

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

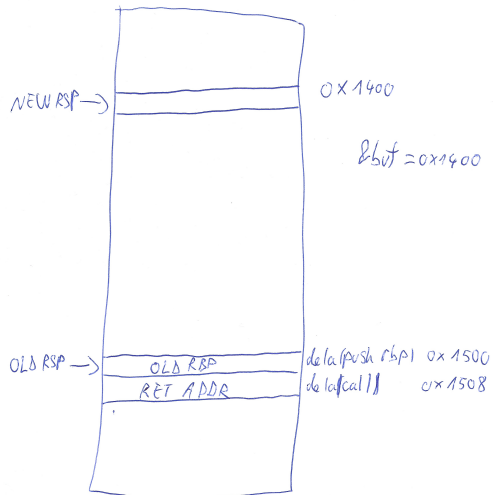
Stack visualization: 1 buffer

The image shows a side-by-side comparison of C source code and its assembly output. The left window, titled 'C source #1', displays the source code for a function named 'stack_frame_demo'. The code includes headers for 'stdio.h' and 'string.h', and defines a function that declares a 256-byte character buffer 'buf' and calls 'fgets' to read input from 'stdin'. The right window, titled 'x86-64 gcc 8.2 (Editor #1, Compiler #1) C', shows the assembly code generated for this function. The assembly includes instructions for pushing the base pointer, adjusting the stack pointer, moving the address of 'stdin' into 'rdx', calculating the address of the buffer, and then calling 'fgets'. The assembly code is color-coded to match the source code: light green for the function definition, yellow for the buffer declaration and 'fgets' call, and light purple for the return and stack cleanup instructions.

```
C source #1 x
A Save/Load + Add new... C
1 #include <stdio.h>
2 #include <string.h>
3
4 void stack_frame_demo(void)
5 {
6     char buf[256];
7     fgets(buf, 256, stdin);
8 }
9
10
11
12

x86-64 gcc 8.2 (Editor #1, Compiler #1) C x
x86-64 gcc 8.2 Compiler options...
A 11010 .LX0: lib.f: .text // \s+
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
```

Stack visualization: 1 buffer



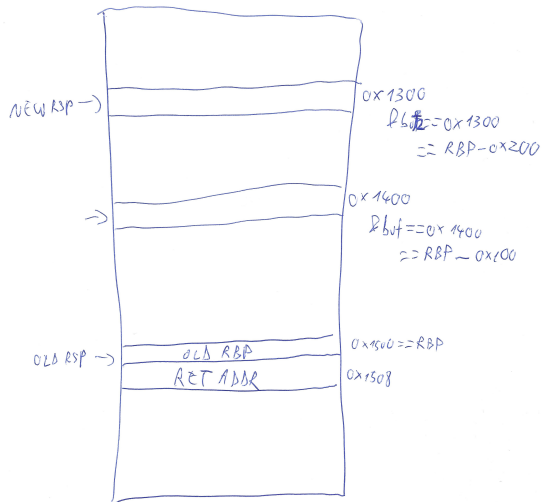
Stack visualization: 2 buffers (1/2)

The image shows a code editor with two panes. The left pane displays C source code for a function named `stack_frame_demo`. The right pane displays the corresponding assembly code for the same function, generated by gcc 8.2.

```
C source #1 x
A Save/Load + Add new... C
1 #include <stdio.h>
2 #include <string.h>
3
4 void stack_frame_demo(void)
5 {
6     char buf[256];
7     char buf2[256];
8     fgets(buf, 256, stdin);
9 }
10
11
12
13

x86-64 gcc 8.2 (Editor #1, Compiler #1) C x
x86-64 gcc 8.2 Compiler options...
A 11010 .LX0: lib.f: .text // \s+
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    ret
```

Stack visualization: 2 buffers (1/2)



Stack visualization: 2 buffers (2/2)

The image shows a code editor with two panes. The left pane displays C source code for a function named `stack_frame_demo`. The right pane shows the corresponding assembly code generated by the compiler.

```
C source #1 x
Save/Load + Add new... C
1 #include <stdio.h>
2 #include <string.h>
3
4 void stack_frame_demo(void)
5 {
6     char buf2[256];
7     char buf[256];
8     fgets(buf, 256, stdin);
9 }
10
11
12
13
```

```
x86-64 gcc 8.2 (Editor #1, Compiler #1) C x
x86-64 gcc 8.2 Compiler options...
A 11010 .LX0: lib.f: .text // \s+
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    ret
```

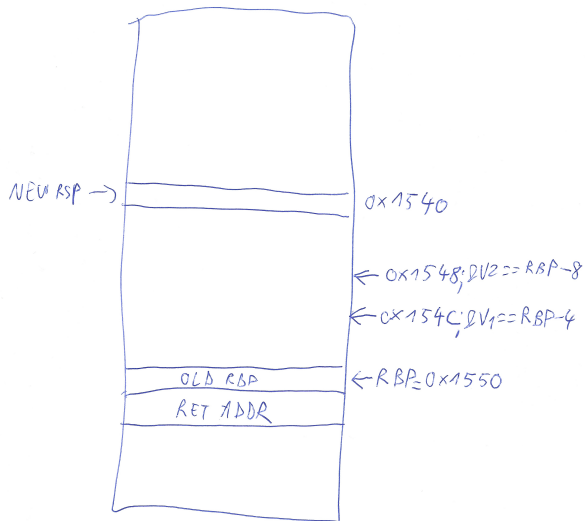
Stack visualization: 2 ints (1/2)

The image shows a code editor with two panes. The left pane displays C source code for a function named `stack_frame_demo`. The right pane displays the corresponding assembly code generated by x86-64 gcc 8.2. The assembly code shows stack frame setup, including pushing `rbp`, moving `rsp` to `rbp`, and subtracting 16 from `rsp`. It then loads the addresses of `v1` and `v2` into `rdx` and `rax`, moves `rsi` to `rax`, and calls `__isoc99_scanf` with the format string `"%d %d\n"` and the addresses of `v1` and `v2`. Finally, it uses `leave` to restore `rbp` and `ret` to return from the function.

```
C source #1 x
A Save/Load + Add new... C
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 }
11
12
13

x86-64 gcc 8.2 (Editor #1, Compiler #1) C x
x86-64 gcc 8.2 Compiler options...
A 11010 [X] .LX0: [ ] lib.f: [X] .text [X] // [ ] \s+ [X]
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 a code editor with two panes. The left pane displays C source code for a function named `stack_frame_demo`. The right pane displays the corresponding assembly code generated by gcc 8.2. The assembly code shows stack frame setup, including pushing the base pointer (rbp), moving the stack pointer (rsp) to rbp, and subtracting 16 bytes from rsp. It then loads the addresses of local variables v1 and v2 from the stack into rax and rdx, respectively, and calls the `__isoc99_scanf` function to read two integers from standard input.

```
C source #1 x
A Save/Load + Add new... C
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 }
11
12
13

x86-64 gcc 8.2 (Editor #1, Compiler #1) C x
x86-64 gcc 8.2 Compiler options...
A 11010 .LX0: lib.f: .text // \s+
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    ret
```

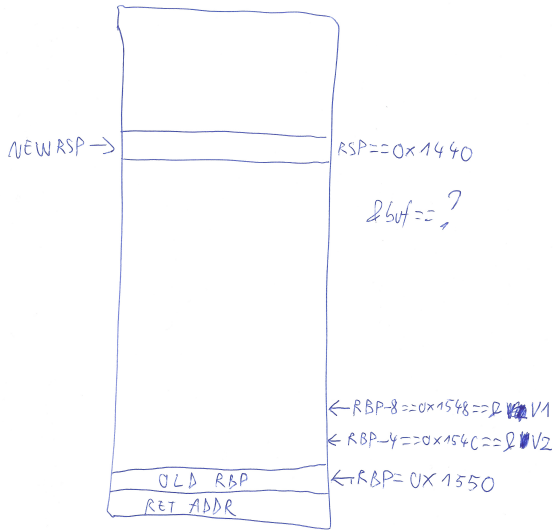
Stack visualization: combination (1/2)

The image shows a side-by-side comparison of C source code and its assembly output. The left window, titled 'C source #1', displays the source code for a function named 'stack_frame_demo'. The right window, titled 'x86-64 gcc 8.2 (Editor #1, Compiler #1) C', shows the corresponding assembly code generated by the compiler. The assembly code includes instructions for setting up the stack frame, pushing the base pointer, moving the stack pointer, and calling the 'scanf' function.

```
C source #1 x
A Save/Load + Add new... C
1 #include <stdio.h>
2 #include <string.h>
3
4 void stack_frame_demo(void)
5 {
6
7
8     int v2;
9     int v1;
10    char buf[256];
11    scanf("%d %d\n", &v1, &v2);
12 }
13
14
15

x86-64 gcc 8.2 (Editor #1, Compiler #1) C x
x86-64 gcc 8.2 Compiler options...
A 11010 [X] .LX0: [ ] lib.f: [X] .text [X] // [ ] \s+ [X]
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
14    nop
15    leave
16    ret
```

Stack visualization: combination (1/2)



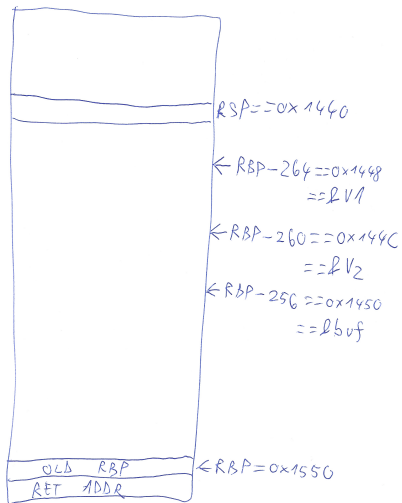
Stack visualization: combination (2/2)

The image shows a side-by-side comparison of C source code and its assembly output. The left window, titled 'C source #1', displays the source code for a function named 'stack_frame_demo'. The right window, titled 'x86-64 gcc 8.2 (Editor #1, Compiler #1) C', shows the corresponding assembly code generated by the compiler. The assembly code includes a label '.LC0' for a string constant and the function 'stack_frame_demo' which performs stack frame setup, pushes arguments, calls 'scanf', and then returns.

```
C source #1 x
A Save/Load + Add new... C
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 }
13
14
15

x86-64 gcc 8.2 (Editor #1, Compiler #1) C x
x86-64 gcc 8.2 Compiler options...
A 11010 .LX0: lib.f: .text // \s+
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 code editor with two panes. The left pane displays C source code for a function named `stack_frame_demo`. The right pane displays the corresponding assembly code for the same function, generated by GCC 8.2. The assembly code shows stack frame setup, including pushing `rbp`, moving `rsp` to `rbp`, and allocating space for a buffer by adjusting `rsp` by `-16`. It then calls `fgets` to read input into the buffer.

```
C source #1 x
Save/Load + Add new... C
1 #include <stdio.h>
2 #include <string.h>
3
4 void stack_frame_demo(long n)
5 {
6     char buf[n];
7     fgets(buf,n,stdin);
8 }
9
10
11
12

x86-64 gcc 8.2 (Editor #1, Compiler #1) C x
x86-64 gcc 8.2 -O1
A 11010 .LX0: lib.f: .text // \s+
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]        | overflow direction
  unsigned int v2; // [rsp+10Ch] [rbp-4h]        | v
  __isoc99_scanf("%d %d %s", &v2, &v1, buf);
  printf("You entered: %d and %d\n", v2, v1);
}
```

```
-00000000000000110 ; D/A/* : change type (data/ascii/array)
-00000000000000110 ; N : rename
-00000000000000110 ; U : undefine
-00000000000000110 ; Use data definition commands to create local variables and function arguments.
-00000000000000110 ; Two special fields " r" and " s" represent return address and saved registers.
-00000000000000110 ; Frame size: 110; Saved regs: 8; Purge: 0
-00000000000000110 ;
-00000000000000110 ;
-00000000000000110 buf db 264 dup(?)
-00000000000000008 var_8 dd ?
-00000000000000004 var_4 dd ?
+00000000000000000 s db 8 dup(?)
+00000000000000008 r db 8 dup(?)
+00000000000000010
+00000000000000010 ; 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`