



CSE 127: Computer Security

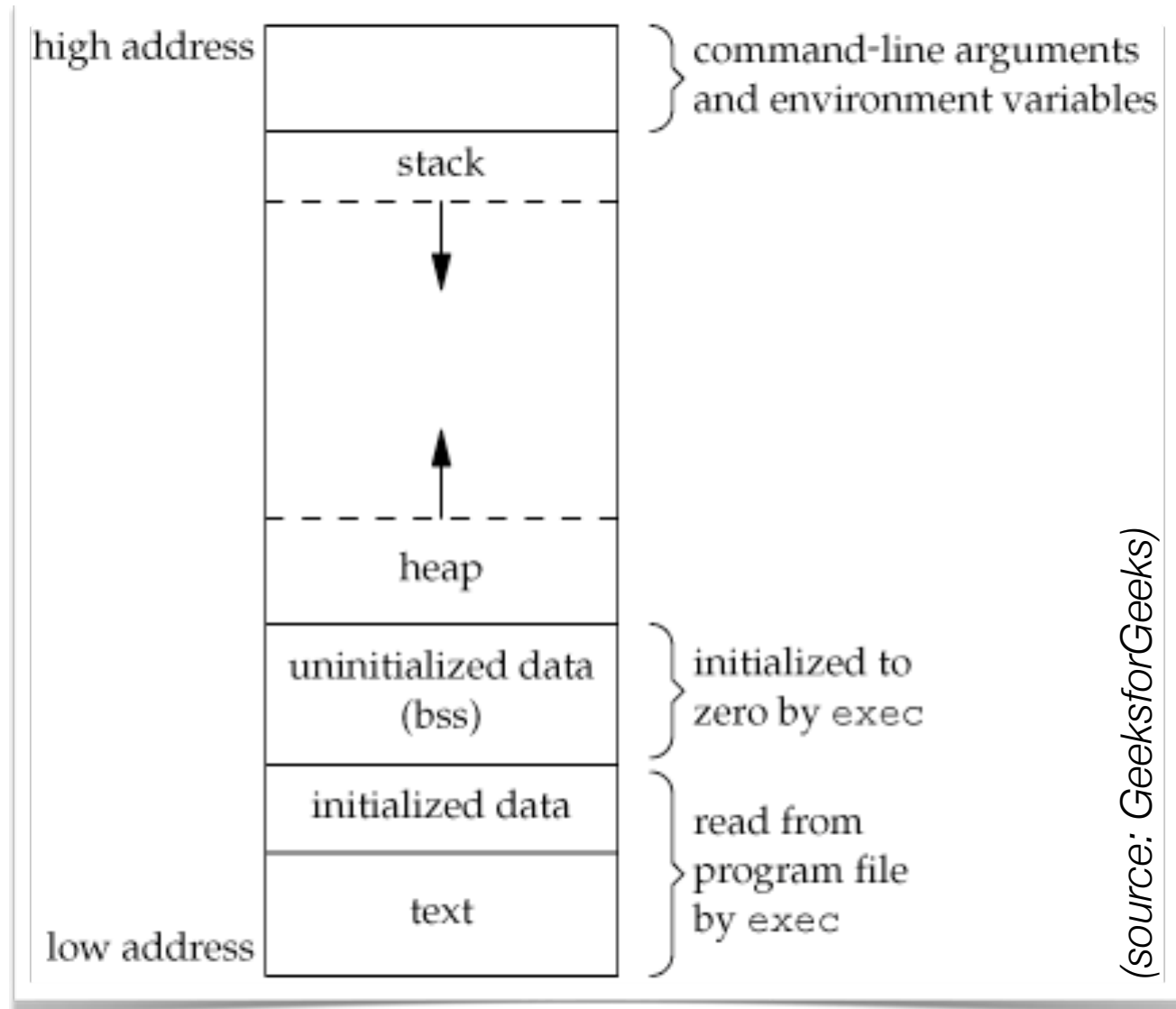
Stack Buffer Overflows

Kirill Levchenko

October 10, 2017

Process Memory Layout

- ❖ Stack
- ❖ Heap
- ❖ Data
 - Static variables
- ❖ Text
 - Executable code



The Stack

- ❖ Function local variables
- ❖ Function arguments
- ❖ Control state

❖ **Stack divided into frames**

- Frame stores function locals and arguments to called functions

❖ **Stack pointer points to top of stack**

- x86: Stack grows down (from high to low addresses)
- x86: Stored in `ESP` register

❖ **Frame pointer points to caller's stack frame**

- Also called **base pointer**
- x86: Stored in `EBP` register

Function Call Example

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```



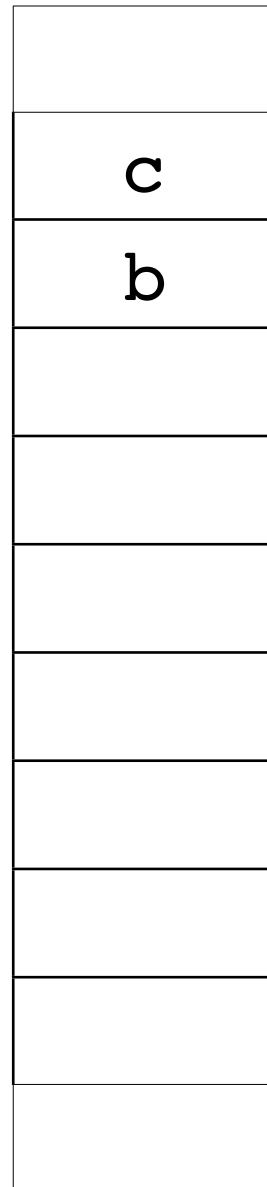
← EBP points to end of main stack frame

← ESP

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```



← EBP points to end of main stack frame

← ESP

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```



← EBP points to end of main stack frame

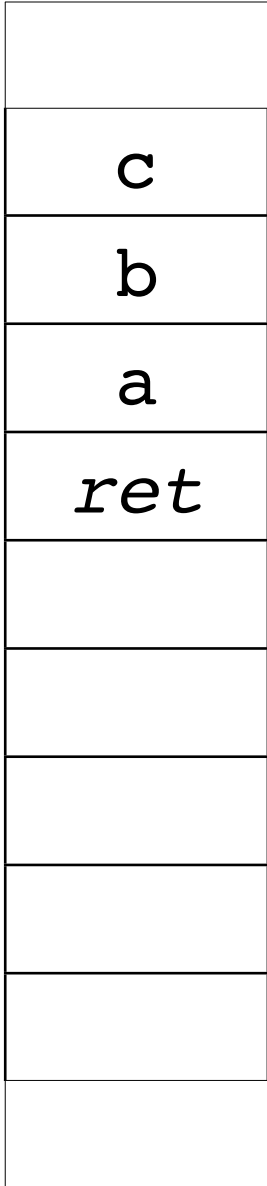
← ESP

← EBP points to end of main stack frame

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```



← ESP

```

_foobar:
; ebp must be preserved across calls. Since
; this function modifies it, it must be
; saved.
;
push    ebp

; From now on, ebp points to the current stack
; frame of the function
;
mov     ebp, esp

; Make space on the stack for local variables
;
sub     esp, 16

; eax <-- a. eax += 2. then store eax in xx
;
mov     eax, DWORD PTR [ebp+8]
add     eax, 2
mov     DWORD PTR [ebp-4], eax

; eax <-- b. eax += 3. then store eax in yy
;
mov     eax, DWORD PTR [ebp+12]
add     eax, 3
mov     DWORD PTR [ebp-8], eax

; eax <-- c. eax += 4. then store eax in zz
;
mov     eax, DWORD PTR [ebp+16]
add     eax, 4
mov     DWORD PTR [ebp-12], eax

```



← EBP points to end of main stack frame

← ESP

```

_foobar:
; ebp must be preserved across calls. Since
; this function modifies it, it must be
; saved.
;
push    ebp

; From now on, ebp points to the current stack
; frame of the function
;
mov     ebp, esp

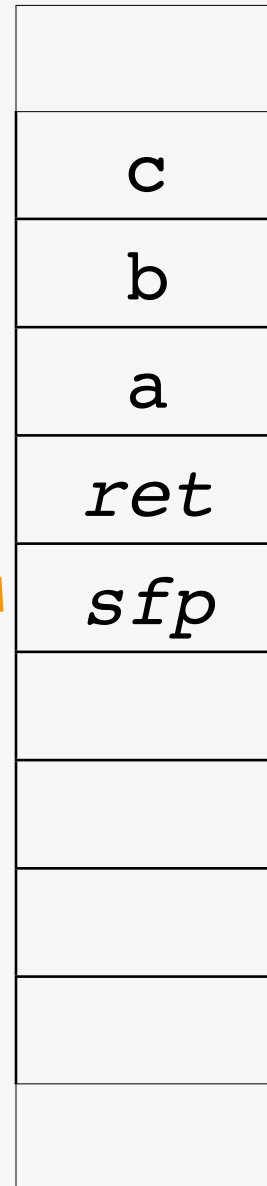
; Make space on the stack for local variables
;
sub     esp, 16

; eax <-- a. eax += 2. then store eax in xx
;
mov     eax, DWORD PTR [ebp+8]
add     eax, 2
mov     DWORD PTR [ebp-4], eax

; eax <-- b. eax += 3. then store eax in yy
;
mov     eax, DWORD PTR [ebp+12]
add     eax, 3
mov     DWORD PTR [ebp-8], eax

; eax <-- c. eax += 4. then store eax in zz
;
mov     eax, DWORD PTR [ebp+16]
add     eax, 4
mov     DWORD PTR [ebp-12], eax

```



```

_foobar:
; ebp must be preserved across calls. Since
; this function modifies it, it must be
; saved.
;
push    ebp

; From now on, ebp points to the current stack
; frame of the function
;
mov     ebp, esp

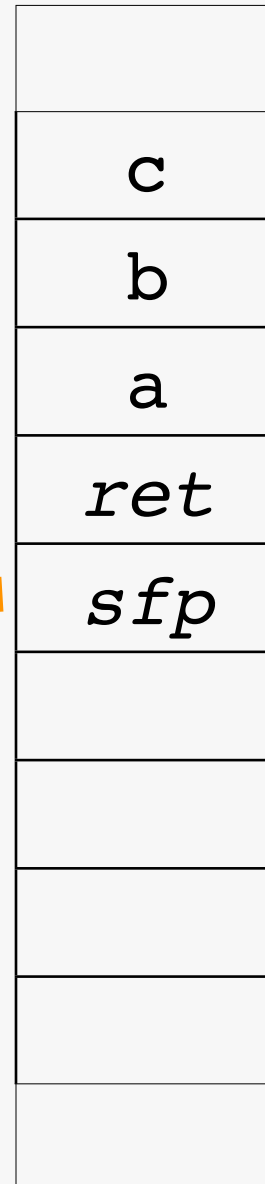
; Make space on the stack for local variables
;
sub     esp, 16

; eax <-- a. eax += 2. then store eax in xx
;
mov     eax, DWORD PTR [ebp+8]
add     eax, 2
mov     DWORD PTR [ebp-4], eax

; eax <-- b. eax += 3. then store eax in yy
;
mov     eax, DWORD PTR [ebp+12]
add     eax, 3
mov     DWORD PTR [ebp-8], eax

; eax <-- c. eax += 4. then store eax in zz
;
mov     eax, DWORD PTR [ebp+16]
add     eax, 4
mov     DWORD PTR [ebp-12], eax

```



← ESP, EBP

```

_foobar:
; ebp must be preserved across calls. Since
; this function modifies it, it must be
; saved.
;
push    ebp

; From now on, ebp points to the current stack
; frame of the function
;
mov     ebp, esp

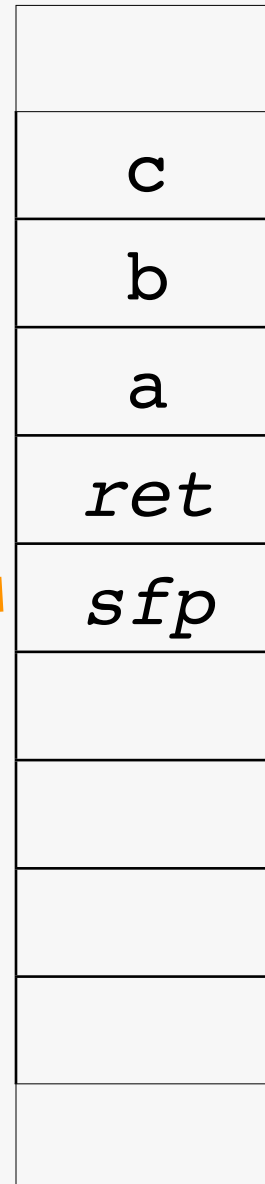
; Make space on the stack for local variables
;
sub     esp, 16

; eax <-- a. eax += 2. then store eax in xx
;
mov     eax, DWORD PTR [ebp+8]
add     eax, 2
mov     DWORD PTR [ebp-4], eax

; eax <-- b. eax += 3. then store eax in yy
;
mov     eax, DWORD PTR [ebp+12]
add     eax, 3
mov     DWORD PTR [ebp-8], eax

; eax <-- c. eax += 4. then store eax in zz
;
mov     eax, DWORD PTR [ebp+16]
add     eax, 4
mov     DWORD PTR [ebp-12], eax

```



```

int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}

```

← EBP

← ESP

```

_foobar:
; ebp must be preserved across calls. Since
; this function modifies it, it must be
; saved.
;
push    ebp

; From now on, ebp points to the current stack
; frame of the function
;
mov     ebp, esp

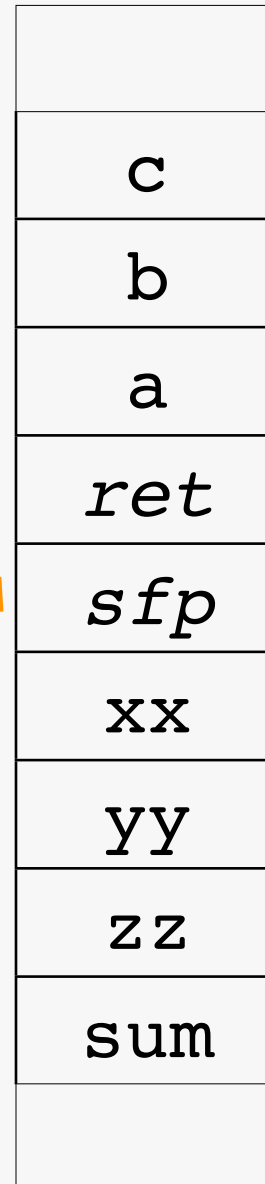
; Make space on the stack for local variables
;
sub     esp, 16

; eax <-- a. eax += 2. then store eax in xx
;
mov     eax, DWORD PTR [ebp+8]
add     eax, 2
mov     DWORD PTR [ebp-4], eax

; eax <-- b. eax += 3. then store eax in yy
;
mov     eax, DWORD PTR [ebp+12]
add     eax, 3
mov     DWORD PTR [ebp-8], eax

; eax <-- c. eax += 4. then store eax in zz
;
mov     eax, DWORD PTR [ebp+16]
add     eax, 4
mov     DWORD PTR [ebp-12], eax

```



```

int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}

```

← EBP

← ESP

```

mov     DWORD PTR [ebp-8], eax

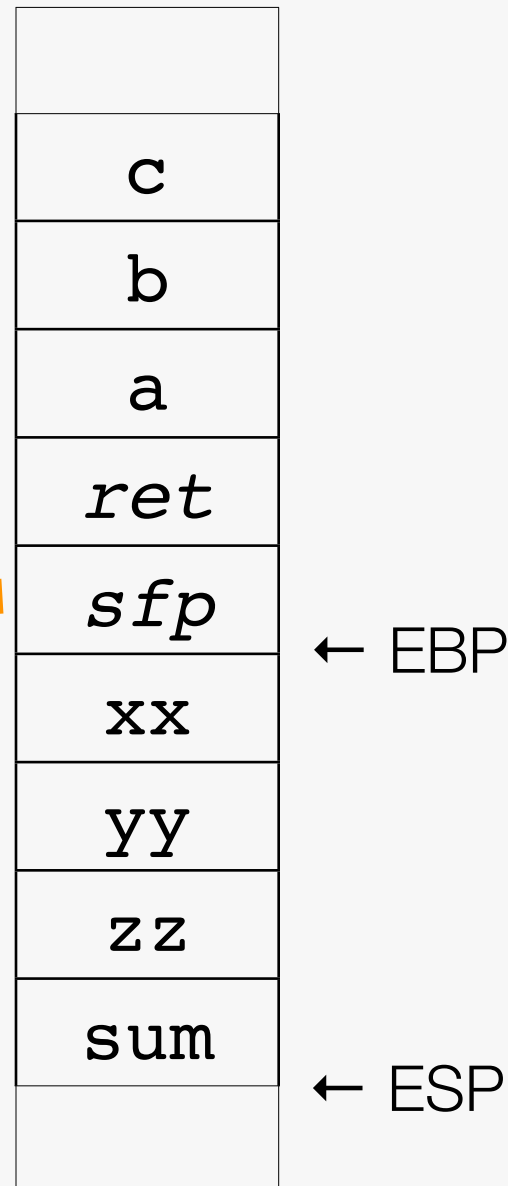
; eax <-- c. eax += 4. then store eax in zz
;
mov     eax, DWORD PTR [ebp+16]
add     eax, 4
mov     DWORD PTR [ebp-12], eax

; add xx + yy + zz and store it in sum
;
mov     eax, DWORD PTR [ebp-8]
mov     edx, DWORD PTR [ebp-4]
lea     eax, [edx+eax]
add     eax, DWORD PTR [ebp-12]
mov     DWORD PTR [ebp-16], eax

; Compute final result into eax, which
; stays there until return
;
mov     eax, DWORD PTR [ebp-4]
imul   eax, DWORD PTR [ebp-8]
imul   eax, DWORD PTR [ebp-12]
add     eax, DWORD PTR [ebp-16]

; The leave instruction here is equivalent to:
;
;   mov esp, ebp
;   pop ebp
;
; Which cleans the allocated locals and restores
; ebp.
;
leave
ret

```



```

mov     DWORD PTR [ebp-8], eax

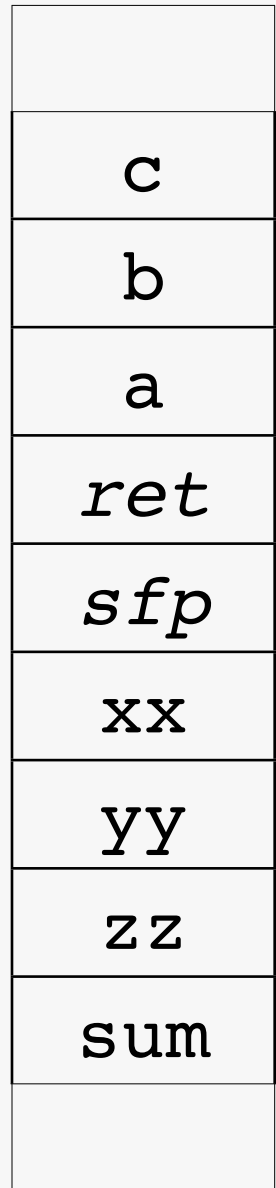
; eax <-- c. eax += 4. then store eax in zz
;
mov     eax, DWORD PTR [ebp+16]
add     eax, 4
mov     DWORD PTR [ebp-12], eax

; add xx + yy + zz and store it in sum
;
mov     eax, DWORD PTR [ebp-8]
mov     edx, DWORD PTR [ebp-4]
lea     eax, [edx+eax]
add     eax, DWORD PTR [ebp-12]
mov     DWORD PTR [ebp-16], eax

; Compute final result into eax, which
; stays there until return
;
mov     eax, DWORD PTR [ebp-4]
imul   eax, DWORD PTR [ebp-8]
imul   eax, DWORD PTR [ebp-12]
add     eax, DWORD PTR [ebp-16]

; The leave instruction here is equivalent to:
;
;   mov esp, ebp
;   pop ebp
;
; Which cleans the allocated locals and restores
; ebp.
;
leave
ret

```



← EBP, ESP

```

mov     DWORD PTR [ebp-8], eax

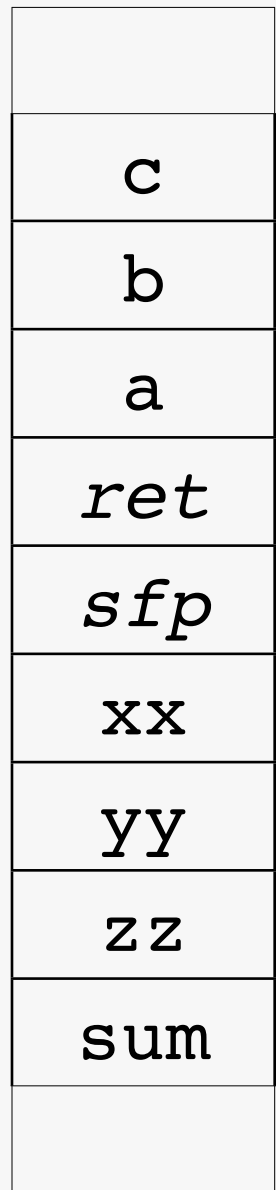
; eax <-- c. eax += 4. then store eax in zz
;
mov     eax, DWORD PTR [ebp+16]
add     eax, 4
mov     DWORD PTR [ebp-12], eax

; add xx + yy + zz and store it in sum
;
mov     eax, DWORD PTR [ebp-8]
mov     edx, DWORD PTR [ebp-4]
lea     eax, [edx+eax]
add     eax, DWORD PTR [ebp-12]
mov     DWORD PTR [ebp-16], eax

; Compute final result into eax, which
; stays there until return
;
mov     eax, DWORD PTR [ebp-4]
imul   eax, DWORD PTR [ebp-8]
imul   eax, DWORD PTR [ebp-12]
add     eax, DWORD PTR [ebp-16]

; The leave instruction here is equivalent to:
;
;   mov esp, ebp
;   pop ebp
;
; Which cleans the allocated locals and restores
; ebp.
;
leave
ret

```



```

mov     DWORD PTR [ebp-8], eax

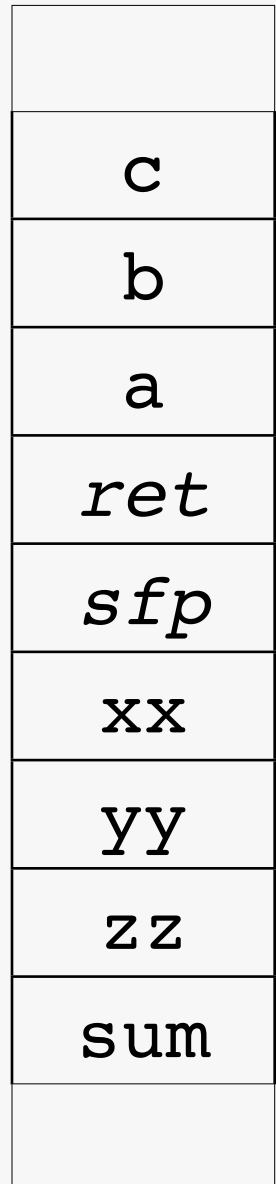
; eax <-- c. eax += 4. then store eax in zz
;
mov     eax, DWORD PTR [ebp+16]
add     eax, 4
mov     DWORD PTR [ebp-12], eax

; add xx + yy + zz and store it in sum
;
mov     eax, DWORD PTR [ebp-8]
mov     edx, DWORD PTR [ebp-4]
lea     eax, [edx+eax]
add     eax, DWORD PTR [ebp-12]
mov     DWORD PTR [ebp-16], eax

; Compute final result into eax, which
; stays there until return
;
mov     eax, DWORD PTR [ebp-4]
imul   eax, DWORD PTR [ebp-8]
imul   eax, DWORD PTR [ebp-12]
add     eax, DWORD PTR [ebp-16]

; The leave instruction here is equivalent to:
;
;   mov esp, ebp
;   pop ebp
;
; Which cleans the allocated locals and restores
; ebp.
;
leave
ret

```



← EBP

← ESP

Control transferred to ret!

With Buffers

example1.c:

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

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

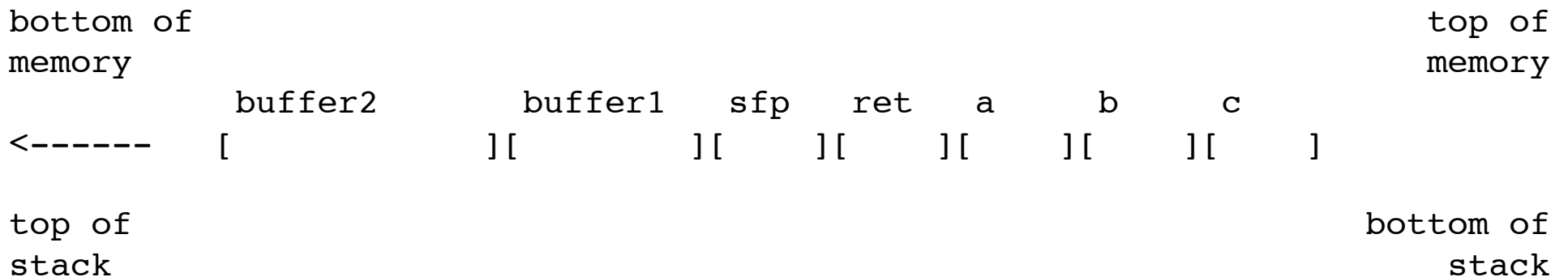
```
-----
```

With Buffers

example1.c:

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

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



With Buffers

```
buffer          sfp   ret   *str  
[               ][     ][     ][     ]
```

- ❖ `strcpy` will copy memory from `str` to `buffer` until a null character `'\0'`
- ❖ If length of string longer than `buffer`, `strcpy` will copy string over `sfp` and `ret`

```
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);  
}  
-----
```

Stack Buffer Overflow

- ❖ If source string of `strcpy` controlled by attacker (and destination is on the stack) — attacker gets to control where the function returns by overwriting `ret`
- ❖ Attacker gets to transfer control to anywhere!
- ❖ Where to jump?

Shellcode

- ❖ **Shellcode:** small code fragment that receives initial control in an control flow hijack exploit
 - **Control flow hijack:** Taking control of CPU instruction pointer
- ❖ Earliest attacks used shellcode to `exec` a shell
 - Target a `setuid` root program, gives you a shell as root

Shellcode

- ❖ We want something that does this

```
void main() {  
    char *name[2];  
  
    name[0] = "/bin/sh";  
    name[1] = NULL;  
    execve(name[0], name, NULL);  
}
```

Shellcode

❖ There are some constraints

- Shellcode cannot contain null characters '\0' which end string
- Need:

- a) Have the null terminated string `"/bin/sh"` somewhere in memory.
- b) Have the address of the string `"/bin/sh"` somewhere in memory followed by a null long word.
- c) Copy `0xb` into the EAX register.
- d) Copy the address of the address of the string `"/bin/sh"` into the EBX register.
- e) Copy the address of the string `"/bin/sh"` into the ECX register.
- f) Copy the address of the null long word into the EDX register.
- g) Execute the `int $0x80` instruction.

└ system call

Shellcode

bottom of memory	DDDDDDDDDEEEEEEEEEEEEEEE	EEEE	FFFF	FFFF	FFFF	FFFF	top of memory
	89ABCDEF0123456789AB	CDEF	0123	4567	89AB	CDEF	
	buffer	sfp	ret	a	b	c	

