Smashing The Stack

A detailed look at buffer overflows as described in

Smashing the Stack for Fun and Profit

by Aleph One

Process Memory Organization

- Text
  - Fixed by program
  - Includes code and read-only data
    - Since read-only, attempts to write to this typically cause seg fault.
- Data
  - Static variables (both initialized and uninitialized)
- Stack
  - Usual LIFO data structure
  - Used because well suited for procedure calls
  - Used for dynamic allocation of local variables, passing of parameters, returning values from functions

Stack Region

- Stack is a contiguous block of memory containing data
  - Size dynamically adjusted by OS kernel at runtime
- Stack pointer (SP) register: points to top of stack
  - Bottom of stack at fixed address
- Stack Frame
  - Parameters to a function
  - Local variables of function
  - Data necessary to recover previous stack frame
    - Including value of instruction pointer (IP) at time of function call
  - PUSHed onto stack on function call, POPped on return

Stack Region

- Assumptions
  - Stack grows down (toward lower addresses)
  - SP points to last address on stack (as opposed to pointing to next free available address)
- Frame Pointer (FP) a.k.a. local base pointer (LP)
  - Points to fixed location within frame
  - Local variables and parameters referenced via FP because their distance from FP do not change with PUSHes and POPs
    - Actual parameters PUSHed before new frame creation, so have positive offsets, local variables after, so negative offsets
  - On Intel CPUs, the EBP (32-bit BP) register is used

On Procedure Call…

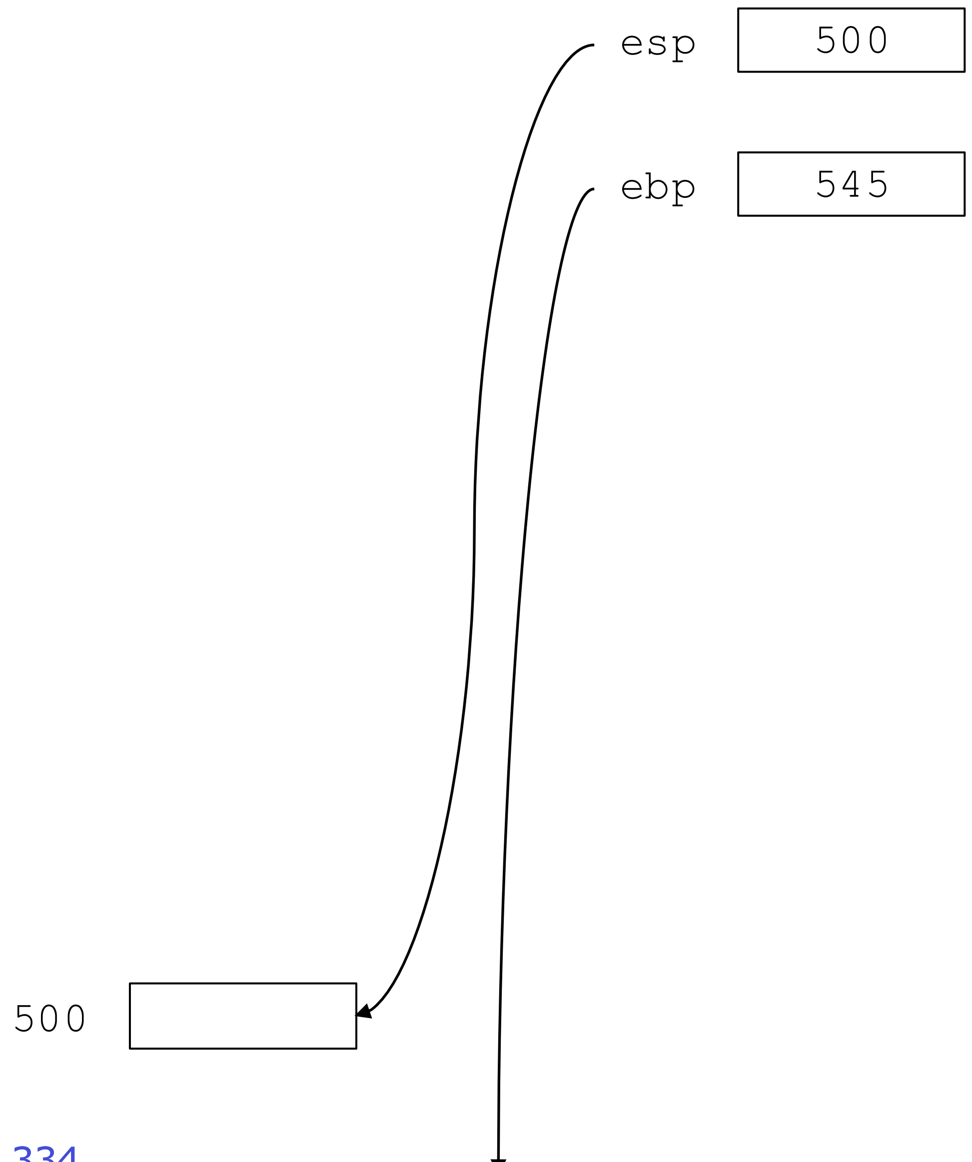
- Procedure prolog (start of call)
  - Save previous FP (to be restored at proc. exit)
  - Copy SP into FP to create new FP
  - Advance SP to reserve space for local variables

- Procedure epilogue (end of procedure)
  - Stack is cleaned up and restored to previous state

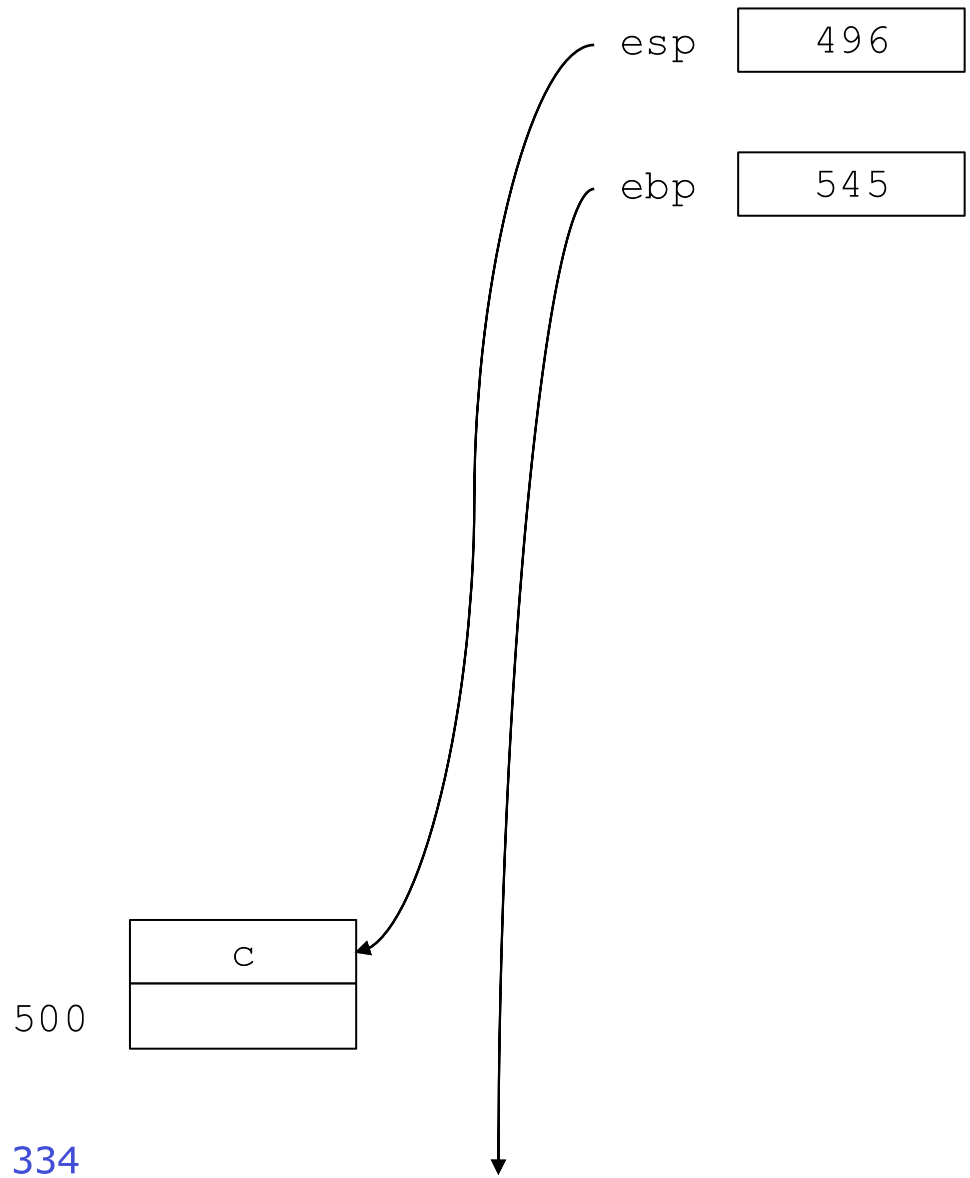
- Often special instructions to handle these
  - Intel: ENTER and LEAVE
  - Motorola: LINK and UNLINK

Example

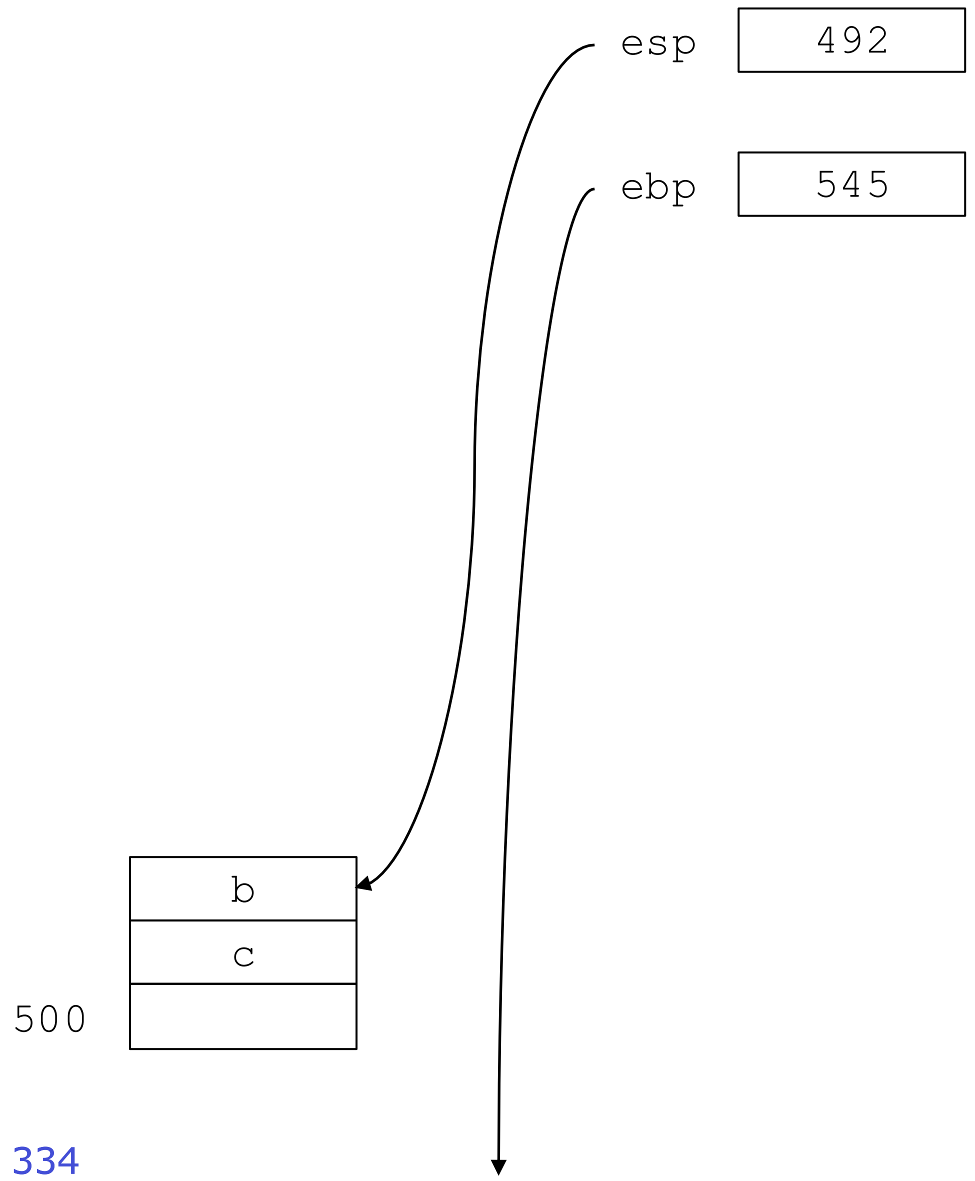
```
example1.c:
------------------------------------------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
------------------------------------------
```


esp
500
ebp
545
500

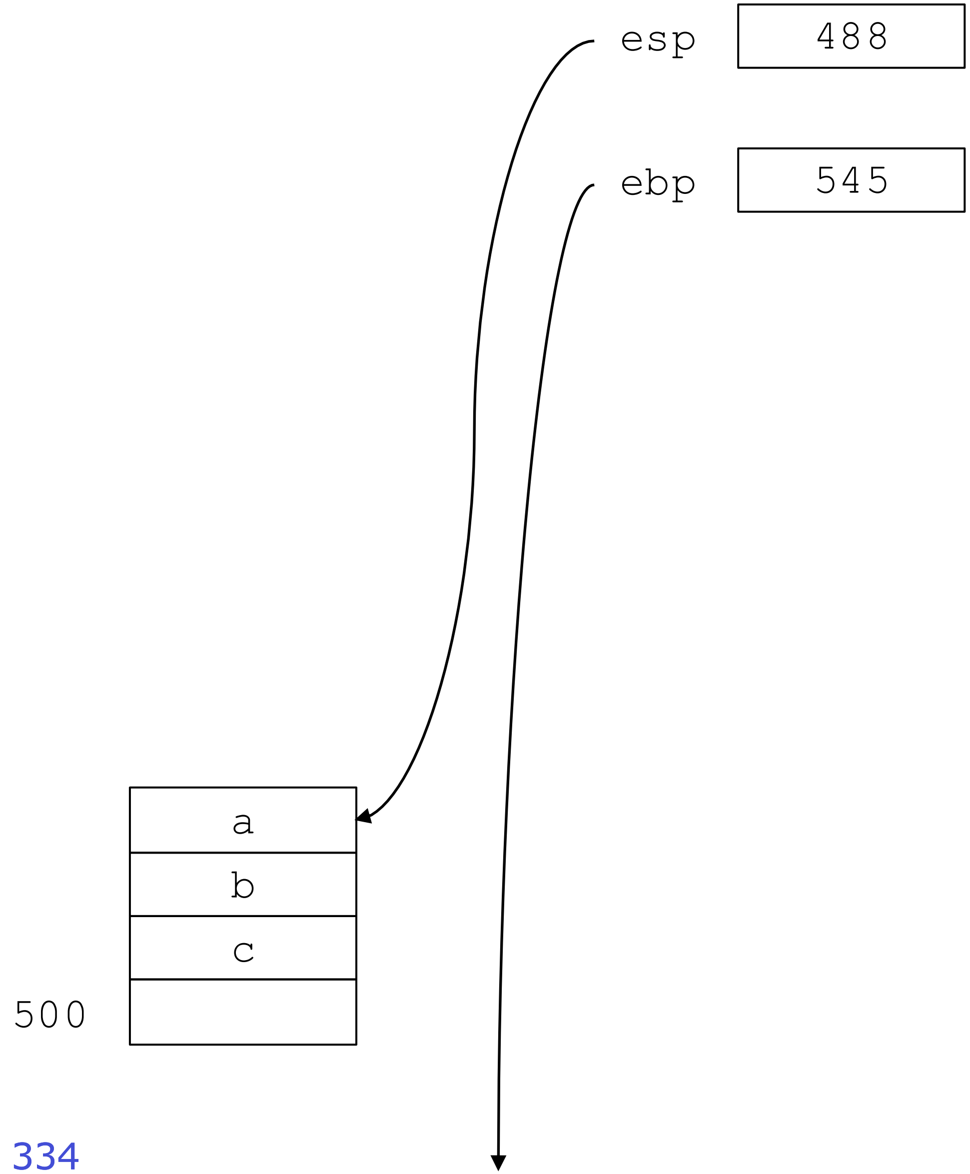
```
pushl $3
```



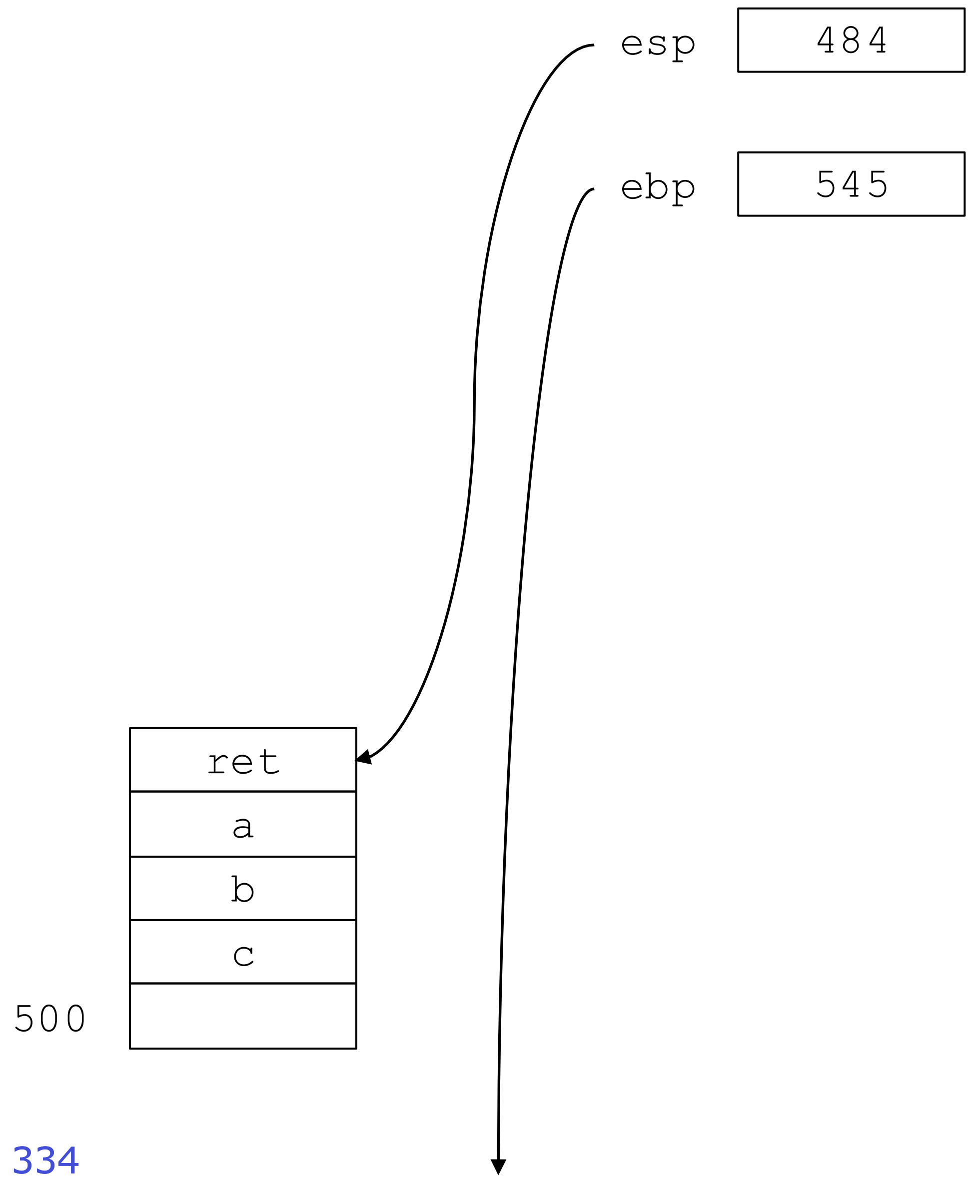
```
pushl $3
pushl $2
```



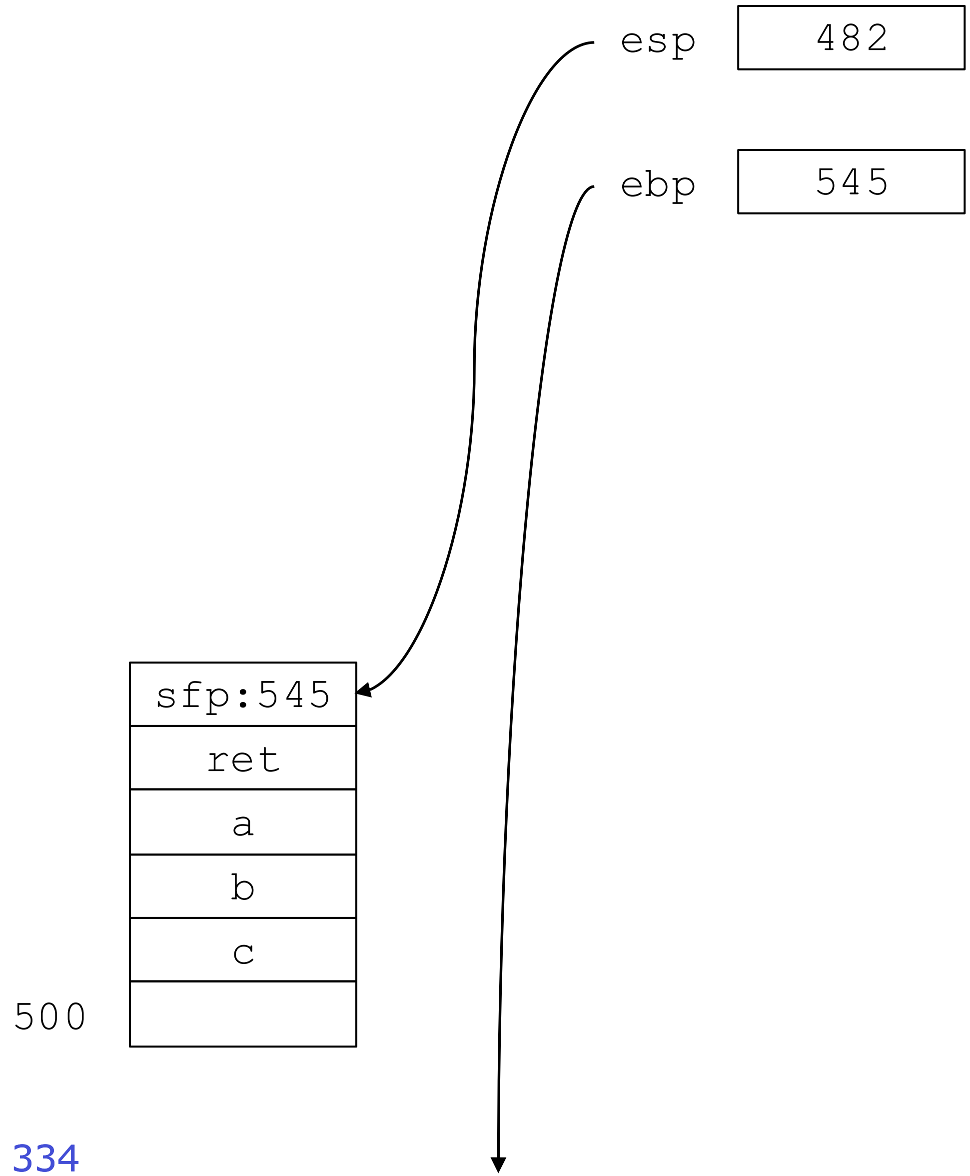
```
pushl $3
pushl $2
pushl $1
```



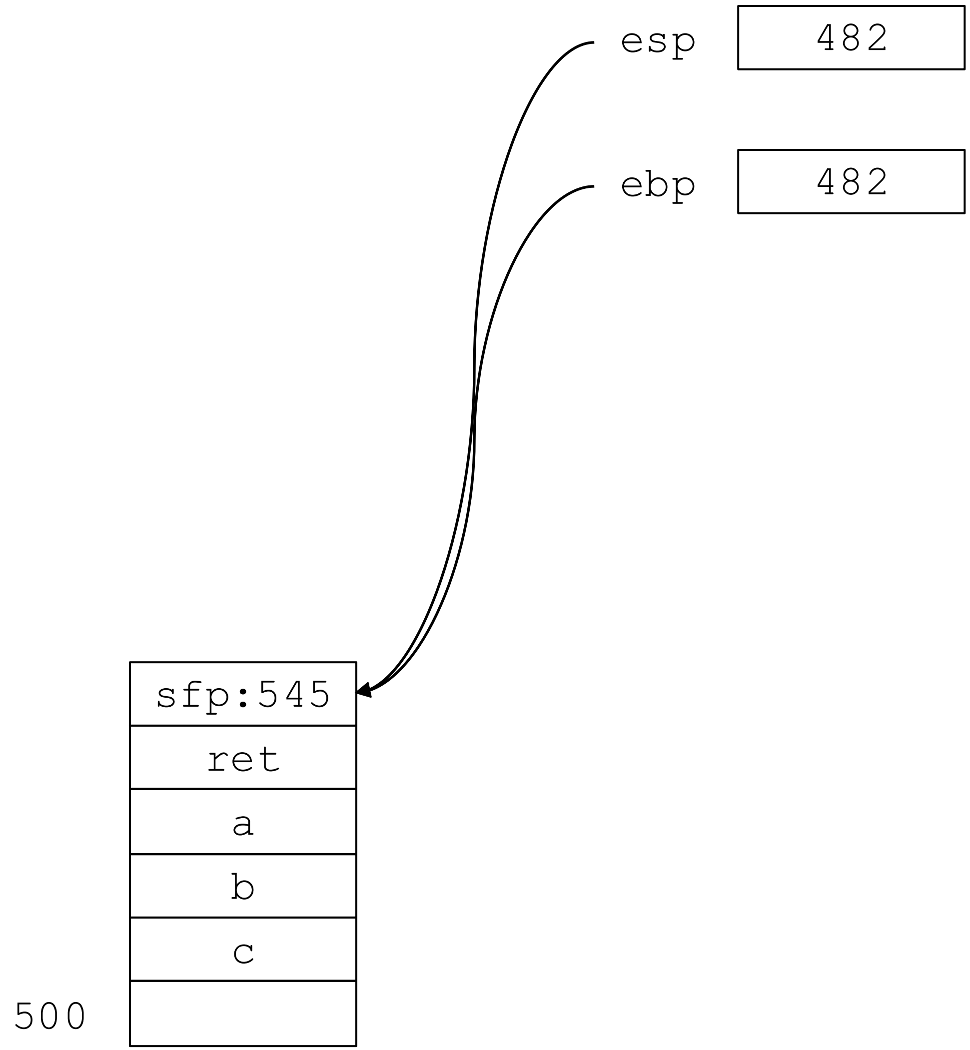
```
pushl $3
pushl $2
pushl $1
call function
```



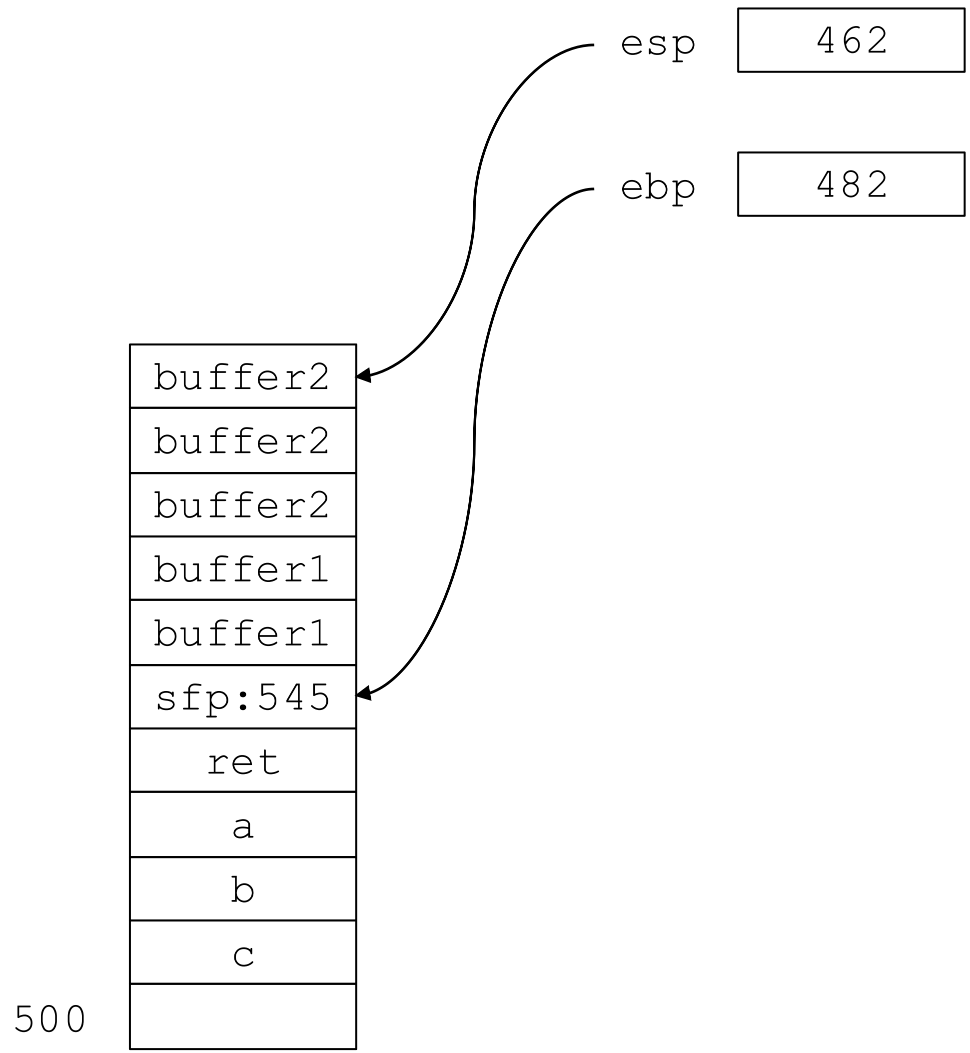
```
pushl $3
pushl $2
pushl $1
call function
pushl %ebp
```



```
pushl $3
pushl $2
pushl $1
call function
pushl %ebp
movl %esp,%ebp
```



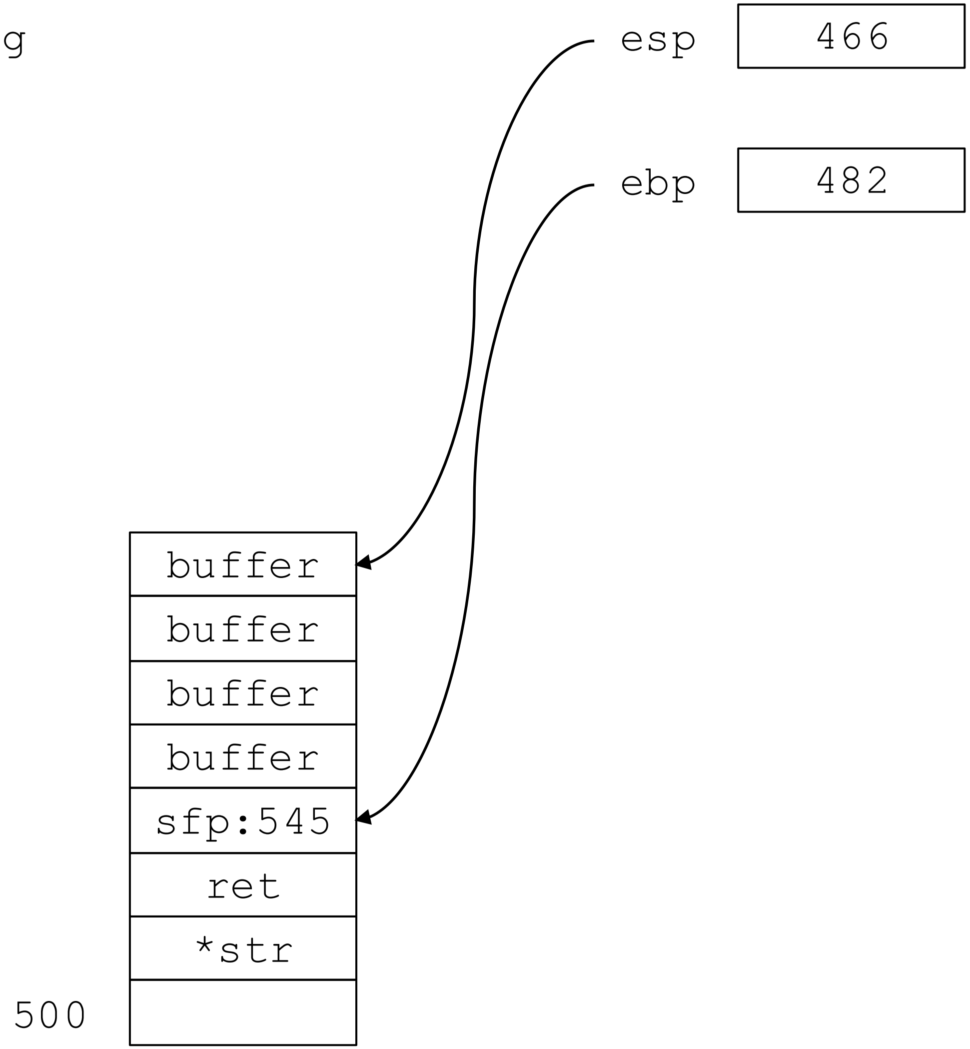
```
pushl $3
pushl $2
pushl $1
call function
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```



Another Example

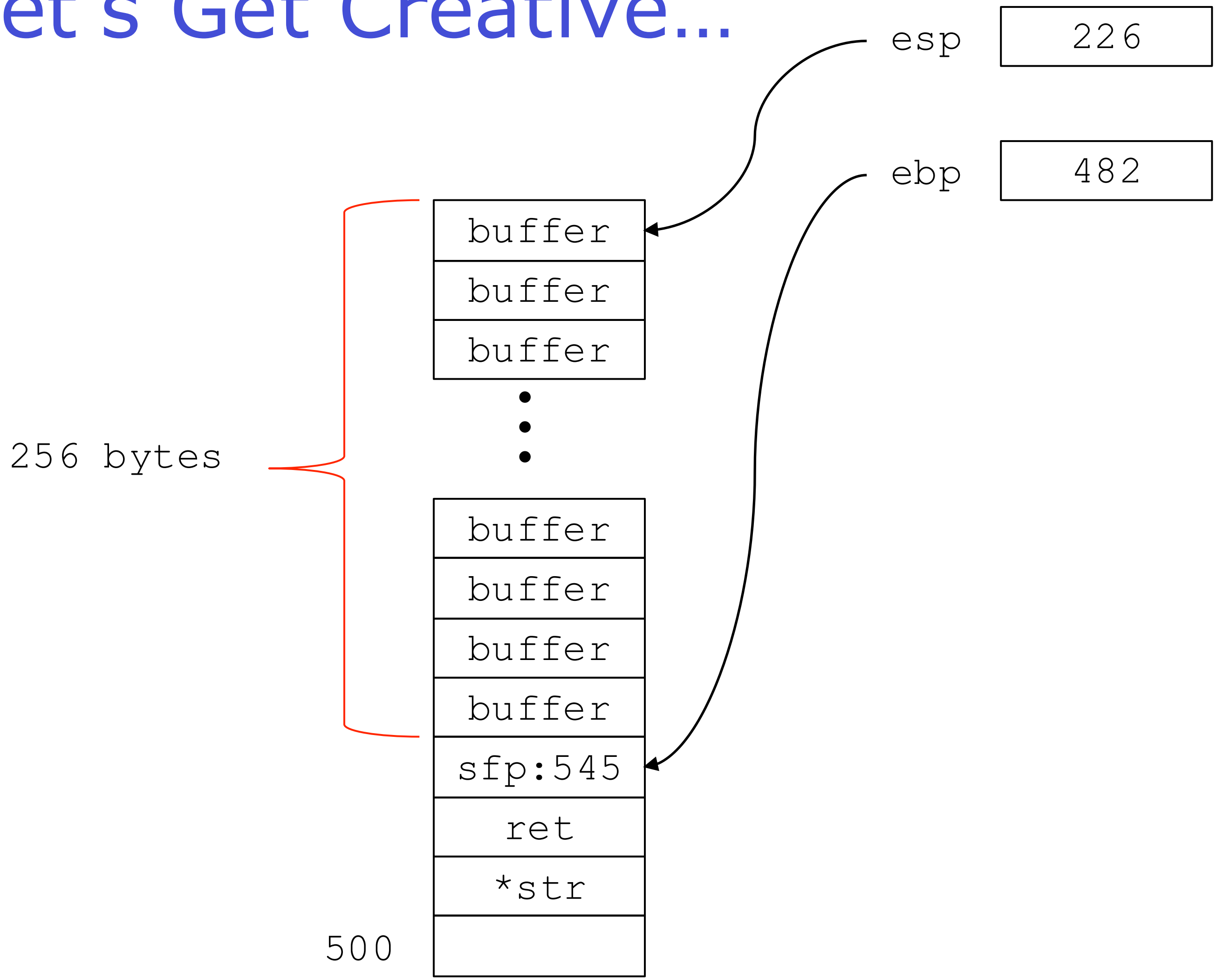
```
example2.c
-------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
-------------------------------
```

Note that code copies a string without using a bounds check (programmer used strcpy() instead of strncpy()). Thus the call to function() causes the buffer to be overwritten, in this case with 0x41414141, the ASCII code for 'A'



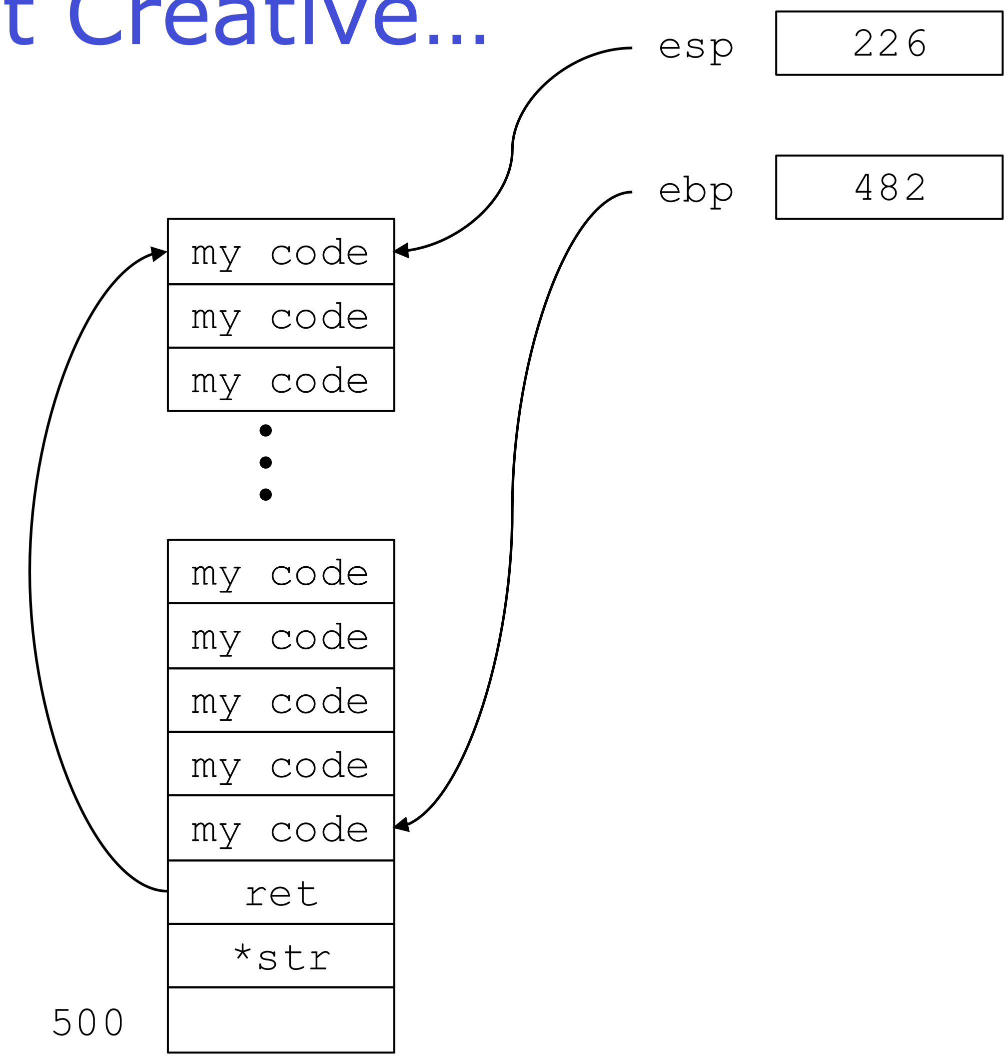
Let's Get Creative…

Let's assume now that buffer is a bit bigger than 20 bytes. Say, e.g., 256 bytes.

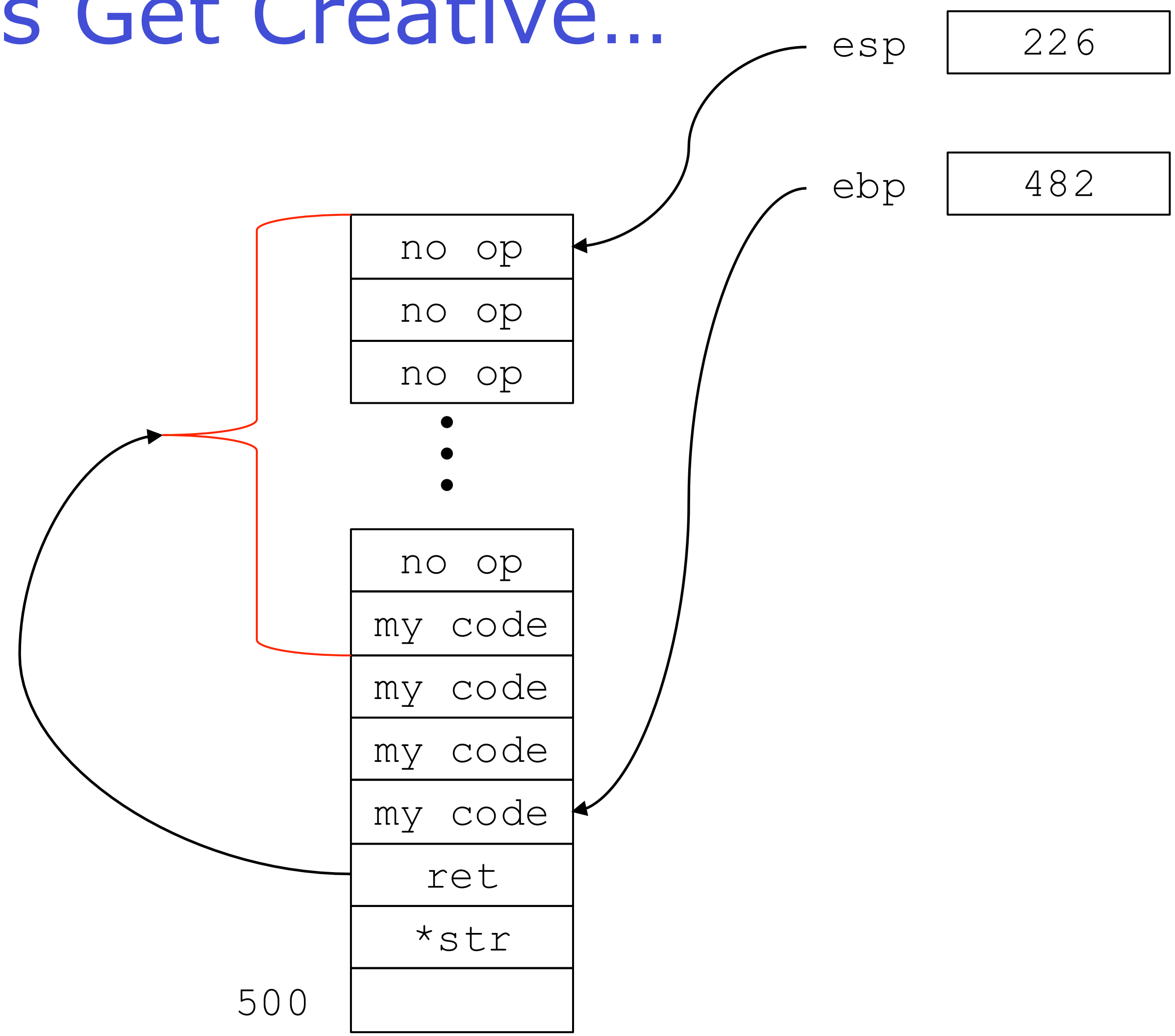


Let’s Get Creative…

Let’s assume now that buffer is a bit bigger than 20 bytes. Say, e.g., 256 bytes. If we know assembly code, we can feed code in as a string, and overwrite the return address to point to this.

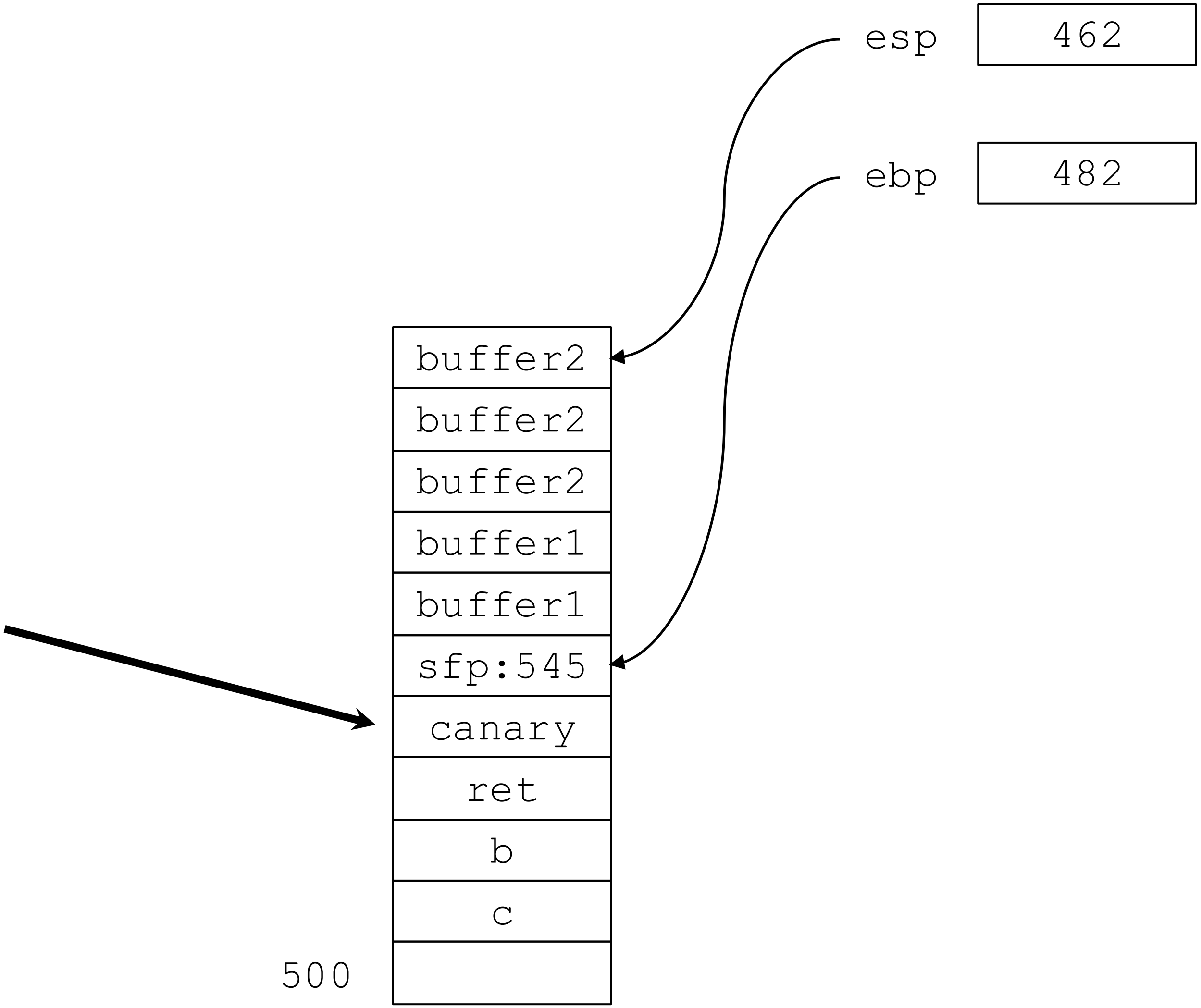


Let’s Get Creative…




```
We don’t even have
to know the exact
address of the start
of the buffer.
```

StackGuard



esp
462
ebp
482
buffer2
buffer2
buffer2
buffer1
buffer1
sfp:545
canary
ret
b
c
500