129d0      140737460120016
rdi            0x7ffffe5129d0      140737460120016
rbp            0x0                 0x0
rsp            0x7ffffe5129c8      0x7ffffe5129c8
```

```
r8             0x2                2
r9             0x2                2
r10            0x7                7
r11            0x246              582
r12            0x4029c0           4204992
r13            0x0                0
r14            0x4a5018           4870168
r15            0x0                0
rip            0x43c4e1           0x43c4e1 <nanosleep+17>
eflags         0x246              [ PF ZF IF ]
cs             0x33               51
ss             0x2b               43
ds             0x0                0
es             0x0                0
fs             0x0                0
gs             0x0                0
```

Note: The register parts and naming are illustrated in the MCD-R1-x64.xlsx A section.

5. The current instruction registers (registers that are used and affected by the current instruction or semantically tied to it) can be checked by listing them individually:

```
(gdb) x/i $rip
=> 0x43c4e1 <nanosleep+17>:     cmp    $0xfffffffffffff000,%rax

(gdb) info registers rax
rax            0xfffffffffffffdfc  -516
```

Note: Most commands require $ to be prefixed to a register name; otherwise, they consider it a symbol name.

```
(gdb) x/i rip
No symbol table is loaded.  Use the "file" command.
```

6. Any register value or its named parts can be checked with the **info registers** command (**i r**):

```
(gdb) i r r12
r12            0x4029c0           4204992

(gdb) i r r12d
r12d            0x4029c0           4204992

(gdb) i r r12w
r12w            0x29c0             10688

(gdb) i r r12l
r12l            0xc0               -64
```

The original x86 registry set can be accessed using mnemonics:

```
(gdb) i r rbx
rbx            0xffffffffffffffc0  -64

(gdb) i r ebx
ebx            0xffffffc0          -64

(gdb) i r bx
bx             0xffc0              -64
```

```
(gdb) i r bh
bh             0xff               -1

(gdb) i r bl
bl             0xc0               -64

(gdb) i r rbp
rbp            0x0                0x0

(gdb) i r ebp
ebp            0x0                0

(gdb) i r rsi
rsi            0x7ffffe5129d0     140737460120016

(gdb) i r esi
esi            0xfe5129d0         -28235312

(gdb) i r si
si             0x29d0             10704

(gdb) i r sil
sil            0xd0               -48
```

7. Individual parts can also be interpreted using the **print** command (**p**):

```
(gdb) p/z $r11
$1 = 0x0000000000000246

(gdb) p/z (int[2])$r11
$2 = {0x00000246, 0x00000000}

(gdb) p/z (short[4])$r11
$3 = {0x0246, 0x0000, 0x0000, 0x0000}

(gdb) p/z (char[8])$r11
$4 = {0x46, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}

(gdb) p/d (char[8])$r11
$5 = {70, 2, 0, 0, 0, 0, 0, 0}
```

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

```
(gdb) x $rax
0xfffffffffffffdfc:     Cannot access memory at address 0xfffffffffffffdfc
```

Note: The following output for RDI is illustrated in the MCD-R1-x64.xlsx B section.

```
(gdb) x/gx $rdi
0x7ffffe5129d0: 0x00000000ffffffff0
```

9. We can also specify value count or limit to just one value and use finer granularity for memory dumping:

```
(gdb) x/20gx $rdi
0x7ffffe5129d0: 0x00000000fffffff0      0x00000000130f2d88
0x7ffffe5129e0: 0x0000000000000140      0x3ebeea4a71786c00
0x7ffffe5129f0: 0x000000000000000a      0x0000000000400470
0x7ffffe512a00: 0x00007ffffe512a20      0x0000000000401c17
0x7ffffe512a10: 0x0000000000000000      0x00007ffffe512a80
0x7ffffe512a20: 0x00007ffffe512a60      0x0000000000401c57
0x7ffffe512a30: 0x0000000000000000      0x0000000000000000
0x7ffffe512a40: 0x0000000000000000      0x00007ffffe512a80
0x7ffffe512a50: 0x0000000000000001      0xffffffffffe512bd8
0x7ffffe512a60: 0x00007ffffe512ab0      0x0000000000401cea

(gdb) x/20wx $rdi
0x7ffffe5129d0: 0xfffffff0      0x00000000      0x130f2d88      0x00000000
0x7ffffe5129e0: 0x00000140      0x00000000      0x71786c00      0x3ebeea4a
0x7ffffe5129f0: 0x0000000a      0x00000000      0x00400470      0x00000000
0x7ffffe512a00: 0xfe512a20      0x00007fff      0x00401c17      0x00000000
0x7ffffe512a10: 0x00000000      0x00000000      0xfe512a80      0x00007fff
```

Note: The similar output for RDI as below is illustrated in the MCD-R1-x64.xlsx C section.

```
(gdb) x/20hx $rdi
0x7ffffe5129d0: 0xfff0  0xffff  0x0000  0x0000  0x2d88  0x130f  0x0000  0x0000
0x7ffffe5129e0: 0x0140  0x0000  0x0000  0x0000  0x6c00  0x7178  0xea4a  0x3ebe
0x7ffffe5129f0: 0x000a  0x0000  0x0000  0x0000

(gdb) x/20bx $rdi
0x7ffffe5129d0: 0xf0    0xff    0xff    0xff    0x00    0x00    0x00    0x00
0x7ffffe5129d8: 0x88    0x2d    0x0f    0x13    0x00    0x00    0x00    0x00
0x7ffffe5129e0: 0x40    0x01    0x00    0x00
```

Note: GDB remembers formatting:

```
(gdb) x/20 $rsp
0x7ffffe5129c8: 0x9a    0xc4    0x43    0x00    0x00    0x00    0x00    0x00
0x7ffffe5129d0: 0xf0    0xff    0xff    0xff    0x00    0x00    0x00    0x00
0x7ffffe5129d8: 0x88    0x2d    0x0f    0x13
```

10. 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):

```
(gdb) p/z $rsp
$6 = 0x00007ffffe5129c8

(gdb) p/z (int[2])$rsp
$7 = {0xfe5129c8, 0x00007fff}

(gdb) p/z $rsp
$8 = 0x00007ffffe5129c8

(gdb) p/z (short[4])$rsp
$9 = {0x29c8, 0xfe51, 0x7fff, 0x0000}

(gdb) p/z $rsp
$10 = 0x00007ffffe5129c8
```

```
(gdb) p/z (char[8])$rsp
$11 = {0xc8, 0x29, 0x51, 0xfe, 0xff, 0x7f, 0x00, 0x00}
```

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

```
(gdb) x/40a $rsp
0x7ffffe5129c8: 0x43c49a <sleep+58>     0xffffffff0
0x7ffffe5129d8: 0x130f2d88      0x140
0x7ffffe5129e8: 0x3ebeea4a71786c00      0xa
0x7ffffe5129f8: 0x400470        0x7ffffe512a20
0x7ffffe512a08: 0x401c17 <internal_get_message+128>     0x0
0x7ffffe512a18: 0x7ffffe512a80  0x7ffffe512a60
0x7ffffe512a28: 0x401c57 <get_message+57>       0x0
0x7ffffe512a38: 0x0     0x0
0x7ffffe512a48: 0x7ffffe512a80  0x1
0x7ffffe512a58: 0xfffffffffe512bd8      0x7ffffe512ab0
0x7ffffe512a68: 0x401cea <main+56>      0x7ffffe512bd8
0x7ffffe512a78: 0x1004029a4     0x0
0x7ffffe512a88: 0x113   0x1
0x7ffffe512a98: 0x401b4d <time_proc>    0x9c04578350
0x7ffffe512aa8: 0x147   0x402930 <__libc_csu_init>
0x7ffffe512ab8: 0x402331 <__libc_start_main+977>        0x0
0x7ffffe512ac8: 0x100000000     0x7ffffe512bd8
0x7ffffe512ad8: 0x401cb2 <main> 0x0
0x7ffffe512ae8: 0x8e00000006    0xa00000080
0x7ffffe512af8: 0x0     0x0
```

```
(gdb) info symbol 0x401b4d
time_proc in section .text of /home/coredump/ADDR-Linux/x64/MemoryDumps/notepad
```

Note: The address `0x7ffffe512a98` that points to `0x401b4d` doesn't have an associated symbol:

```
(gdb) info symbol 0x7ffffe512a98
No symbol matches 0x7ffffe512a98.
```

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

```
(gdb) p/z $rip
$12 = 0x000000000043c4e1

(gdb) info symbol 0x000000000043c4e1
nanosleep + 17 in section .text of /home/coredump/ADDR-Linux/x64/MemoryDumps/notepad

(gdb) info symbol $rip
nanosleep + 17 in section .text of /home/coredump/ADDR-Linux/x64/MemoryDumps/notepad

(gdb) i r $rip
rip            0x43c4e1            0x43c4e1 <nanosleep+17>

(gdb) x/i $rip
=> 0x43c4e1 <nanosleep+17>:     cmp    $0xfffffffffffff000,%rax
```

12. Now we come to the next pointer level after its value and symbol: its interpretation. We call it an **Interpreted Pointer**, **S[3]**. Such interpretation is implemented either via typed structures (the **print** command) or via various debugger commands or scripts that format information for us. In our example, we would like to check the memory pointed to by the address 0x7ffffe512a80. We suspect it might be **msg_t** structure related to the "get message" loop:

```
typedef struct
{
  hwnd_t    hwnd;
  uint64_t  message;
  uint64_t  param1;
  uint64_t  param2;
  uint32_t  time;
  POINT     pt;
  uint32_t  private;
} msg_t, *p_msg_t;
```

```
(gdb) x/40a $rsp
0x7ffffe5129c8: 0x43c49a <sleep+58>     0xfffffff0
0x7ffffe5129d8: 0x130f2d88      0x140
0x7ffffe5129e8: 0x3ebeea4a71786c00      0xa
0x7ffffe5129f8: 0x400470        0x7ffffe512a20
0x7ffffe512a08: 0x401c17 <internal_get_message+128>     0x0
0x7ffffe512a18: 0x7ffffe512a80  0x7ffffe512a60
0x7ffffe512a28: 0x401c57 <get_message+57>       0x0
0x7ffffe512a38: 0x0     0x0
0x7ffffe512a48: 0x7ffffe512a80  0x1
0x7ffffe512a58: 0xffffffffe512bd8       0x7ffffe512ab0
0x7ffffe512a68: 0x401cea <main+56>      0x7ffffe512bd8
0x7ffffe512a78: 0x1004029a4     0x0
0x7ffffe512a88: 0x113   0x1
0x7ffffe512a98: 0x401b4d <time_proc>    0x9c04578350
0x7ffffe512aa8: 0x147   0x402930 <__libc_csu_init>
0x7ffffe512ab8: 0x402331 <__libc_start_main+977>        0x0
0x7ffffe512ac8: 0x100000000     0x7ffffe512bd8
0x7ffffe512ad8: 0x401cb2 <main> 0x0
0x7ffffe512ae8: 0x8e00000006    0xa00000080
0x7ffffe512af8: 0x0     0x0
```

```
(gdb) x/20gx 0x7ffffe512a80
0x7ffffe512a80: 0x0000000000000000      0x0000000000000113
0x7ffffe512a90: 0x0000000000000001      0x0000000000401b4d
0x7ffffe512aa0: 0x0000009c04578350      0x0000000000000147
0x7ffffe512ab0: 0x0000000000402930      0x0000000000402331
0x7ffffe512ac0: 0x0000000000000000      0x0000000100000000
0x7ffffe512ad0: 0x00007ffffe512bd8      0x0000000000401cb2
0x7ffffe512ae0: 0x0000000000000000      0x0000008e00000006
0x7ffffe512af0: 0x0000000a00000080      0x0000000000000000
0x7ffffe512b00: 0x0000000000000000      0x0000000000000000
0x7ffffe512b10: 0x0000000000000000      0x0000000000000000
```

```
(gdb) x/20wx 0x7ffffe512a80
0x7ffffe512a80: 0x00000000      0x00000000      0x00000113      0x00000000
0x7ffffe512a90: 0x00000001      0x00000000      0x00401b4d      0x00000000
0x7ffffe512aa0: 0x04578350      0x0000009c      0x00000147      0x00000000
0x7ffffe512ab0: 0x00402930      0x00000000      0x00402331      0x00000000
0x7ffffe512ac0: 0x00000000      0x00000000      0x00000000      0x00000001
```

```
(gdb) x/20wd 0x7ffffe512a80
0x7ffffe512a80: 0       0       275     0
0x7ffffe512a90: 1       0       4201293 0
0x7ffffe512aa0: 72844112        156     327     0
0x7ffffe512ab0: 4204848 0       4203313 0
0x7ffffe512ac0: 0       0       0       1

(gdb) p *(msg_t *)0x7ffffe512a80
No symbol table is loaded.  Use the "file" command.

(gdb) info types msg_t
All types matching regular expression "msg_t":
```

Note: Suppose that the raw structure makes sense for a timer message (**0x113**) where param1 is a timer ID (**1**), and usually a callback function (param2) is NULL (0x0), but in our case, it is not (**0x0000000000401b4d**, as we saw previously, *time_proc*). Also, mouse pointer data makes sense (**0x9c, 0x147**). Unfortunately, the **msg_t** structure type is not available in the *notepad* memory dump and executable, and suppose we don't have a debug symbol file for it either. However, we can add a different unrelated symbol file, for example, *extra-symbols* from x64/MemoryDumps/ExtraSymbols, which was compiled with full debug symbols and should have structures necessary for this message loop processing (**Injected Symbols** memory analysis pattern).

13. We specify an additional symbol file path:

```
(gdb) add-symbol-file ./ExtraSymbols/extra-symbols
add symbol table from file "./ExtraSymbols/extra-symbols"
(y or n) y
Reading symbols from ./ExtraSymbols/extra-symbols...done.
```

14. Now we can use the **msg_t** structure:

```
(gdb) info types msg_t
All types matching regular expression "msg_t":

File extra-symbols.c:
26:     typedef struct {
    hwnd_t hwnd;
    uint64_t message;
    uint64_t param1;
    uint64_t param2;
    uint32_t time;
    POINT pt;
    uint32_t private;
} msg_t;

(gdb) p *(msg_t *)0x7ffffe512a80
$13 = {hwnd = 0, message = 275, param1 = 1, param2 = 4201293, time = 72844112, pt = {x = 156, y
= 327}, private = 0}

(gdb) set print pretty on
```

```
(gdb) p *(msg_t *)0x7ffffe512a80
$14 = {
  hwnd = 0,
  message = 275,
  param1 = 1,
  param2 = 4201293,
  time = 72844112,
  pt = {
    x = 156,
    y = 327
  },
  private = 0
}

(gdb) p/x *(msg_t *)0x7ffffe512a80
$15 = {
  hwnd = 0x0,
  message = 0x113,
  param1 = 0x1,
  param2 = 0x401b4d,
  time = 0x4578350,
  pt = {
    x = 0x9c,
    y = 0x147
  },
  private = 0x0
}

(gdb) p/z *(msg_t *)0x7ffffe512a80
$16 = {
  hwnd = 0x0000000000000000,
  message = 0x0000000000000113,
  param1 = 0x0000000000000001,
  param2 = 0x0000000000401b4d,
  time = 0x04578350,
  pt = {
    x = 0x0000009c,
    y = 0x00000147
  },
  private = 0x00000000
}
```

Note: Don't forget to unload the loaded symbol file if it interferes with the original section symbols:

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000401b4d <+0>:     push   %rbp
   0x0000000000401b4e <+1>:     mov    %rsp,%rbp
   0x0000000000401b51 <+4>:     mov    $0x1,%eax
   0x0000000000401b56 <+9>:     pop    %rbp
   0x0000000000401b57 <+10>:    retq
   0x0000000000401b58 <+0>:     push   %rbp
   0x0000000000401b59 <+1>:     mov    %rsp,%rbp
   0x0000000000401b5c <+4>:     mov    %edi,-0x4(%rbp)
End of assembler dump.

(gdb) info symbol main
time_proc in section .text of /home/coredump/ADDR-Linux/x64/MemoryDumps/notepad
main in section .text of /home/coredump/ADDR-Linux/x64/MemoryDumps/ExtraSymbols/extra-symbols
```

```
(gdb) remove-symbol-file /home/coredump/ADDR-Linux/x64/MemoryDumps/ExtraSymbols/extra-symbols
Remove symbol table from file "/home/coredump/ADDR-Linux/x64/MemoryDumps/ExtraSymbols/extra-
symbols"? (y or n) y

(gdb) info symbol main
main in section .text of /home/coredump/ADDR-Linux/x64/MemoryDumps/notepad

(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000401cb2 <+0>:     push   %rbp
   0x0000000000401cb3 <+1>:     mov    %rsp,%rbp
   0x0000000000401cb6 <+4>:     sub    $0x40,%rsp
   0x0000000000401cba <+8>:     mov    %edi,-0x34(%rbp)
   0x0000000000401cbd <+11>:    mov    %rsi,-0x40(%rbp)
   0x0000000000401cc1 <+15>:    jmp    0x401ccf <main+29>
   0x0000000000401cc3 <+17>:    lea    -0x30(%rbp),%rax
   0x0000000000401cc7 <+21>:    mov    %rax,%rdi
   0x0000000000401cca <+24>:    callq  0x401c87 <dispatch_message>
   0x0000000000401ccf <+29>:    lea    -0x30(%rbp),%rax
   0x0000000000401cd3 <+33>:    mov    $0x0,%ecx
   0x0000000000401cd8 <+38>:    mov    $0x0,%edx
   0x0000000000401cdd <+43>:    mov    $0x0,%esi
   0x0000000000401ce2 <+48>:    mov    %rax,%rdi
   0x0000000000401ce5 <+51>:    callq  0x401c1e <get_message>
   0x0000000000401cea <+56>:    test   %eax,%eax
   0x0000000000401cec <+58>:    jne    0x401cc3 <main+17>
   0x0000000000401cee <+60>:    mov    $0x0,%eax
   0x0000000000401cf3 <+65>:    leaveq
   0x0000000000401cf4 <+66>:    retq
End of assembler dump.
```

15. When we have an exception such as a breakpoint or access violation, the values of the thread CPU registers are saved, 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:

```
(gdb) bt
#0  0x000000000043c4e1 in nanosleep ()
#1  0x000000000043c49a in sleep ()
#2  0x0000000000401c17 in internal_get_message ()
#3  0x0000000000401c57 in get_message ()
#4  0x0000000000401cea in main ()

(gdb) i r
rax            0xfffffffffffffdfc  -516
rbx            0xffffffffffffffc0  -64
rcx            0x43c4e1            4441313
rdx            0x401b4d            4201293
rsi            0x7ffffe5129d0      140737460120016
rdi            0x7ffffe5129d0      140737460120016
rbp            0x0                 0x0
rsp            0x7ffffe5129c8      0x7ffffe5129c8
r8             0x2                 2
r9             0x2                 2
r10            0x7                 7
r11            0x246               582
r12            0x4029c0            4204992
r13            0x0                 0
r14            0x4a5018            4870168
```

```
r15            0x0                0
rip            0x43c4e1           0x43c4e1 <nanosleep+17>
eflags         0x246              [ PF ZF IF ]
cs             0x33               51
ss             0x2b               43
ds             0x0                0
es             0x0                0
fs             0x0                0
gs             0x0                0
```

16. In any situation when we move down to the next frame, for example, to *sleep* (0x000000000043c49a points to the next instruction after *nanosleep* was called) and to *main*, we don't have most CPU registers' values saved previously (**i r** command gives accurate values only for the topmost frame 0 except RIP and RSP and perhaps a few other registers such as RBP):

```
(gdb) bt
#0  0x000000000043c4e1 in nanosleep ()
#1  0x000000000043c49a in sleep ()
#2  0x0000000000401c17 in internal_get_message ()
#3  0x0000000000401c57 in get_message ()
#4  0x0000000000401cea in main ()

(gdb) disassemble 0x000000000043c49a
Dump of assembler code for function sleep:
   0x000000000043c460 <+0>:     push   %rbp
   0x000000000043c461 <+1>:     push   %rbx
   0x000000000043c462 <+2>:     sub    $0x28,%rsp
   0x000000000043c466 <+6>:     mov    $0xfffffffffffffc0,%rbx
   0x000000000043c46d <+13>:    mov    %fs:0x28,%rax
   0x000000000043c476 <+22>:    mov    %rax,0x18(%rsp)
   0x000000000043c47b <+27>:    xor    %eax,%eax
   0x000000000043c47d <+29>:    mov    %edi,%eax
   0x000000000043c47f <+31>:    mov    %rsp,%rdi
   0x000000000043c482 <+34>:    mov    %rdi,%rsi
   0x000000000043c485 <+37>:    mov    %fs:(%rbx),%ebp
   0x000000000043c488 <+40>:    mov    %rax,(%rsp)
   0x000000000043c48c <+44>:    movq   $0x0,0x8(%rsp)
   0x000000000043c495 <+53>:    callq  0x43c4d0 <nanosleep>
   0x000000000043c49a <+58>:    test   %eax,%eax
   0x000000000043c49c <+60>:    js     0x43c4c0 <sleep+96>
   0x000000000043c49e <+62>:    mov    %ebp,%fs:(%rbx)
   0x000000000043c4a1 <+65>:    xor    %eax,%eax
   0x000000000043c4a3 <+67>:    mov    0x18(%rsp),%rdx
   0x000000000043c4a8 <+72>:    xor    %fs:0x28,%rdx
   0x000000000043c4b1 <+81>:    jne    0x43c4c5 <sleep+101>
   0x000000000043c4b3 <+83>:    add    $0x28,%rsp
   0x000000000043c4b7 <+87>:    pop    %rbx
   0x000000000043c4b8 <+88>:    pop    %rbp
   0x000000000043c4b9 <+89>:    retq
   0x000000000043c4ba <+90>:    nopw   0x0(%rax,%rax,1)
   0x000000000043c4c0 <+96>:    mov    (%rsp),%eax
   0x000000000043c4c3 <+99>:    jmp    0x43c4a3 <sleep+67>
   0x000000000043c4c5 <+101>:   callq  0x43edd0 <__stack_chk_fail_local>
End of assembler dump.
```

```
(gdb) i r
rax            0xffffffffffffffdc  -516
rbx            0xffffffffffffffc0  -64
rcx            0x43c4e1            4441313
rdx            0x401b4d            4201293
rsi            0x7ffffe5129d0      140737460120016
rdi            0x7ffffe5129d0      140737460120016
rbp            0x0                 0x0
rsp            0x7ffffe5129c8      0x7ffffe5129c8
r8             0x2                 2
r9             0x2                 2
r10            0x7                 7
r11            0x246               582
r12            0x4029c0            4204992
r13            0x0                 0
r14            0x4a5018            4870168
r15            0x0                 0
rip            0x43c4e1            0x43c4e1 <nanosleep+17>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0

(gdb) frame 1
#1  0x000000000043c49a in sleep ()

(gdb) i r
rax            0xffffffffffffffdc  -516
rbx            0xffffffffffffffc0  -64
rcx            0x43c4e1            4441313
rdx            0x401b4d            4201293
rsi            0x7ffffe5129d0      140737460120016
rdi            0x7ffffe5129d0      140737460120016
rbp            0x0                 0x0
rsp            0x7ffffe5129d0      0x7ffffe5129d0
r8             0x2                 2
r9             0x2                 2
r10            0x7                 7
r11            0x246               582
r12            0x4029c0            4204992
r13            0x0                 0
r14            0x4a5018            4870168
r15            0x0                 0
rip            0x43c49a            0x43c49a <sleep+58>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0

(gdb) frame 2
#2  0x0000000000401c17 in internal_get_message ()
```

```
(gdb) i r
rax            0xfffffffffffffdfc  -516
rbx            0x400470            4195440
rcx            0x43c4e1            4441313
rdx            0x401b4d            4201293
rsi            0x7ffffe5129d0      140737460120016
rdi            0x7ffffe5129d0      140737460120016
rbp            0x7ffffe512a20      0x7ffffe512a20
rsp            0x7ffffe512a10      0x7ffffe512a10
r8             0x2                 2
r9             0x2                 2
r10            0x7                 7
r11            0x246               582
r12            0x4029c0            4204992
r13            0x0                 0
r14            0x4a5018            4870168
r15            0x0                 0
rip            0x401c17            0x401c17 <internal_get_message+128>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0

(gdb) frame 3
#3  0x0000000000401c57 in get_message ()

(gdb) i r
rax            0xfffffffffffffdfc  -516
rbx            0x400470            4195440
rcx            0x43c4e1            4441313
rdx            0x401b4d            4201293
rsi            0x7ffffe5129d0      140737460120016
rdi            0x7ffffe5129d0      140737460120016
rbp            0x7ffffe512a60      0x7ffffe512a60
rsp            0x7ffffe512a30      0x7ffffe512a30
r8             0x2                 2
r9             0x2                 2
r10            0x7                 7
r11            0x246               582
r12            0x4029c0            4204992
r13            0x0                 0
r14            0x4a5018            4870168
r15            0x0                 0
rip            0x401c57            0x401c57 <get_message+57>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0

(gdb) frame 4
#4  0x0000000000401cea in main ()
```

```
(gdb) i r
rax            0xffffffffffffffdc  -516
rbx            0x400470            4195440
rcx            0x43c4e1            4441313
rdx            0x401b4d            4201293
rsi            0x7ffffe5129d0      140737460120016
rdi            0x7ffffe5129d0      140737460120016
rbp            0x7ffffe512ab0      0x7ffffe512ab0
rsp            0x7ffffe512a70      0x7ffffe512a70
r8             0x2                 2
r9             0x2                 2
r10            0x7                 7
r11            0x246               582
r12            0x4029c0            4204992
r13            0x0                 0
r14            0x4a5018            4870168
r15            0x0                 0
rip            0x401cea            0x401cea <main+56>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
```

Note: It also appears that RBX changed between frame 1 and frame 2. This is because it was previously saved on the stack, so its value is taken from that location:

```
(gdb) disassemble sleep
Dump of assembler code for function sleep:
   0x000000000043c460 <+0>:     push   %rbp
   0x000000000043c461 <+1>:     push   %rbx
   0x000000000043c462 <+2>:     sub    $0x28,%rsp
   0x000000000043c466 <+6>:     mov    $0xffffffffffffffc0,%rbx
   0x000000000043c46d <+13>:    mov    %fs:0x28,%rax
   0x000000000043c476 <+22>:    mov    %rax,0x18(%rsp)
   0x000000000043c47b <+27>:    xor    %eax,%eax
   0x000000000043c47d <+29>:    mov    %edi,%eax
   0x000000000043c47f <+31>:    mov    %rsp,%rdi
   0x000000000043c482 <+34>:    mov    %rdi,%rsi
   0x000000000043c485 <+37>:    mov    %fs:(%rbx),%ebp
   0x000000000043c488 <+40>:    mov    %rax,(%rsp)
   0x000000000043c48c <+44>:    movq   $0x0,0x8(%rsp)
   0x000000000043c495 <+53>:    callq  0x43c4d0 <nanosleep>
   0x000000000043c49a <+58>:    test   %eax,%eax
   0x000000000043c49c <+60>:    js     0x43c4c0 <sleep+96>
   0x000000000043c49e <+62>:    mov    %ebp,%fs:(%rbx)
   0x000000000043c4a1 <+65>:    xor    %eax,%eax
   0x000000000043c4a3 <+67>:    mov    0x18(%rsp),%rdx
   0x000000000043c4a8 <+72>:    xor    %fs:0x28,%rdx
   0x000000000043c4b1 <+81>:    jne    0x43c4c5 <sleep+101>
   0x000000000043c4b3 <+83>:    add    $0x28,%rsp
   0x000000000043c4b7 <+87>:    pop    %rbx
   0x000000000043c4b8 <+88>:    pop    %rbp
   0x000000000043c4b9 <+89>:    retq
   0x000000000043c4ba <+90>:    nopw   0x0(%rax,%rax,1)
   0x000000000043c4c0 <+96>:    mov    (%rsp),%eax
   0x000000000043c4c3 <+99>:    jmp    0x43c4a3 <sleep+67>
```

```
   0x000000000043c4c5 <+101>:   callq  0x43edd0 <__stack_chk_fail_local>
End of assembler dump.
```

17. Some CPU registers can be recovered manually, such as RIP (saved address when using *call* instruction) and RSP (the stack pointer value 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, as we saw in the case of RBX). Let's recover some registers for the first few frames.

```
(gdb) frame 0
#0  0x000000000043c4e1 in nanosleep ()
```

Let's disassemble the current function:

```
(gdb) disassemble nanosleep
Dump of assembler code for function nanosleep:
   0x000000000043c4d0 <+0>:     mov    0x6c356(%rip),%eax        # 0x4a882c
<__libc_multiple_threads>
   0x000000000043c4d6 <+6>:     test   %eax,%eax
   0x000000000043c4d8 <+8>:     jne    0x43c4f0 <nanosleep+32>
   0x000000000043c4da <+10>:    mov    $0x23,%eax
   0x000000000043c4df <+15>:    syscall
=> 0x000000000043c4e1 <+17>:    cmp    $0xfffffffffffff000,%rax
   0x000000000043c4e7 <+23>:    ja     0x43c530 <nanosleep+96>
   0x000000000043c4e9 <+25>:    retq
   0x000000000043c4ea <+26>:    nopw   0x0(%rax,%rax,1)
   0x000000000043c4f0 <+32>:    push   %rbp
   0x000000000043c4f1 <+33>:    mov    %rsi,%rbp
   0x000000000043c4f4 <+36>:    push   %rbx
   0x000000000043c4f5 <+37>:    mov    %rdi,%rbx
   0x000000000043c4f8 <+40>:    sub    $0x18,%rsp
   0x000000000043c4fc <+44>:    callq  0x43ed00 <__libc_enable_asynccancel>
   0x000000000043c501 <+49>:    mov    %rbp,%rsi
   0x000000000043c504 <+52>:    mov    %rbx,%rdi
   0x000000000043c507 <+55>:    mov    %eax,%edx
   0x000000000043c509 <+57>:    mov    $0x23,%eax
   0x000000000043c50e <+62>:    syscall
   0x000000000043c510 <+64>:    cmp    $0xfffffffffffff000,%rax
   0x000000000043c516 <+70>:    ja     0x43c542 <nanosleep+114>
   0x000000000043c518 <+72>:    mov    %edx,%edi
   0x000000000043c51a <+74>:    mov    %eax,0xc(%rsp)
   0x000000000043c51e <+78>:    callq  0x43ed60 <__libc_disable_asynccancel>
   0x000000000043c523 <+83>:    mov    0xc(%rsp),%eax
   0x000000000043c527 <+87>:    add    $0x18,%rsp
   0x000000000043c52b <+91>:    pop    %rbx
   0x000000000043c52c <+92>:    pop    %rbp
   0x000000000043c52d <+93>:    retq
   0x000000000043c52e <+94>:    xchg   %ax,%ax
   0x000000000043c530 <+96>:    mov    $0xffffffffffffffc0,%rdx
   0x000000000043c537 <+103>:   neg    %eax
   0x000000000043c539 <+105>:   mov    %eax,%fs:(%rdx)
   0x000000000043c53c <+108>:   mov    $0xffffffff,%eax
   0x000000000043c541 <+113>:   retq
   0x000000000043c542 <+114>:   mov    $0xffffffffffffffc0,%rcx
   0x000000000043c549 <+121>:   neg    %eax
   0x000000000043c54b <+123>:   mov    %eax,%fs:(%rcx)
   0x000000000043c54e <+126>:   mov    $0xffffffff,%eax
   0x000000000043c553 <+131>:   jmp    0x43c518 <nanosleep+72>
End of assembler dump.
```

It is a short function. We see it overwrites EAX. Note that the EAX value is different:

```
(gdb) i r $rax
rax            0xfffffffffffffdfc  -516
```

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

```
(gdb) x/a $rsp
0x7ffffe5129c8: 0x43c49a <sleep+58>
```

Note: The RIP value of the caller, **0x43c49a**, is saved before the call, and RSP for the caller should be the value before the **call** instruction was executed. When a return address is saved, RSP is decremented by 8, so the value of RSP before the *nanosleep* call should be the current value of RSP pointing to the saved return address + 8:

```
(gdb) p/x $rsp+8
$17 = 0x7ffffe5129d0
```

Note: We see that for the next frame RIP is **0x43c49a <sleep+58>** and RSP is **0x7ffffe5129d0**. Let's now find out RIP and RSP for the next frame (the caller of the *sleep* function). To find out RSP pointing to the caller's saved RIP, we need to see how it was used in the callee, the *sleep* function, before the callee called *nanosleep*. We disassemble the *sleep* function:

```
(gdb) disassemble sleep
Dump of assembler code for function sleep:
   0x000000000043c460 <+0>:     push   %rbp
   0x000000000043c461 <+1>:     push   %rbx
   0x000000000043c462 <+2>:     sub    $0x28,%rsp
   0x000000000043c466 <+6>:     mov    $0xffffffffffffffc0,%rbx
   0x000000000043c46d <+13>:    mov    %fs:0x28,%rax
   0x000000000043c476 <+22>:    mov    %rax,0x18(%rsp)
   0x000000000043c47b <+27>:    xor    %eax,%eax
   0x000000000043c47d <+29>:    mov    %edi,%eax
   0x000000000043c47f <+31>:    mov    %rsp,%rdi
   0x000000000043c482 <+34>:    mov    %rdi,%rsi
   0x000000000043c485 <+37>:    mov    %fs:(%rbx),%ebp
   0x000000000043c488 <+40>:    mov    %rax,(%rsp)
   0x000000000043c48c <+44>:    movq   $0x0,0x8(%rsp)
   0x000000000043c495 <+53>:    callq  0x43c4d0 <nanosleep>
   0x000000000043c49a <+58>:    test   %eax,%eax
   0x000000000043c49c <+60>:    js     0x43c4c0 <sleep+96>
   0x000000000043c49e <+62>:    mov    %ebp,%fs:(%rbx)
   0x000000000043c4a1 <+65>:    xor    %eax,%eax
   0x000000000043c4a3 <+67>:    mov    0x18(%rsp),%rdx
   0x000000000043c4a8 <+72>:    xor    %fs:0x28,%rdx
   0x000000000043c4b1 <+81>:    jne    0x43c4c5 <sleep+101>
   0x000000000043c4b3 <+83>:    add    $0x28,%rsp
   0x000000000043c4b7 <+87>:    pop    %rbx
   0x000000000043c4b8 <+88>:    pop    %rbp
   0x000000000043c4b9 <+89>:    retq
   0x000000000043c4ba <+90>:    nopw   0x0(%rax,%rax,1)
   0x000000000043c4c0 <+96>:    mov    (%rsp),%eax
   0x000000000043c4c3 <+99>:    jmp    0x43c4a3 <sleep+67>
   0x000000000043c4c5 <+101>:   callq  0x43edd0 <__stack_chk_fail_local>
End of assembler dump.
```

We see that the stack pointer was decremented by 0x28 (*sub* instruction) and also by 16 (two *push* instructions), and so we add these values to RSP we found out previously for the *nanosleep* call, **0x7ffffe5129d0**:

```
(gdb) x/a 0x7ffffe5129d0 + 0x28 + 16
0x7ffffe512a08: 0x401c17 <internal_get_message+128>
```

We see that *sleep* was called from the *internal_get_message* function. The value of RSP before the call should be adjusted by 8 because of the saved return address:

```
(gdb) p/x 0x7ffffe512a08+8
$18 = 0x7ffffe512a10
```

18. We continue with a few more frames. We disassemble the caller of *sleep*, *internal_get_message*:

```
(gdb) disassemble internal_get_message
Dump of assembler code for function internal_get_message:
   0x0000000000401b97 <+0>:     push   %rbp
   0x0000000000401b98 <+1>:     mov    %rsp,%rbp
   0x0000000000401b9b <+4>:     sub    $0x10,%rsp
   0x0000000000401b9f <+8>:     mov    %rdi,-0x8(%rbp)
   0x0000000000401ba3 <+12>:    mov    %rsi,-0x10(%rbp)
   0x0000000000401ba7 <+16>:    cmpq   $0x0,-0x8(%rbp)
   0x0000000000401bac <+21>:    je     0x401c0d <internal_get_message+118>
   0x0000000000401bae <+23>:    mov    -0x8(%rbp),%rax
   0x0000000000401bb2 <+27>:    movq   $0x0,(%rax)
   0x0000000000401bb9 <+34>:    mov    -0x8(%rbp),%rax
   0x0000000000401bbd <+38>:    movq   $0x113,0x8(%rax)
   0x0000000000401bc5 <+46>:    mov    $0x1,%edx
   0x0000000000401bca <+51>:    mov    -0x8(%rbp),%rax
   0x0000000000401bce <+55>:    mov    %rdx,0x10(%rax)
   0x0000000000401bd2 <+59>:    lea    -0x8c(%rip),%rdx        # 0x401b4d <time_proc>
   0x0000000000401bd9 <+66>:    mov    -0x8(%rbp),%rax
   0x0000000000401bdd <+70>:    mov    %rdx,0x18(%rax)
   0x0000000000401be1 <+74>:    mov    -0x8(%rbp),%rax
   0x0000000000401be5 <+78>:    movl   $0x4578350,0x20(%rax)
   0x0000000000401bec <+85>:    mov    -0x8(%rbp),%rax
   0x0000000000401bf0 <+89>:    movl   $0x9c,0x24(%rax)
   0x0000000000401bf7 <+96>:    mov    -0x8(%rbp),%rax
   0x0000000000401bfb <+100>:   movl   $0x147,0x28(%rax)
   0x0000000000401c02 <+107>:   mov    -0x8(%rbp),%rax
   0x0000000000401c06 <+111>:   movl   $0x0,0x2c(%rax)
   0x0000000000401c0d <+118>:   mov    $0xffffffff,%edi
   0x0000000000401c12 <+123>:   callq  0x43c460 <sleep>
   0x0000000000401c17 <+128>:   mov    $0x0,%eax
   0x0000000000401c1c <+133>:   leaveq
   0x0000000000401c1d <+134>:   retq
End of assembler dump.
```

We see that the stack pointer was decremented by 0x10 (*sub* instruction) and also by 8 (*push* instruction), and so we add these values to RSP we found out previously for the *sleep* call, **0x7ffffe512a10**:

```
(gdb) x/a 0x7ffffe512a10 + 0x10 + 8
0x7ffffe512a28: 0x401c57 <get_message+57>
```

We see that *internal_get_message* was called from the *get_message* function. The value of RSP before the call should be adjusted by 8 because of the saved return address:

```
(gdb) p/x 0x7ffffe512a28+8
$19 = 0x7ffffe512a30
```

Now we disassemble the caller of *internal_get_message*, *get_message*:

```
(gdb) disassemble get_message
Dump of assembler code for function get_message:
   0x0000000000401c1e <+0>:     push   %rbp
   0x0000000000401c1f <+1>:     mov    %rsp,%rbp
   0x0000000000401c22 <+4>:     sub    $0x30,%rsp
   0x0000000000401c26 <+8>:     mov    %rdi,-0x18(%rbp)
   0x0000000000401c2a <+12>:    mov    %rsi,-0x20(%rbp)
   0x0000000000401c2e <+16>:    mov    %rdx,-0x28(%rbp)
   0x0000000000401c32 <+20>:    mov    %rcx,-0x30(%rbp)
   0x0000000000401c36 <+24>:    movl   $0xffffffff,-0x4(%rbp)
   0x0000000000401c3d <+31>:    cmpq   $0x0,-0x18(%rbp)
   0x0000000000401c42 <+36>:    je     0x401c5a <get_message+60>
   0x0000000000401c44 <+38>:    mov    -0x20(%rbp),%rdx
   0x0000000000401c48 <+42>:    mov    -0x18(%rbp),%rax
   0x0000000000401c4c <+46>:    mov    %rdx,%rsi
   0x0000000000401c4f <+49>:    mov    %rax,%rdi
   0x0000000000401c52 <+52>:    callq  0x401b97 <internal_get_message>
   0x0000000000401c57 <+57>:    mov    %eax,-0x4(%rbp)
   0x0000000000401c5a <+60>:    cmpl   $0x0,-0x4(%rbp)
   0x0000000000401c5e <+64>:    je     0x401c6c <get_message+78>
   0x0000000000401c60 <+66>:    mov    -0x4(%rbp),%eax
   0x0000000000401c63 <+69>:    mov    %eax,%edi
   0x0000000000401c65 <+71>:    callq  0x401b58 <set_last_ui_error>
   0x0000000000401c6a <+76>:    jmp    0x401c7b <get_message+93>
   0x0000000000401c6c <+78>:    mov    -0x20(%rbp),%rax
   0x0000000000401c70 <+82>:    mov    %rax,%rdi
   0x0000000000401c73 <+85>:    callq  0x401b6b <call_ui_hooks>
   0x0000000000401c78 <+90>:    mov    %eax,-0x4(%rbp)
   0x0000000000401c7b <+93>:    cmpl   $0x0,-0x4(%rbp)
   0x0000000000401c7f <+97>:    sete   %al
   0x0000000000401c82 <+100>:   movzbl %al,%eax
   0x0000000000401c85 <+103>:   leaveq
   0x0000000000401c86 <+104>:   retq
End of assembler dump.
```

We see that the stack pointer was decremented by 0x30 (*sub* instruction) and also by 8 (*push* instruction), and so we add these values to RSP we found out previously for the *sleep* call, **0x7ffffe512a30**:

```
(gdb) x/a 0x7ffffe512a30 + 0x30 + 8
0x7ffffe512a68: 0x401cea <main+56>
```

We see that *get_message* was called from the *main* function.

Note: We can reconstruct the stack trace like a debugger. It works whether or not the RBP pointer is used to track the previous RSP. Note that we can correctly disassemble functions using the **disassemble** command because function boundaries are saved in symbol files, or the start of the function is available from the image file section. If such information is not available, we would most likely have a truncated or incorrect stack trace:

```
(gdb) symbol-file
No symbol file now.
```

```
(gdb) bt
#0  0x000000000043c4e1 in ?? ()
#1  0x000000000043c49a in ?? ()
#2  0x00000000ffffffff0 in ?? ()
#3  0x00000000130f2d88 in ?? ()
#4  0x0000000000000140 in ?? ()
#5  0x3ebeea4a71786c00 in ?? ()
#6  0x000000000000000a in ?? ()
#7  0x0000000000400470 in ?? ()
#8  0x00007ffffe512a20 in ?? ()
#9  0x0000000000401c17 in ?? ()
#10 0x0000000000000000 in ?? ()
```

Note: In such a case, we may still find the beginning of the function heuristically by disassembling the return address backward and looking for the possible function prologue, for example:

```
(gdb) x/20i 0x43c49a-72
   0x43c452:    lea    0x43cbb(%rip),%edi        # 0x480113
   0x43c458:    callq  0x402b70
   0x43c45d:    nopl   (%rax)
   0x43c460:    push   %rbp
   0x43c461:    push   %rbx
   0x43c462:    sub    $0x28,%rsp
   0x43c466:    mov    $0xfffffffffffffc0,%rbx
   0x43c46d:    mov    %fs:0x28,%rax
   0x43c476:    mov    %rax,0x18(%rsp)
   0x43c47b:    xor    %eax,%eax
   0x43c47d:    mov    %edi,%eax
   0x43c47f:    mov    %rsp,%rdi
   0x43c482:    mov    %rdi,%rsi
   0x43c485:    mov    %fs:(%rbx),%ebp
   0x43c488:    mov    %rax,(%rsp)
   0x43c48c:    movq   $0x0,0x8(%rsp)
   0x43c495:    callq  0x43c4d0
   0x43c49a:    test   %eax,%eax
   0x43c49c:    js     0x43c4c0
   0x43c49e:    mov    %ebp,%fs:(%rbx)
```

19. 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**.

20. We also introduce special **Stack Frame** memory cell diagrams. For example, the case of the stack frame for the *get_message* function before calling *internal_get_message* is illustrated in the MCD-R1.xlsx section D (offsets are hexadecimal), where [RSP+38] corresponds to the stored return address of the *get_message* caller, *main*.