Accessing Parameters

Working the stack!

© Government of Canada
This document is the property of the Government of Canada. It shall not be altered, distributed beyond its intended audience, produced, reproduced or published, in whole or in any substantial part thereof, without the express permission of CSE.

Quick reminder about C

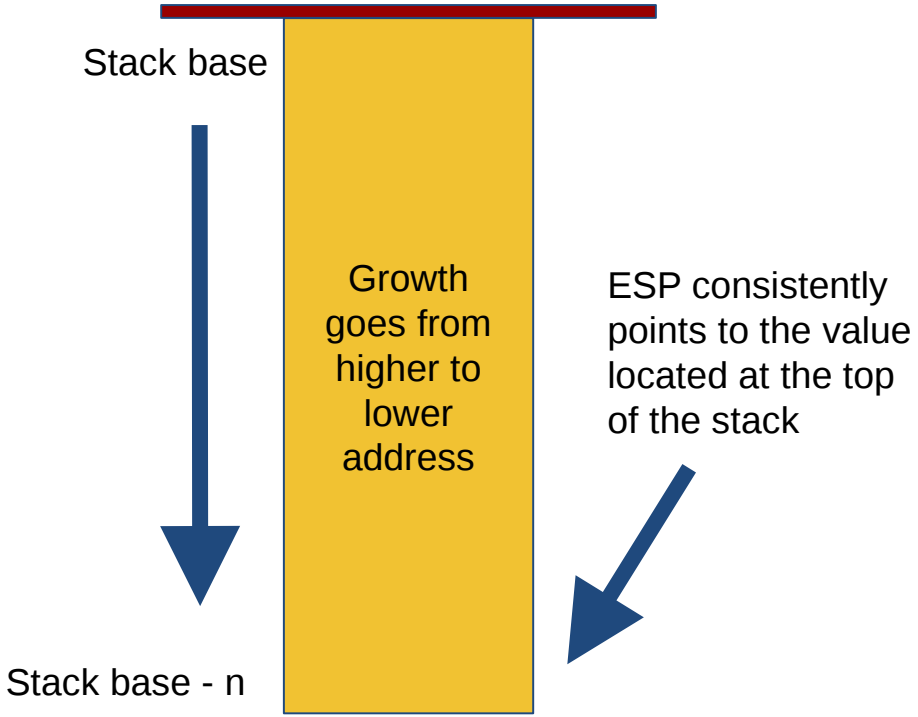
```c
#include <stdio.h>


int main(int argc, char* argv[]){
        printf("%d arguments\n", argc);
        printf("%s\n", argv[0]);

        if (argc > 1){
                printf("%s\n", argv[1]);
        }
}
```

main receives 2 parameters

- **argc** is the **number of** command line **parameters**
- **argv** is an array of char* (strings) each containing **separate** command line **arguments**

On IA-32, parameters are passed on the stack in reverse order of appearance in a function declaration.

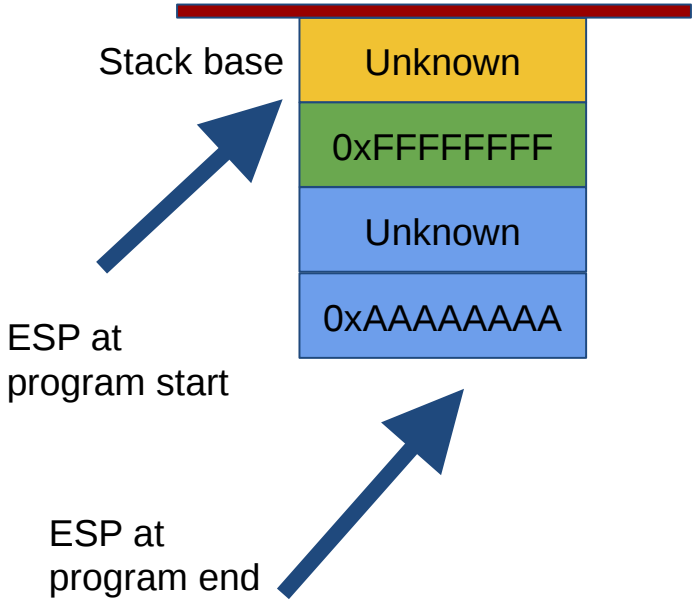
The stack



Multiple instructions can change the stack pointer location and add or remove elements from the stack among the most popular:

- *push value*
- *pop register*
- *call label*
- *ret*
- *add* destination, source (addition)
- *sub* destination, source (subtraction)
- etc...

Example



The following instructions changes the stack to the presented state:

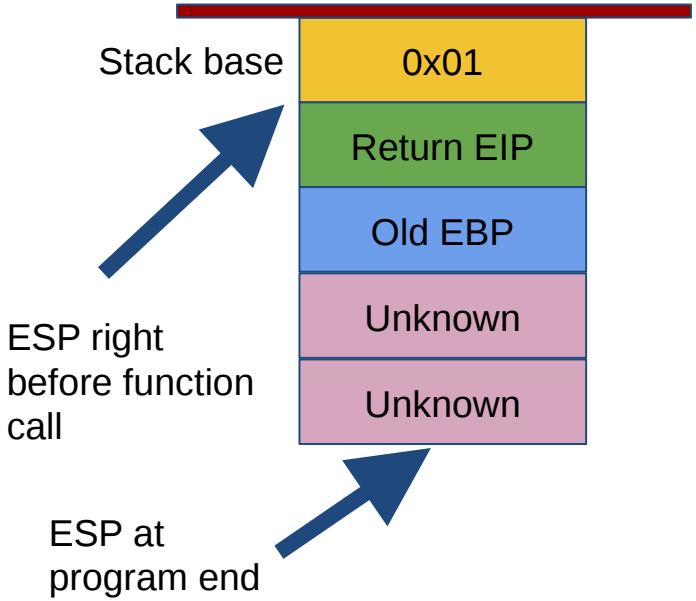
- *push 0xFFFFFFFF*
- *sub esp, 0x08*
- *mov [esp], 0xAAAAAAAA*

What would be the result of instruction:

- *add esp, 0xC*

What are the consequence of using that instruction?

More complicated example

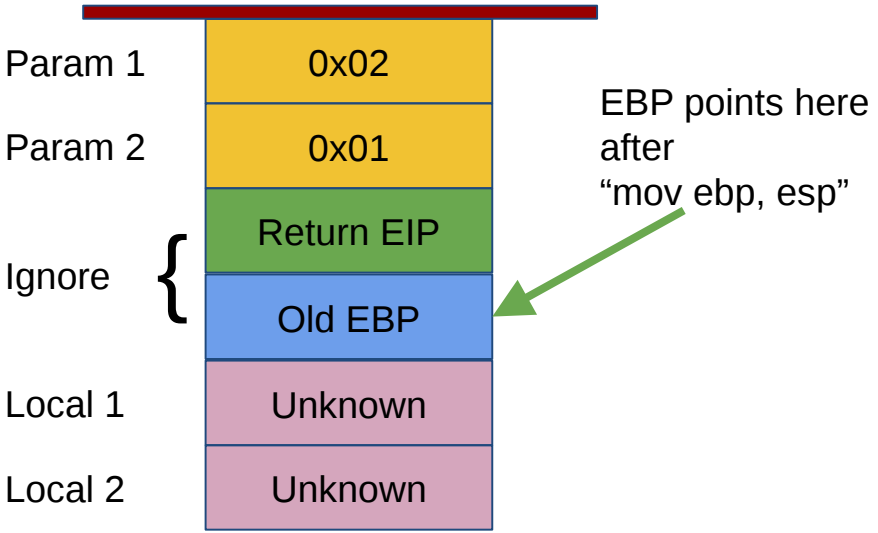


- *push 0x01*
- *call randomFunction*
- **** Code for randomFunction
- *push ebp*
- *mov ebp, esp*
- *sub esp, 0x08*

For now, ignore the return EIP and the old EBP parts. We will be coming back to this later.

We will call everything from the red line to the esp after the sub a “Stack frame”

Stack Frame



- *push 0x02*
- *push 0x01*
- *call randomFunction*
- **** Code for randomFunction
- *push ebp*
- *mov ebp, esp*
- *sub esp, 0x08*

For now, ignore the return EIP and the old EBP parts. We will be coming back to this later.

Param 1 can be accessed as [ebp + 0xC]
Param 2 can be accessed as [ebp + 0x8]
Why?

We will call everything from the red line to the esp after the sub a "Stack frame"

The compiler monster!

Different compilers, different flavors of code!

```
(gdb) disass $eip
Dump of assembler code for function main:
   0x08048410 <+0>:     push   ebp
   0x08048411 <+1>:     mov    ebp,esp
   0x08048413 <+3>:     sub    esp,0x18
=> 0x08048416 <+6>:     mov    eax,DWORD PTR [ebp+0xc]
   0x08048419 <+9>:     mov    ecx,DWORD PTR [ebp+0x8]
   0x0804841c <+12>:    mov    DWORD PTR [ebp-0x4],0x0
   0x08048423 <+19>:    mov    DWORD PTR [ebp-0x8],ecx
   0x08048426 <+22>:    mov    DWORD PTR [ebp-0xc],eax
   0x08048429 <+25>:    mov    DWORD PTR [ebp-0x10],0x0
   0x08048430 <+32>:    mov    DWORD PTR [ebp-0x14],0x0
   0x08048437 <+39>:    mov    eax,DWORD PTR [ebp-0xc]
   0x0804843a <+42>:    mov    eax,DWORD PTR [eax]
   0x0804843c <+44>:    mov    ecx,esp
   0x0804843e <+46>:    mov    DWORD PTR [ecx],eax
   0x08048440 <+48>:    call   0x80482e0 <strlen@plt>
   0x08048445 <+53>:    mov    DWORD PTR [ebp-0x10],eax
   0x08048448 <+56>:    cmp    DWORD PTR [ebp-0x8],0x1
   0x0804844c <+60>:    jle    0x8048464 <main+84>
   0x08048452 <+66>:    mov    eax,DWORD PTR [ebp-0xc]
   0x08048455 <+69>:    mov    eax,DWORD PTR [eax+0x4]
   0x08048458 <+72>:    mov    ecx,esp
   0x0804845a <+74>:    mov    DWORD PTR [ecx],eax
   0x0804845c <+76>:    call   0x80482e0 <strlen@plt>
   0x08048461 <+81>:    mov    DWORD PTR [ebp-0x14],eax
   0x08048464 <+84>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048467 <+87>:    add    esp,0x18
   0x0804846a <+90>:    pop    ebp
   0x0804846b <+91>:    ret
End of assembler dump.
```

clang is presented **at the left** while **gcc** is presented **at the right.** The compiler has a direct impact on the code being generated!
Visual C++ is closer to clang. The important thing is the end code. All of these are relatively equivalent.

```
Dump of assembler code for function main:
   0x0804840b <+0>:     lea    ecx,[esp+0x4]
   0x0804840f <+4>:     and    esp,0xfffffff0
   0x08048412 <+7>:     push   DWORD PTR [ecx-0x4]
   0x08048415 <+10>:    push   ebp
   0x08048416 <+11>:    mov    ebp,esp
   0x08048418 <+13>:    push   ebx
   0x08048419 <+14>:    push   ecx
=> 0x0804841a <+15>:    sub    esp,0x10
   0x0804841d <+18>:    mov    ebx,ecx
   0x0804841f <+20>:    mov    DWORD PTR [ebp-0x10],0x0
   0x08048426 <+27>:    mov    DWORD PTR [ebp-0xc],0x0
   0x0804842d <+34>:    mov    eax,DWORD PTR [ebx+0x4]
   0x08048430 <+37>:    mov    eax,DWORD PTR [eax]
   0x08048432 <+39>:    sub    esp,0xc
   0x08048435 <+42>:    push   eax
   0x08048436 <+43>:    call   0x80482e0 <strlen@plt>
   0x0804843b <+48>:    add    esp,0x10
   0x0804843e <+51>:    mov    DWORD PTR [ebp-0x10],eax
   0x08048441 <+54>:    cmp    DWORD PTR [ebx],0x1
   0x08048444 <+57>:    jle    0x804845d <main+82>
   0x08048446 <+59>:    mov    eax,DWORD PTR [ebx+0x4]
   0x08048449 <+62>:    add    eax,0x4
   0x0804844c <+65>:    mov    eax,DWORD PTR [eax]
   0x0804844e <+67>:    sub    esp,0xc
   0x08048451 <+70>:    push   eax
   0x08048452 <+71>:    call   0x80482e0 <strlen@plt>
   0x08048457 <+76>:    add    esp,0x10
   0x0804845a <+79>:    mov    DWORD PTR [ebp-0xc],eax
   0x0804845d <+82>:    mov    eax,DWORD PTR [ebx]
   0x0804845f <+84>:    lea    esp,[ebp-0x8]
   0x08048462 <+87>:    pop    ecx
   0x08048463 <+88>:    pop    ebx
   0x08048464 <+89>:    pop    ebp
   0x08048465 <+90>:    lea    esp,[ecx-0x4]
   0x08048468 <+93>:    ret
```

Different level of optimization also change the code

```
(gdb) disass main
Dump of assembler code for function main:
   0x080483db <+0>:     mov    eax,DWORD PTR [esp+0x4]
   0x080483df <+4>:     ret
End of assembler dump.
```

GCC used with -O3 optimization

Why was all the code removed?

Understanding stack frames is central to assembly programming and software reverse engineering.

Review

- Let’s say a function has 4 parameters, how can we access parameter #3?
- And if we wanted to access parameter #4?
- And the first parameter?
- EBP is normally used to access both local variables and function parameters. How can we quickly know if an access (relative to EBP) is done to function parameters or to the local variables?
- How about writing some code…