

Binary Reverse Engineering And Analysis

Course 5: Stack frames 101

Caragea Radu

March 16, 2021

Recap: stack micro-operations (demo 1)

```
POP RAX          ; rax = *(int64_t*)rsp; rsp += 8
PUSH RAX        ; rsp -= 8; *(int64_t*)rsp = rax;

CALL 0x12345    ; PUSH RIP; JMP 0x12345

RET             ; POP RIP
```

Recap: stack macro-operations

```
PUSH RBP ; save previous frame base
MOV RBP, RSP ; move frame base to current top
SUB RSP, 100 ; allocate 100 bytes on the stack
              ; "push new stack frame"

MOV RBX, [RBP - 0x20] ; rbx = *(int64_t*)(rbp-0x20)
                      ; use the allocated space for storage

LEAVE ; MOV RSP, RBP ; POP RBP
       ; "pop current stack frame"
```

Today

- Better understanding of stack variable allocation
- Better understanding of function calls
- Common vulnerabilities
- Ways to exploit
- Next time: ways to prevent

Stack visualization: 1 buffer

The image shows a debugger interface with two panes. The left pane is a C source editor titled "C source #1" containing the following code:

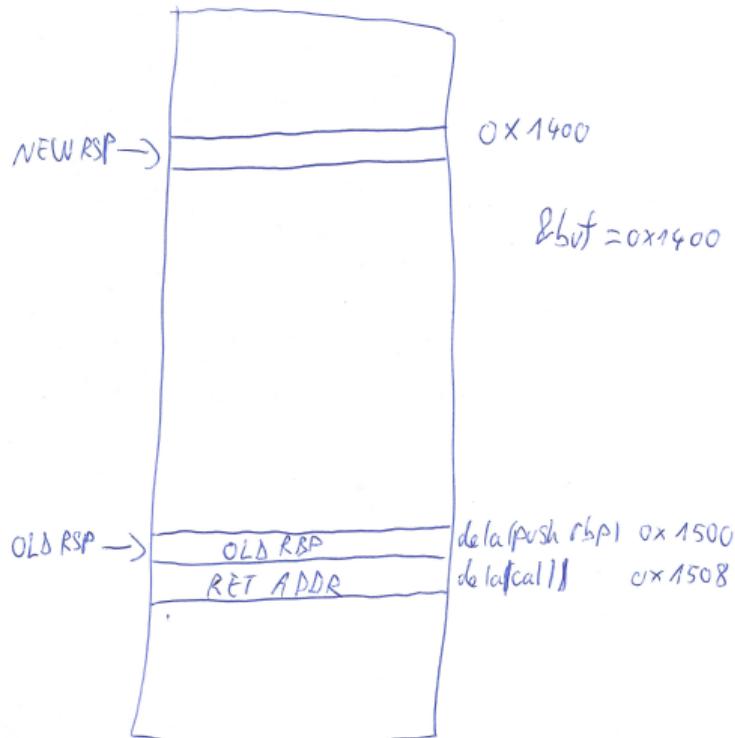
```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void stack_frame_demo(void)
6 {
7     char buf[256];
8     fgets(buf, 256, stdin);
9 }
```

The code is syntax-highlighted, with the buffer declaration and the fgets call highlighted in yellow. The right pane is an assembly editor titled "x86-64 gcc 8.2 (Editor #1, Compiler #1) C" showing the generated assembly code:

```
1 stack_frame_demo:
2     push    rbp
3     mov     rbp, rsp
4     sub    rsp, 256
5     mov     rdx, QWORD PTR stdin[rip]
6     lea     rax, [rbp-256]
7     mov     esi, 256
8     mov     rdi, rax
9     call   fgets
10    nop
11    leave
12    ret
```

The assembly code is also syntax-highlighted, with the stack frame setup (push rbp, mov rbp, sub rsp) highlighted in light blue, the fgets call (mov rdx, lea rax, mov esi, mov rdi, call fgets) highlighted in yellow, and the cleanup (nop, leave, ret) highlighted in light purple.

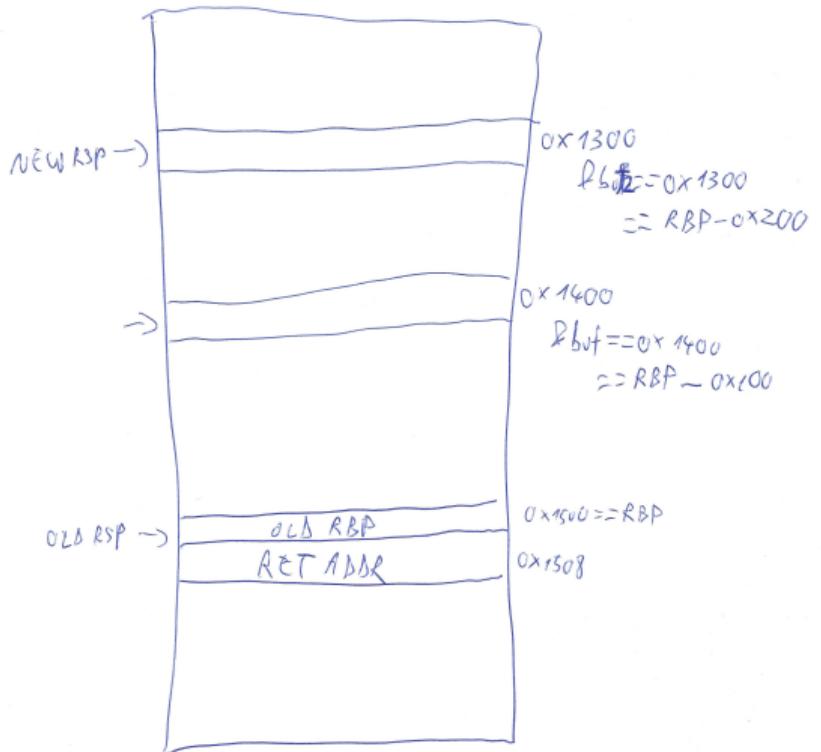
Stack visualization: 1 buffer



Stack visualization: 2 buffers (1/2)

The image shows two windows side-by-side. The left window is a C source code editor titled "C source #1" with the file type set to "C". It contains the following code:1 #include <stdio.h>
2 #include <string.h>
3
4
5 void stack_frame_demo(void)
6 {
7 char buf[256];
8 char buf2[256];
9 fgets(buf, 256, stdin);
10}Lines 6 through 9 are highlighted with different colors: line 6 is green, line 7 is yellow, line 8 is light blue, and line 9 is light green. The right window is a debugger titled "x86-64 gcc 8.2 (Editor #1, Compiler #1) C" showing assembly output. The assembly code is:1 stack_frame_demo:
2 push rbp
3 mov rbp, rsp
4 sub rsp, 512
5 mov rdx, QWORD PTR stdin[rip]
6 lea rax, [rbp-256]
7 mov esi, 256
8 mov rdi, rax
9 call fgets
10 nop
11 leave
12 retLines 1 through 12 are colored to match the corresponding lines in the C code: line 1 is cyan, lines 2-4 are light blue, lines 5-9 are yellow, line 10 is light green, line 11 is light blue, and line 12 is light green.

Stack visualization: 2 buffers (1/2)



Stack visualization: 2 buffers (2/2)

The image shows two windows side-by-side. The left window is a C source code editor titled "C source #1" with the file extension ".c". It contains the following code:1 #include <stdio.h>
2 #include <string.h>
3
4
5 void stack_frame_demo(void)
6 {
7 char buf2[256];
8 char buf[256];
9 fgets(buf, 256, stdin);
10 }Lines 6 through 9 are highlighted with a light yellow background. The right window is an assembly editor titled "x86-64 gcc 8.2 (Editor #1, Compiler #1) C" with the file extension ".c". It shows the assembly output for the code above:1 stack_frame_demo:
2 push rbp
3 mov rbp, rsp
4 sub rsp, 512
5 mov rdx, QWORD PTR stdin[rip]
6 lea rax, [rbp-512]
7 mov esi, 256
8 mov rdi, rax
9 call fgets
10 nop
11 leave
12 retLines 5 through 12 are highlighted with a light yellow background, corresponding to the highlighted lines in the C code.

Stack visualization: 2 ints (1/2)

The image shows a debugger interface with two panes. The left pane displays the C source code, and the right pane displays the generated assembly code.

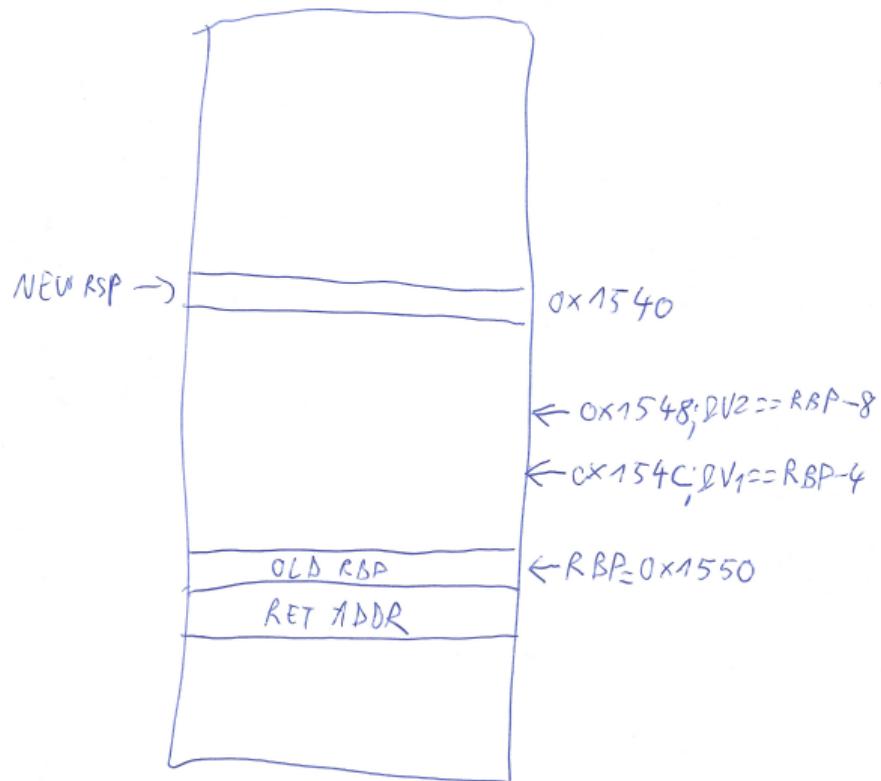
C source #1:

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void stack_frame_demo(void)
6 {
7     int v1;
8     int v2;
9     scanf("%d %d\n", &v1, &v2);
10 }
```

x86-64 gcc 8.2 (Editor #1, Compiler #1) C:

```
1 .LC0:
2     .string "%d %d\n"
3 stack_frame_demo:
4     push    rbp
5     mov     rbp, rsp
6     sub    rsp, 16
7     lea    rdx, [rbp-8]
8     lea    rax, [rbp-4]
9     mov    rsi, rax
10    mov    edi, OFFSET FLAT:.LC0
11    mov    eax, 0
12    call   __isoc99_scanf
13    nop
14    leave
15    ret
```

Stack visualization: 2 ints (1/2)



Stack visualization: 2 ints (2/2)

The image shows two windows side-by-side. The left window is a C code editor titled "C source #1" with the file type set to "C". It contains the following code:1 #include <stdio.h>
2 #include <string.h>
3
4
5 void stack_frame_demo(void)
6 {
7 int v2;
8 int v1;
9 scanf("%d %d\n", &v1, &v2);
10}The code is highlighted with syntax coloring. The right window is a debugger titled "x86-64 gcc 8.2 (Editor #1, Compiler #1) C" with the file type set to "C". It shows the assembly output for the same code:1 .LC0:
2 .string "%d %d\n"
3 stack_frame_demo:
4 push rbp
5 mov rbp, rsp
6 sub rsp, 16
7 lea rdx, [rbp-4]
8 lea rax, [rbp-8]
9 mov rsi, rax
10 mov edi, OFFSET FLAT:.LC0
11 mov eax, 0
12 call __isoc99_scanf
13 nop
14 leave
15 retThe assembly code is also highlighted with syntax coloring. The assembly code corresponds to the C code, showing the stack frame setup, variable declarations, and the `scanf` function call.

Stack visualization: combination (1/2)

The screenshot shows two windows side-by-side. The left window is titled "C source #1" and contains C code. The right window is titled "x86-64 gcc 8.2 (Editor #1, Compiler #1)" and contains the generated assembly code.

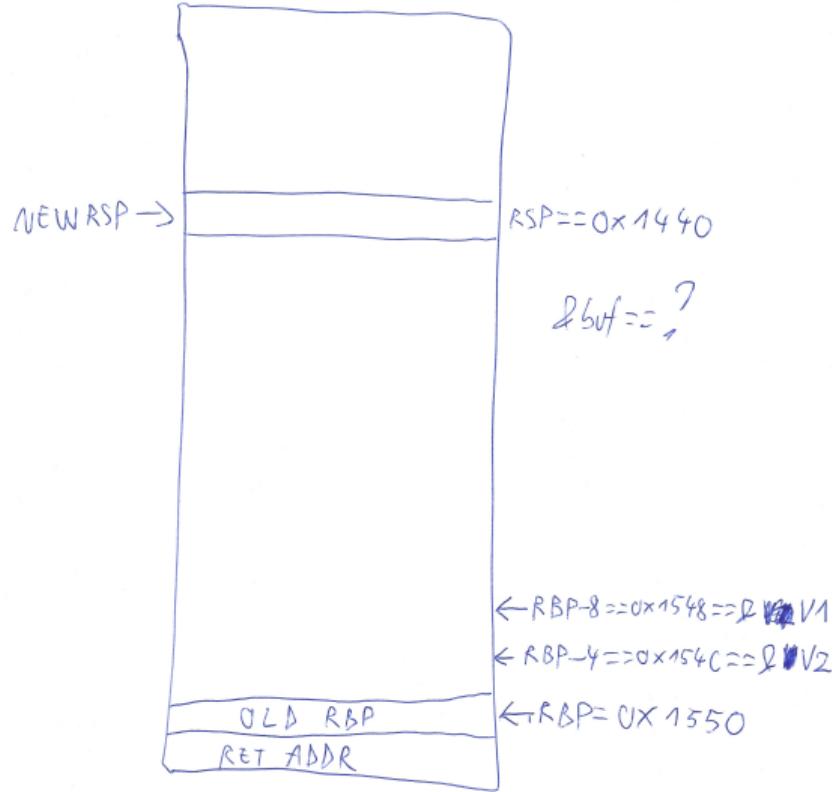
C source #1:

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void stack_frame_demo(void)
6 {
7
8     int v2;
9     int v1;
10    char buf[256];
11    scanf("%d %d\n", &v1, &v2);
12 }
```

x86-64 gcc 8.2:

```
1 .LC0:
2     .string "%d %d\n"
3 stack_frame_demo:
4     push   rbp
5     mov    rbp, rsp
6     sub    rsp, 272
7     lea    rdx, [rbp-4]
8     lea    rax, [rbp-8]
9     mov    rsi, rax
10    mov   edi, OFFSET FLAT:.LC0
11    mov   eax, 0
12    call  __isoc99_scanf
13    nop
14    leave
15    ret
```

Stack visualization: combination (1/2)



Stack visualization: combination (2/2)

The image shows a debugger interface with two panes. The left pane displays the C source code, and the right pane displays the generated assembly code.

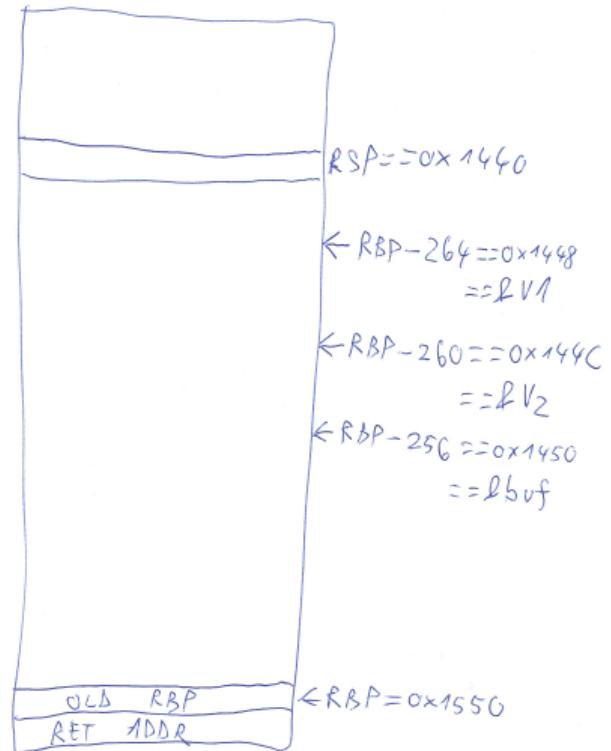
C source #1

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void stack_frame_demo(void)
6 {
7     char buf[256];
8     int v2;
9     int v1;
10
11     scanf("%d %d\n", &v1, &v2);
12 }
```

x86-64 gcc 8.2 (Editor #1, Compiler #1) C

```
1 .LC0:
2     .string "%d %d\n"
3 stack_frame_demo:
4     push    rbp
5     mov     rbp, rsp
6     sub    rsp, 272
7     lea     rdx, [rbp-260]
8     lea     rax, [rbp-264]
9     mov     rsi, rax
10    mov     edi, OFFSET FLAT:.LC0
11    mov     eax, 0
12    call    __isoc99_scanf
13    nop
14    leave
15    ret
```

Stack visualization: combination (2/2)



Stack visualization: variable length buffers

The image shows a debugger interface with two panes. The left pane displays the C source code for a function named `stack_frame_demo`. The right pane shows the generated assembly code for the same function, produced by the x86-64 gcc 8.2 compiler.

C source #1:

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void stack_frame_demo(long n)
6 {
7     char buf[n];
8     fgets(buf,n,stdin);
9 }
```

x86-64 gcc 8.2 (Editor #1, Compiler #1) C -O1:

```
1 stack_frame_demo:
2     push    rbp
3     mov     rbp, rsp
4     lea     rax, [rdi+15]
5     and     rax, -16
6     sub     rsp, rax
7     mov     rdx, QWORD PTR stdin[rip]
8     mov     esi, edi
9     mov     rdi, rsp
10    call    fgets
11    leave
12    ret
```

Vulnerability 1: locality (demo 1)

- Since all variables are "packed", mishaps can happen

Vulnerability 1: locality (demo 1)

- Since all variables are "packed", mishaps can happen
- Buffers read improperly can overflow (spill) into adjacent variables
- In extreme cases, the overflow can hijack the execution
- Let's see a DEMO!

Demo 1 key takeaway

```
void stack_vuln_demo()
{
    char buf[264]; // [rsp+0h] [rbp-110h]
    unsigned int v1; // [rsp+108h] [rbp-8h]
    unsigned int v2; // [rsp+10Ch] [rbp-4h]

    __isoc99_scanf("%d %d %s", &v2, &v1, buf);
    printf("You entered: %d and %d\n", v2, v1);
}

-0000000000000000110 ; D/A/*      : change type (data/ascii/array)
-0000000000000000110 ; N        : rename
-0000000000000000110 ; U        : undefined
-0000000000000000110 ; Use data definition commands to create local variables and function arguments.
-0000000000000000110 ; Two special fields " r" and " s" represent return address and saved registers.
-0000000000000000110 ; Frame size: 110; Saved regs: 8; Purge: 0
-0000000000000000110 ;
-0000000000000000110
-0000000000000000110 buf          db 264 dup(?)
-0000000000000008 var_8        dd ?
-0000000000000004 var_4        dd ?
+0000000000000000 s           db 8 dup(?)
+0000000000000008 r           db 8 dup(?)
+0000000000000010
+0000000000000010 ; end of stack variables
```

Function call recap (demo 2)

- We now know a bit about debuggers
- Let's see a function call DEMO

Function return hijack (demo 3)

- Functions (usually) return to the call site
- The call site (return address) is stored on the stack
- When other variables cannot be overflowed: ret addr
- Let's see another DEMO

Vulnerability 2: Reuse (demo 4)

■ https://godbolt.org/z/92rh_U

```
#include <stdlib.h>
#include <stdio.h>
void f1(){
    char buf[256];
    scanf("%s", buf);
}

void f2(){
    char buf[256];
    printf("%s\n", buf);
}

int main()
{
    f1();
    f2();
}
```

```
1 .LC0:
2     .string "%s"
3 f1:
4     push  rbp
5     mov   rbp,  rsp
6     sub   rsp,  256
7     lea   rax,  [rbp-256]
8     mov   rsi,  rax
9     mov   edi,  OFFSET FLAT:.LC0
10    mov   eax,  0
11    call  __isoc99_scanf
12    nop
13    leave
14    ret
15 f2:
16    push  rbp
17    mov   rbp,  rsp
18    sub   rsp,  256
19    lea   rax,  [rbp-256]
20    mov   rdi,  rax
21    call  puts
22    nop
23    leave
24    ret
25 main:
26    push  rbp
27    mov   rbp,  rsp
28    mov   eax,  0
29    call  f1
30    mov   eax,  0
31    call  f2
32    mov   eax,  0
```

Practice

- Any Questions?
- http://pwnthybytes.ro/unibuc_re/05-lab.html