Buffer Overflow Attacks

David Oswald and Eike Ritter

Introduction into Computer Security,

Based on a course by Tom Chothia

Introduction

- A simplified, high-level view of buffer overflow attacks
  - x86 architecture
  - overflows on the stack
  - Focus on 32-bit mode, but most things directly apply to 64-bit mode as well

- Exploiting buffer overflows using Metasploit

Introduction

- In languages like C, you have to tell the compiler how to manage the memory.
  - This is hard.
- If you get it wrong, then an attacker can usually exploit this bug to make your application run *arbitrary code.*
- Countless worms, attacks against SQL servers, web servers, iPhone jailbreaks, SSH servers, …

USS Yorktown

US Navy Aegis missile cruiser

Dead in the water for 2 and a half hours due to a buffer overflow.

USS Yorktown

US Navy Aegis missile cruiser

Dead in the water for 2 and a half hours due to a buffer overflow.

“Because of politics, some things are being forced on us that without political pressure we might not do, …

Ron Redman, deputy technical director Aegis

USS Yorktown

US Navy Aegis missile cruiser

Dead in the water for 2 and a half hours due to a buffer overflow.

"Because of politics, some things are being forced on us that without political pressure we might not do, like Windows NT. If it were up to me I probably would not have used Windows NT in this particular application."

Ron Redman, deputy technical director Aegis

What‘s wrong with this code?

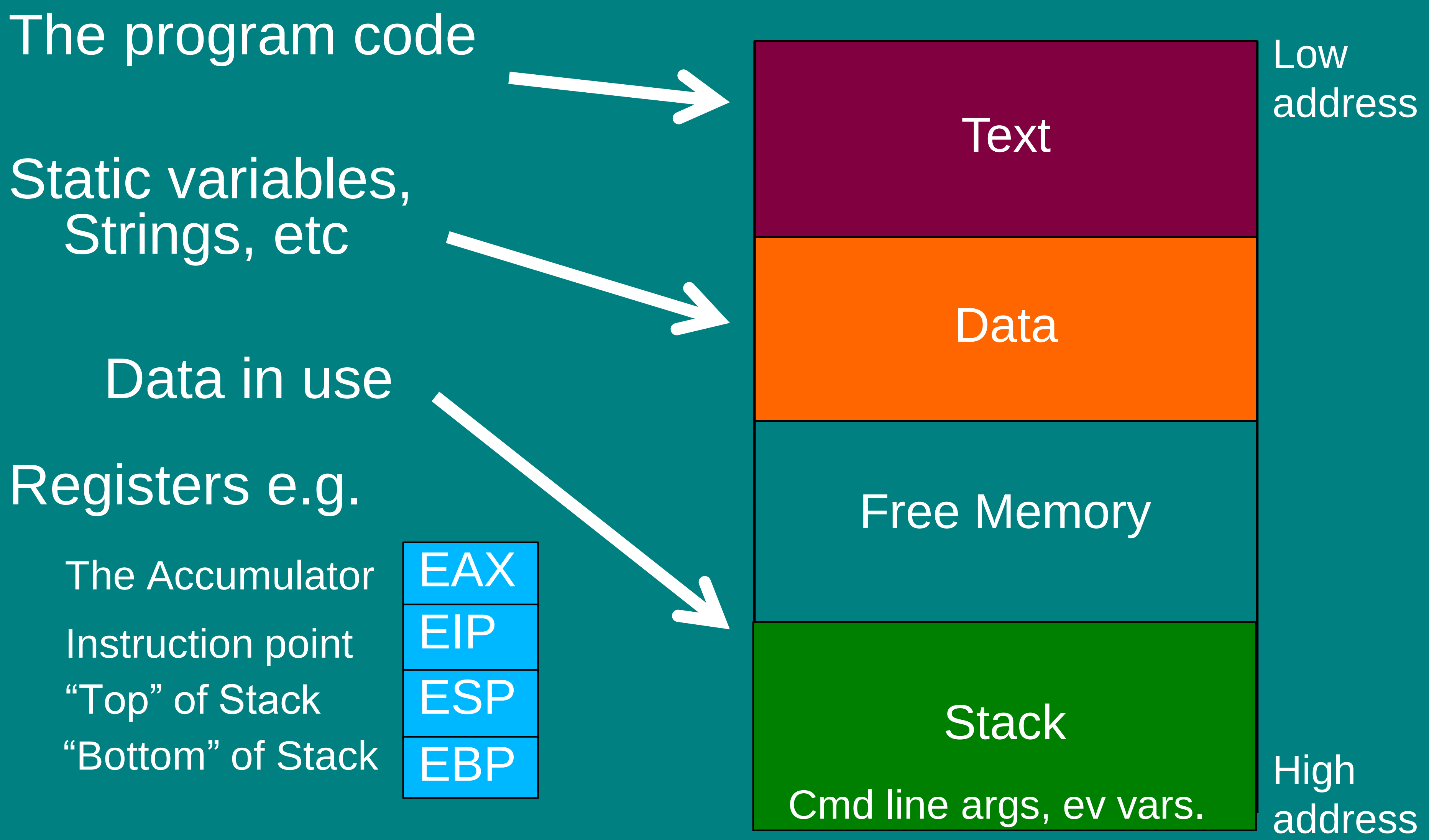
```
void getname() {
    char buffer[16];
    gets(buffer);
    printf("Your name is %s.\n", buffer);
}

int main(void) {
    printf("Enter your name:" );
    getname();
    return 0;
}
```

Live-Demo

"Anything that can go wrong, will go wrong"

Triggering a buffer overflow

The x86 Architecture



The x86 Architecture

The program code

Static variables,
Strings, etc

Data in use

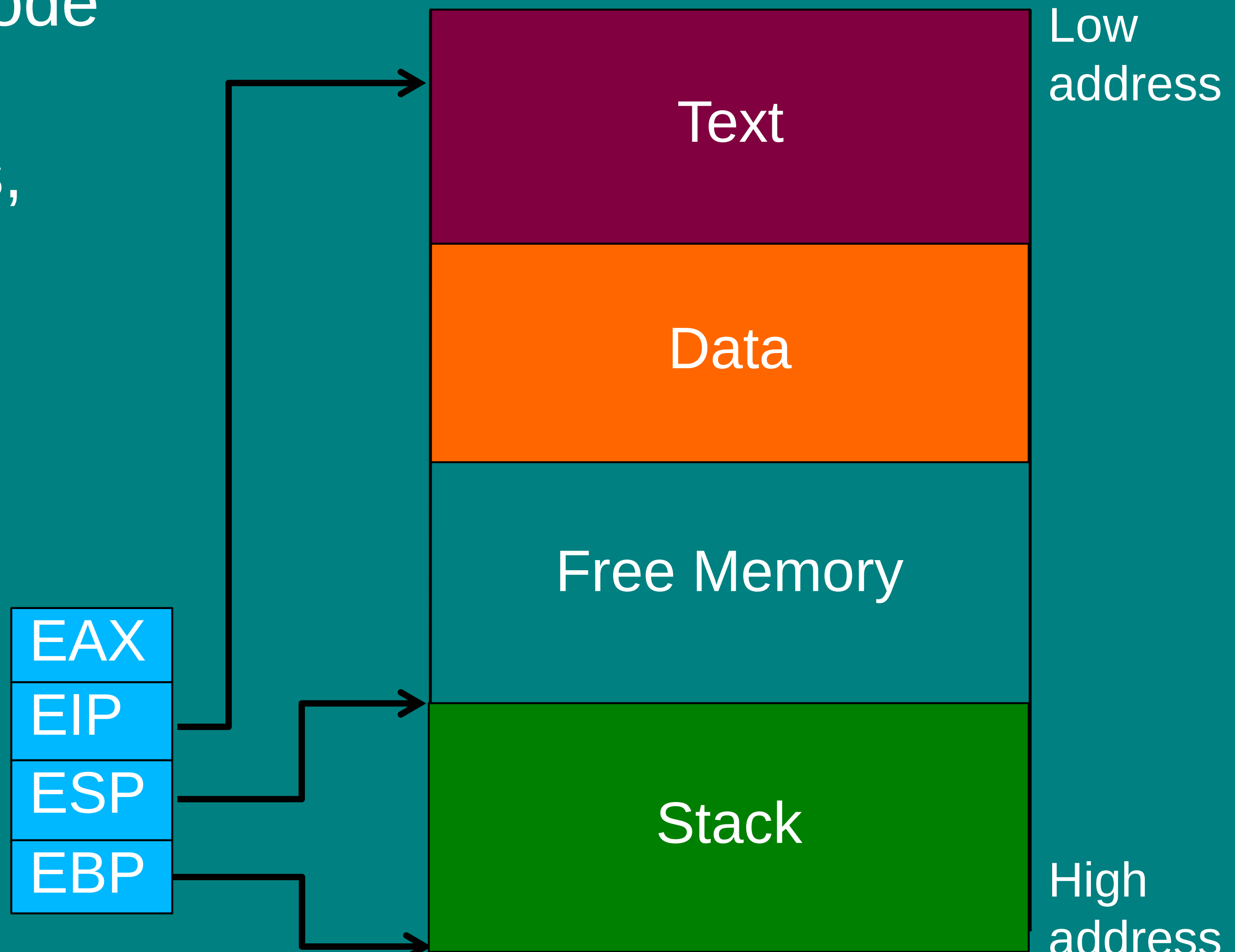
Registers e.g.

The Accumulator

Instruction point

“Top” of Stack

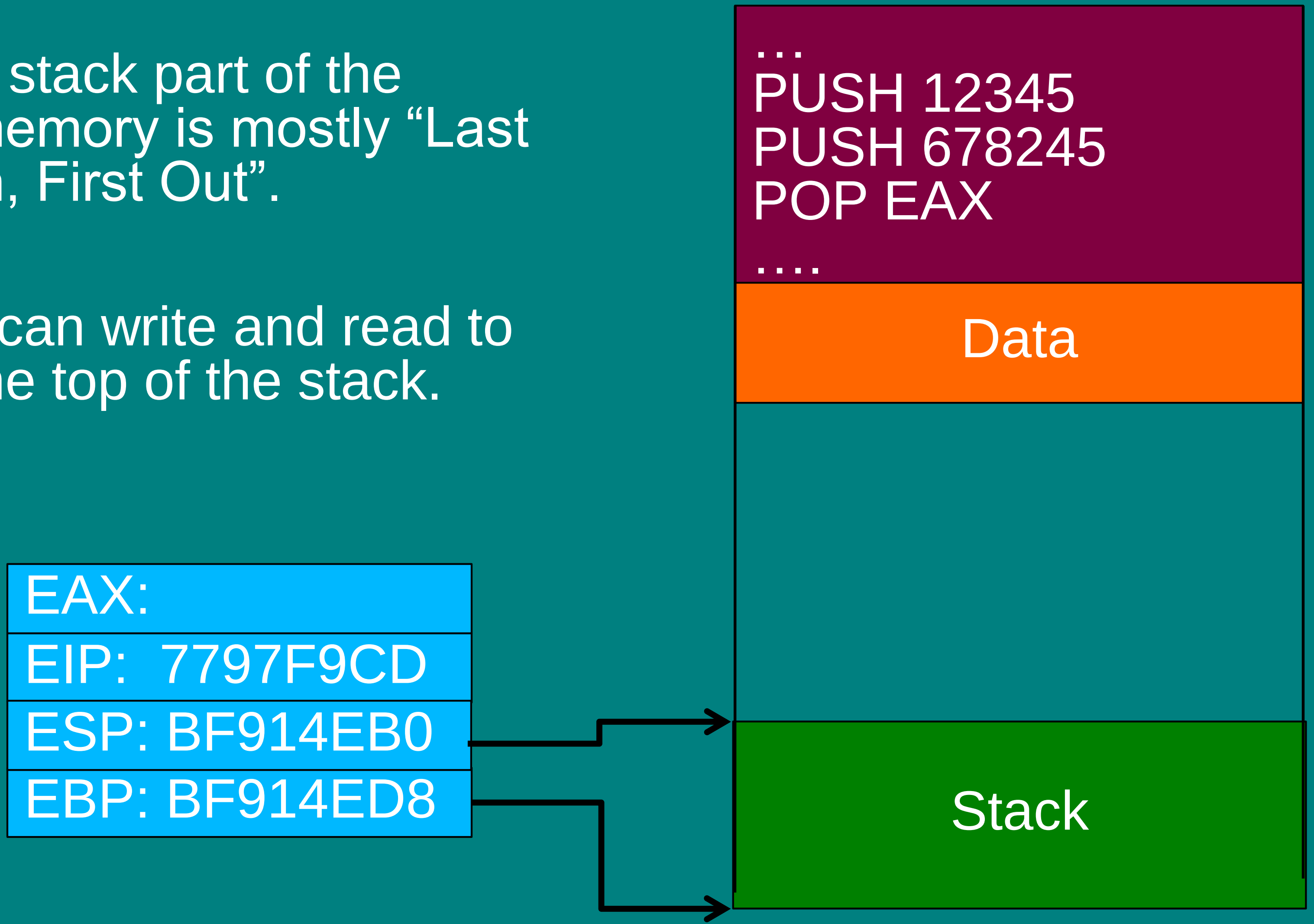
“Bottom” of Stack



The Stack

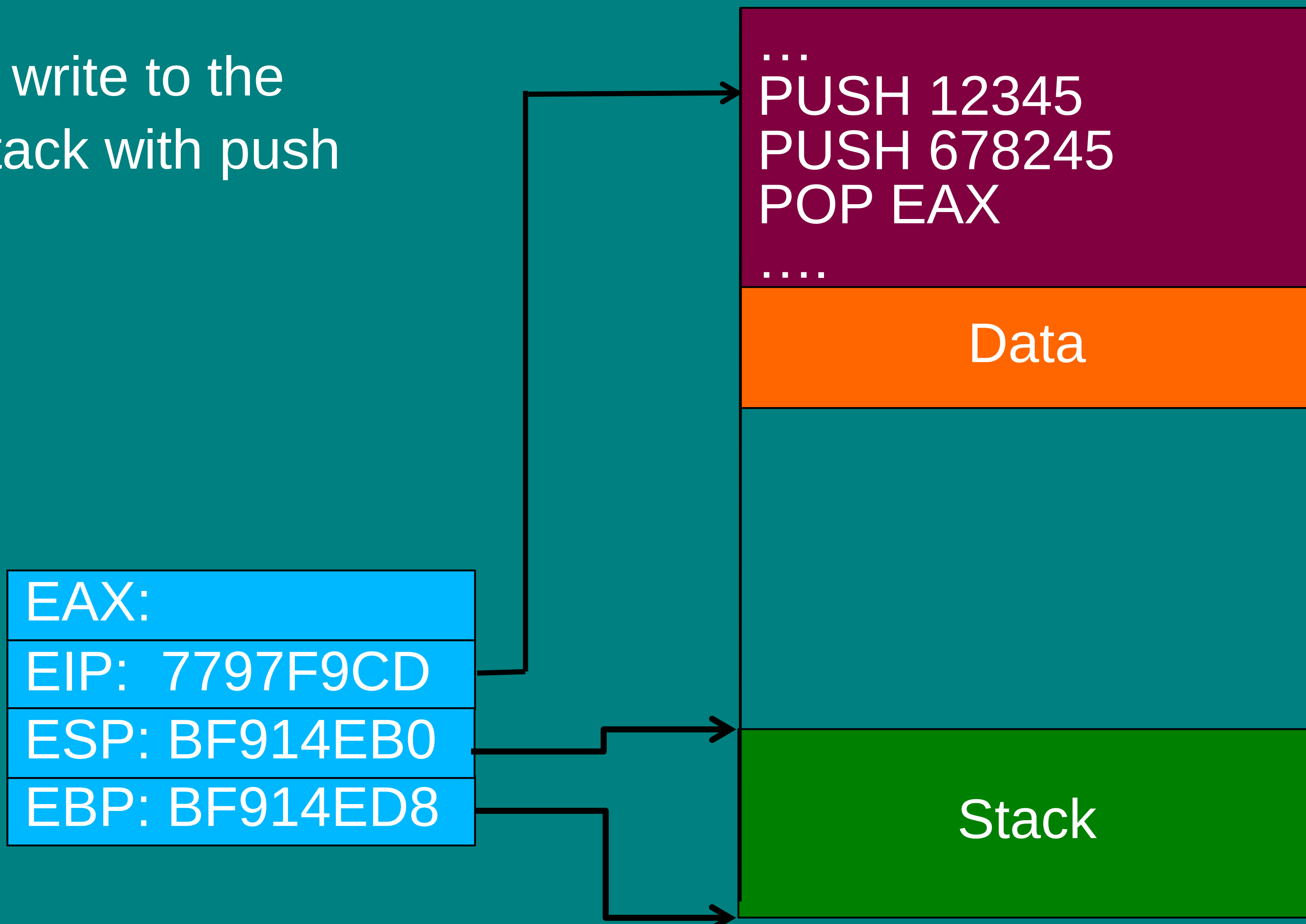
The stack part of the memory is mostly "Last In, First Out".

We can write and read to the top of the stack.

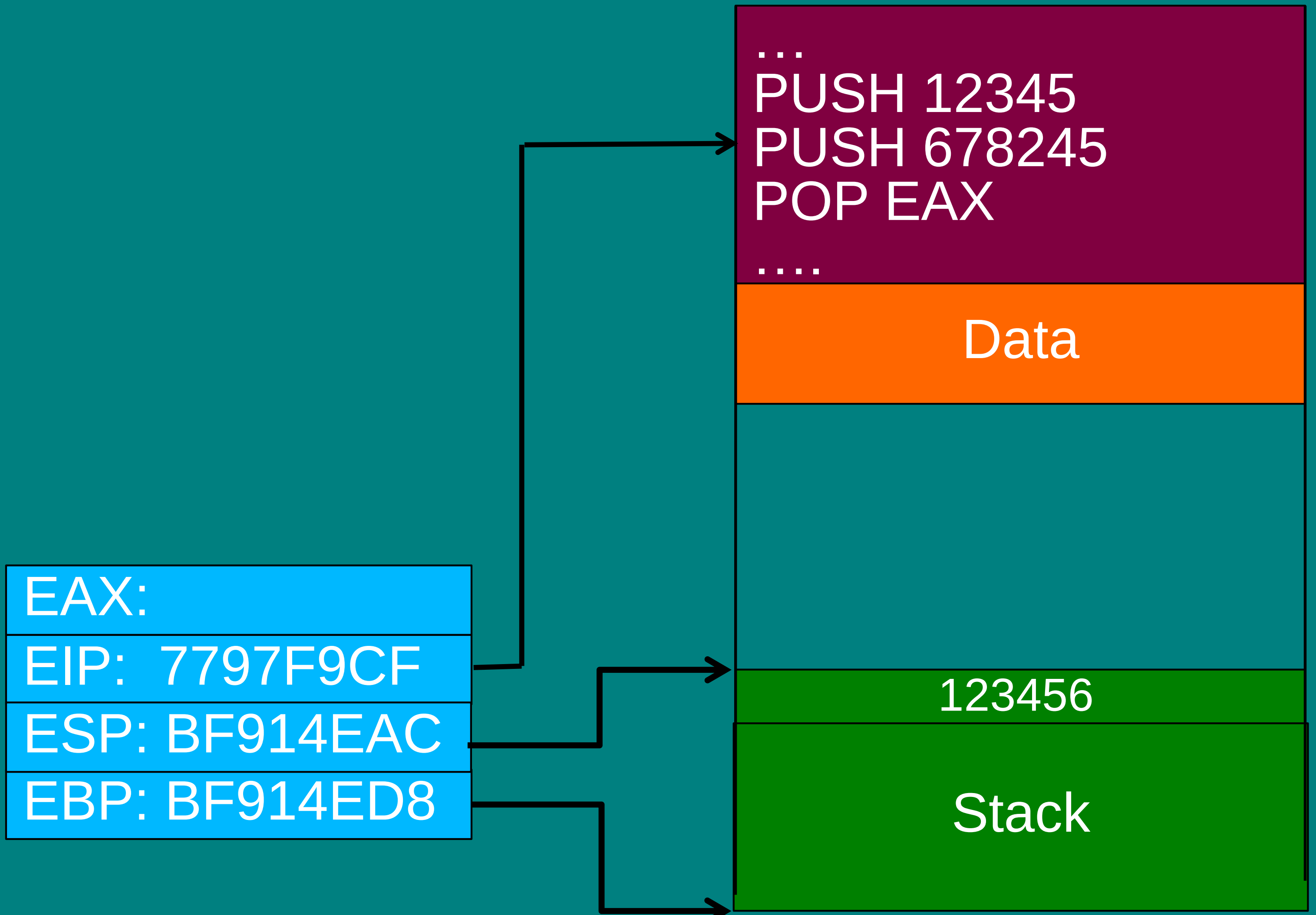


The Stack

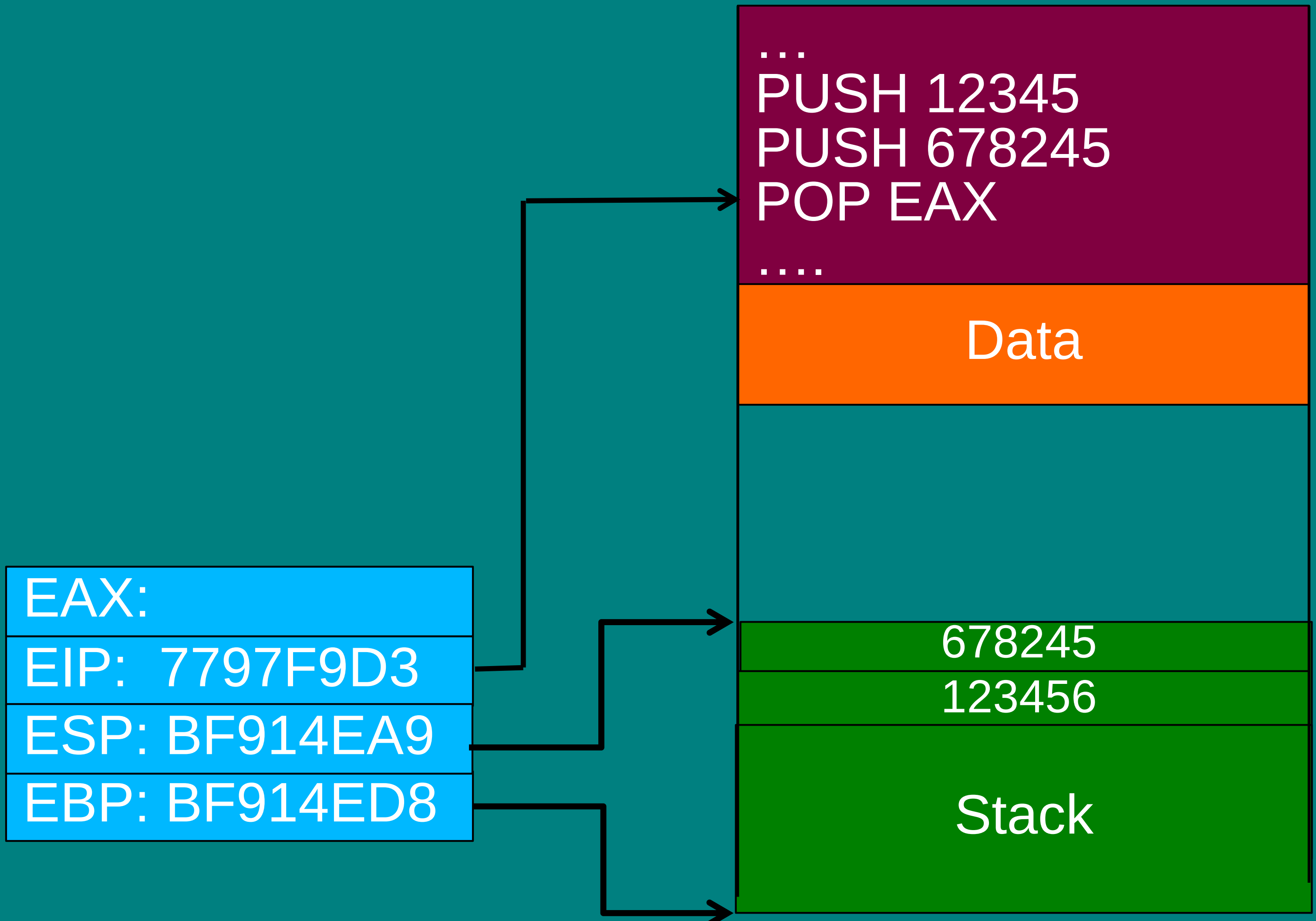
You write to the stack with push



The Stack

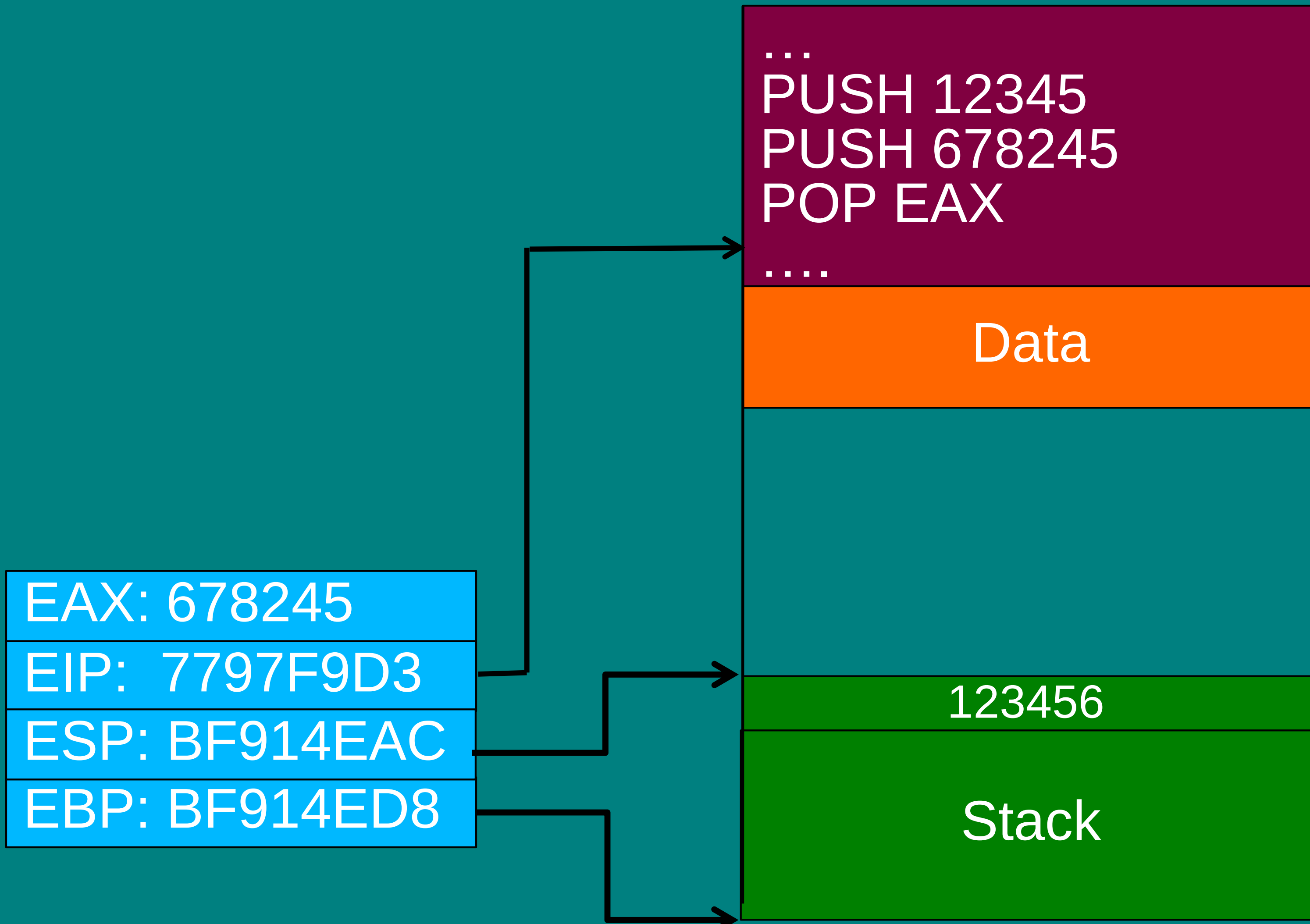

...
PUSH 12345
PUSH 678245
POP EAX
....
Data
123456
Stack
EAX:
EIP: 7797F9CF
ESP: BF914EAC
EBP: BF914ED8

The Stack



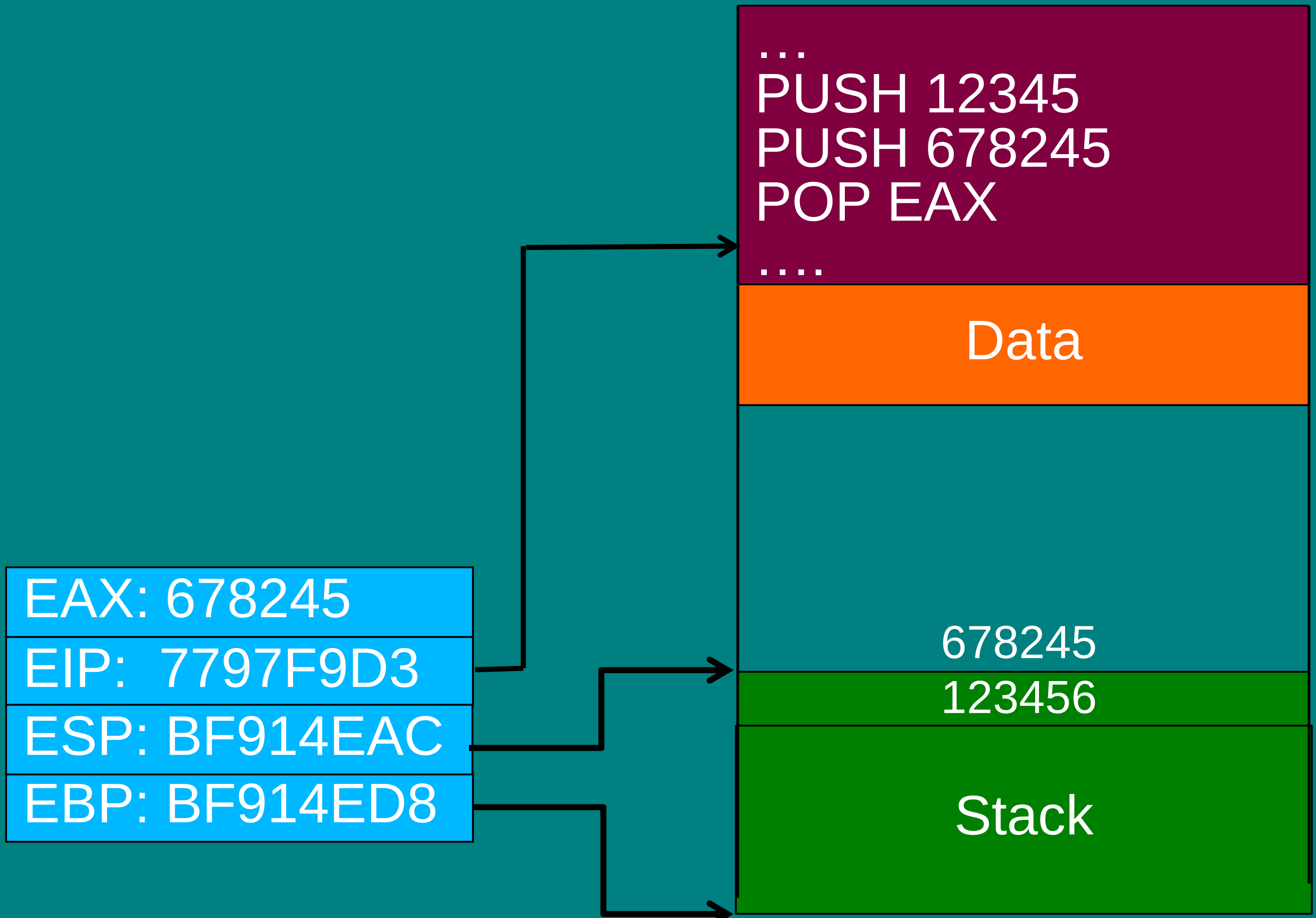
...
PUSH 12345
PUSH 678245
POP EAX
....
Data
678245
123456
Stack
EAX:
EIP: 7797F9D3
ESP: BF914EA9
EBP: BF914ED8

The Stack



...
PUSH 12345
PUSH 678245
POP EAX
....
Data
123456
Stack
EAX: 678245
EIP: 7797F9D3
ESP: BF914EAC
EBP: BF914ED8

The Stack


...
PUSH 12345
PUSH 678245
POP EAX
....
Data
678245
123456
Stack
EAX: 678245
EIP: 7797F9D3
ESP: BF914EAC
EBP: BF914ED8

Function calls (32-bit)

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

Function calls (32-bit)

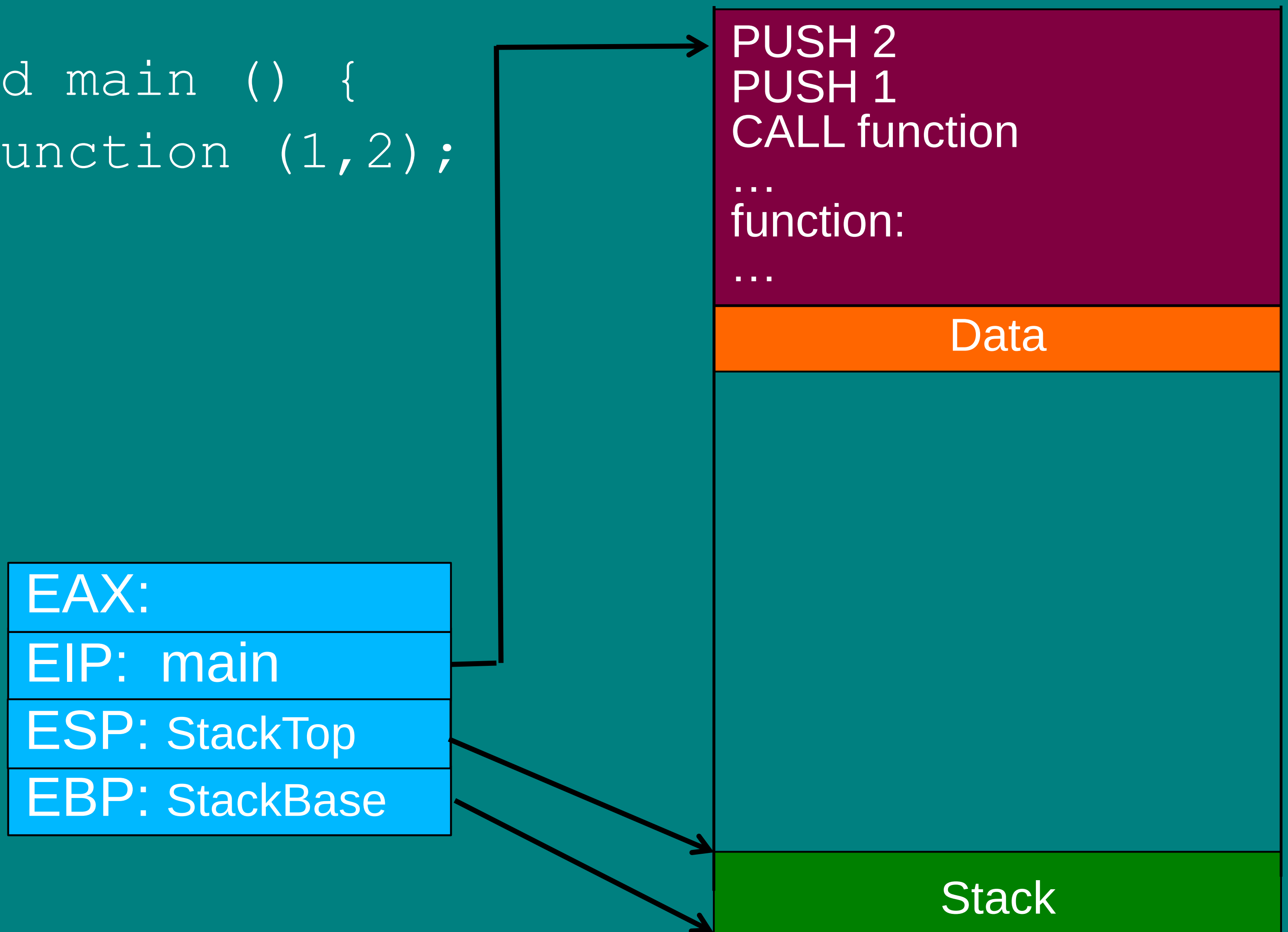
```
void main () {
   function (1,2);
}
```

```
PUSH <2>
PUSH <1>
CALL <function>
```

- Arguments 1 & 2 are passed on the stack.
- The CALL instruction puts the address of `function` into EIP and stores the old EIP on the stack.

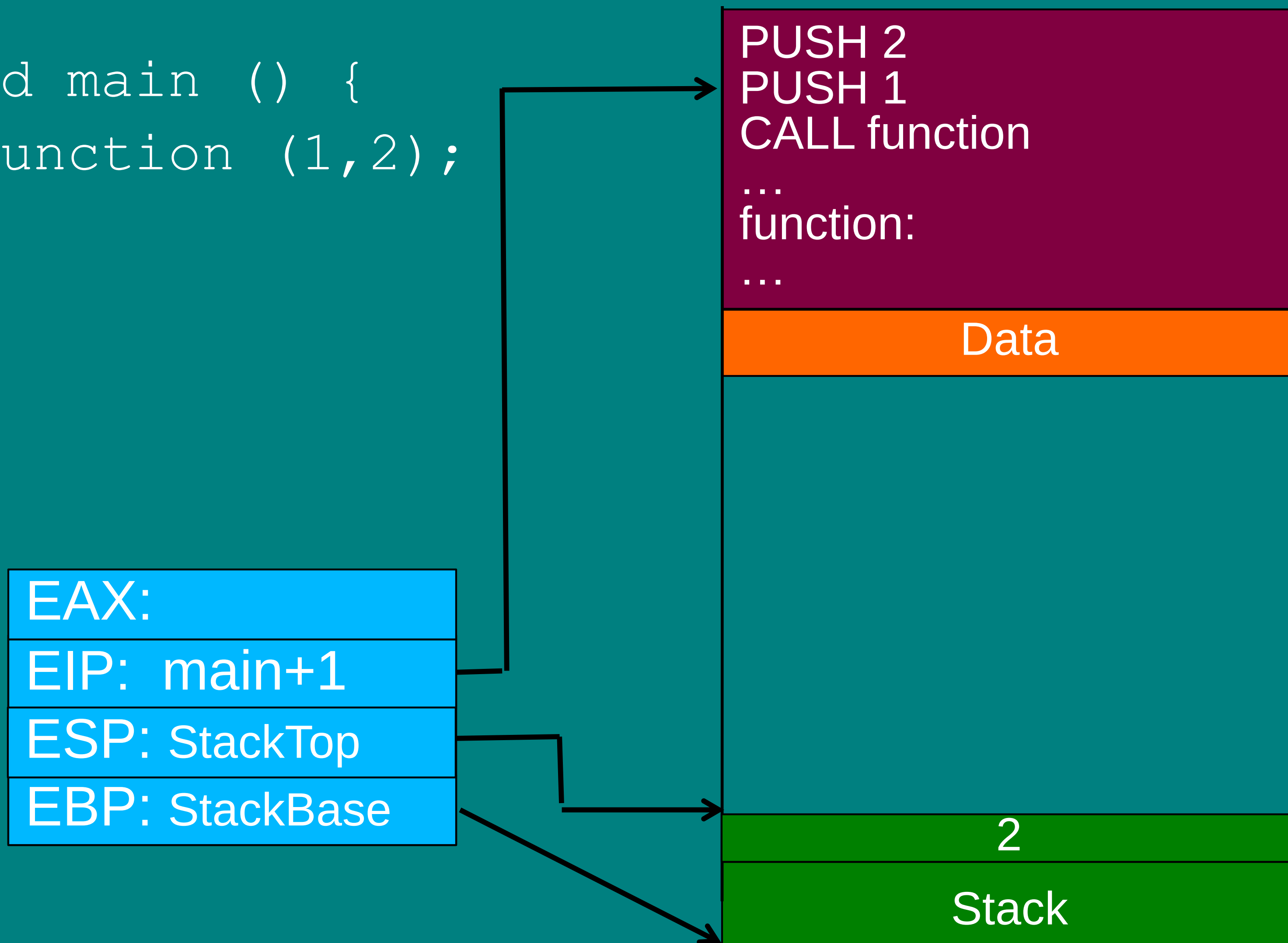
The Stack

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



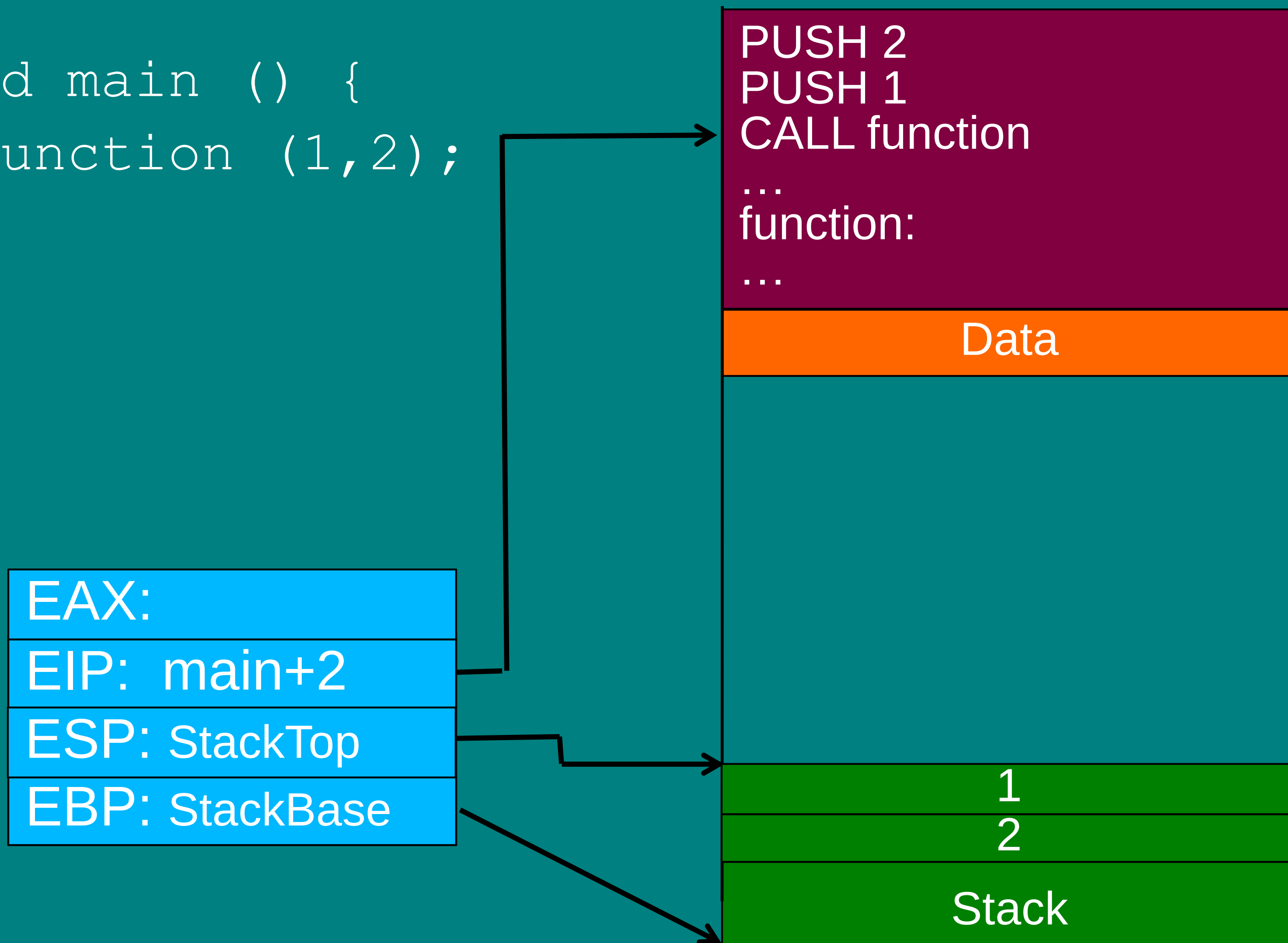
The Stack

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



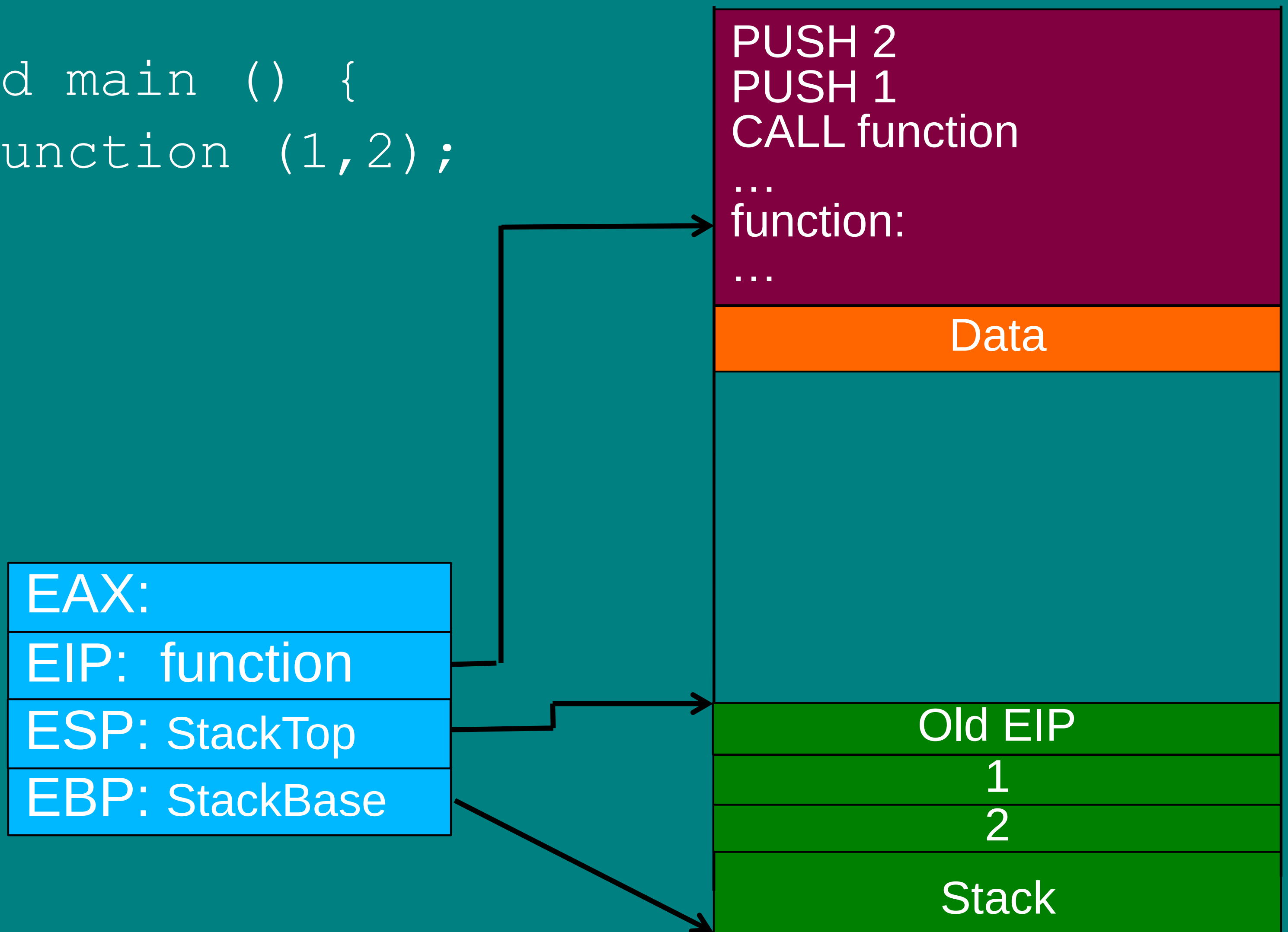
The Stack

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



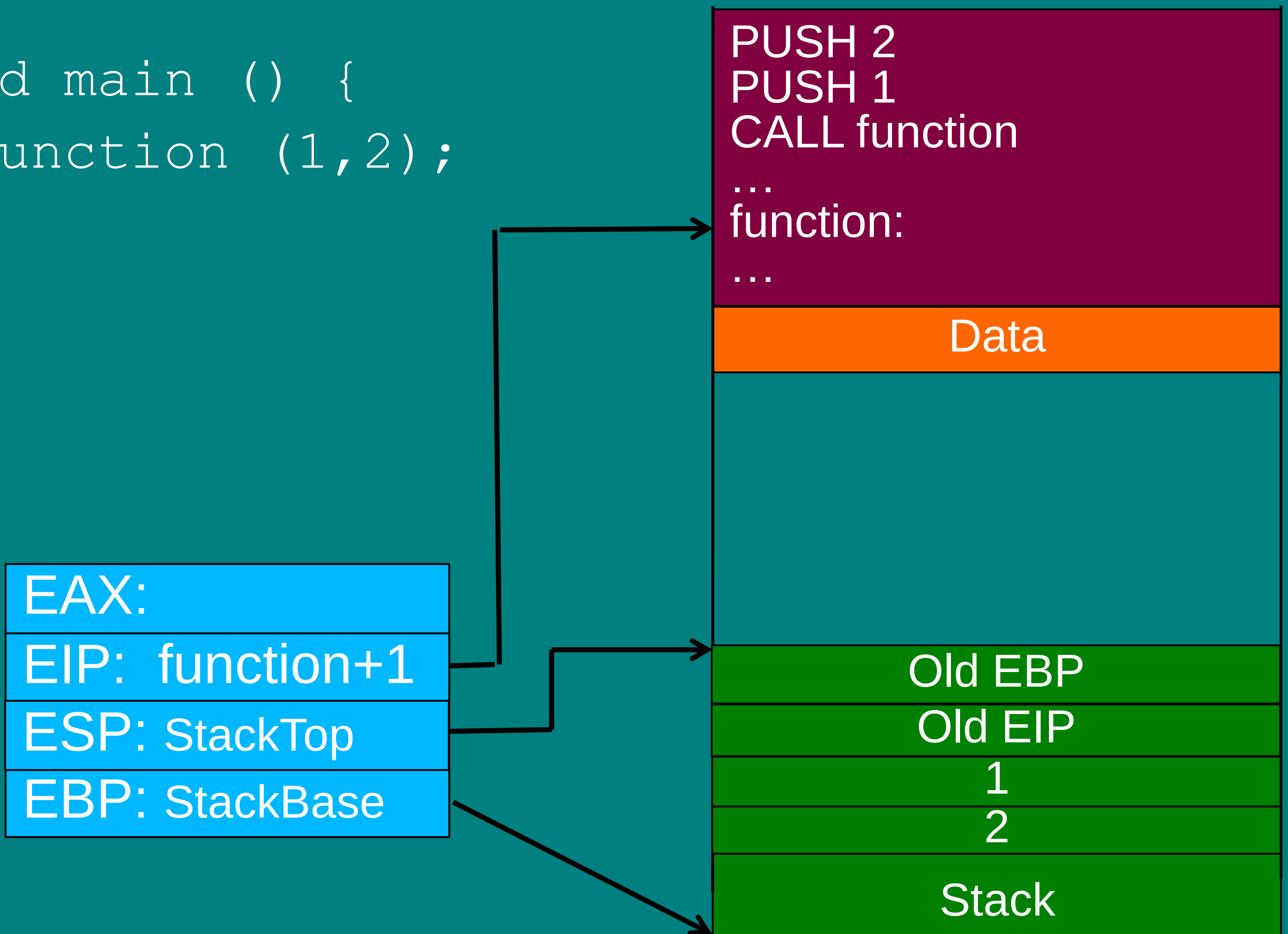
The Stack

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



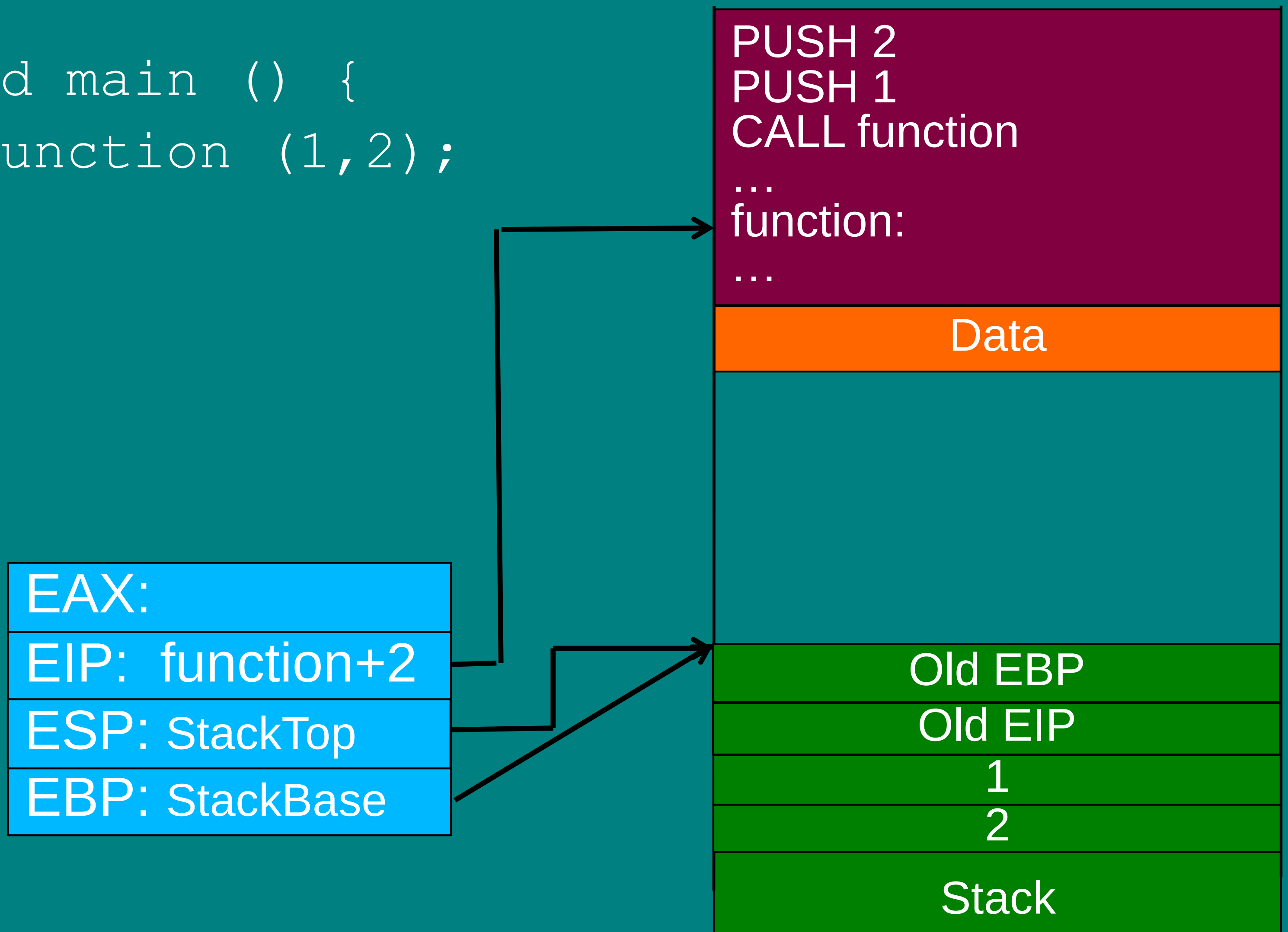
The Stack

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



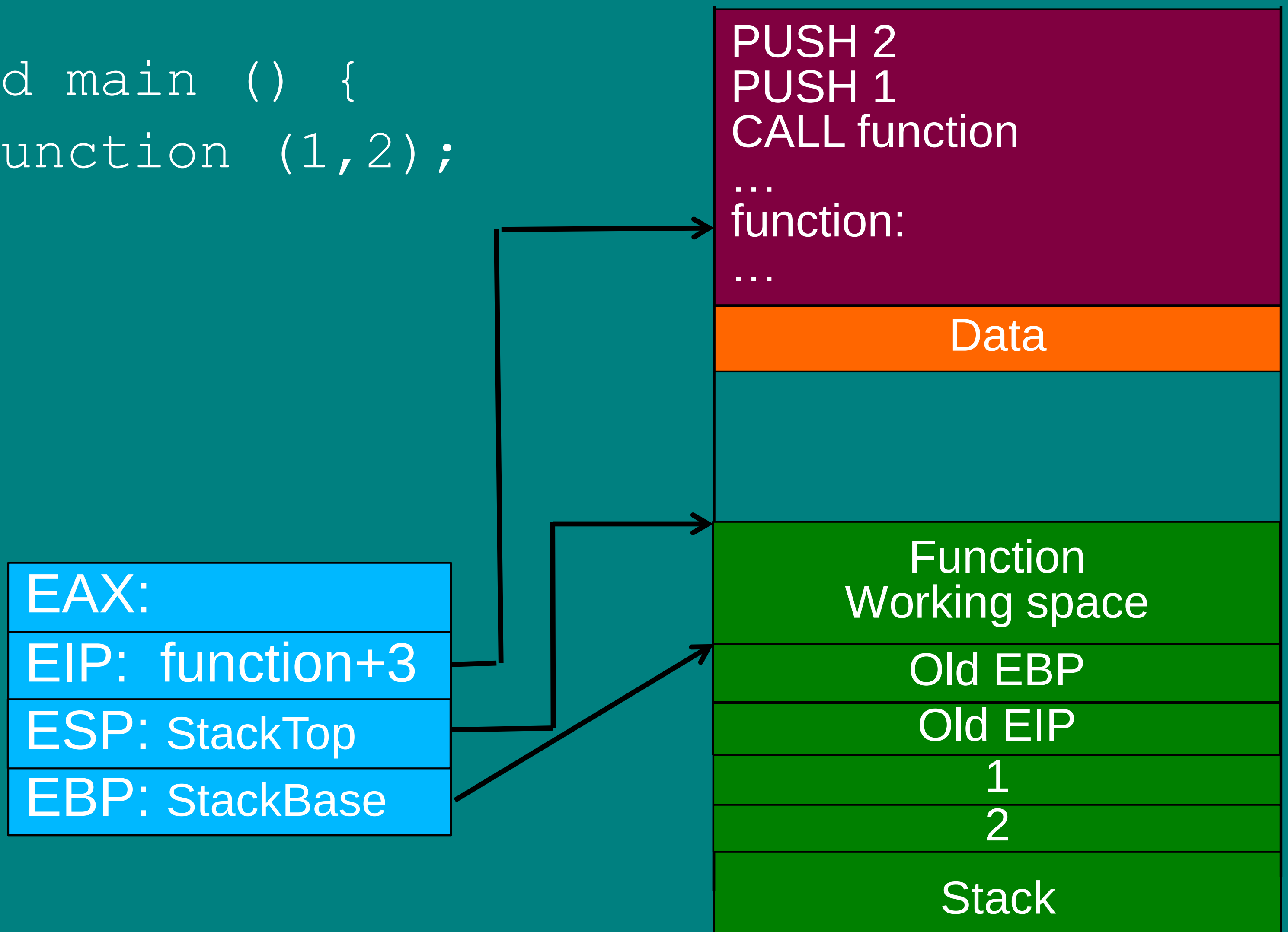
The Stack

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



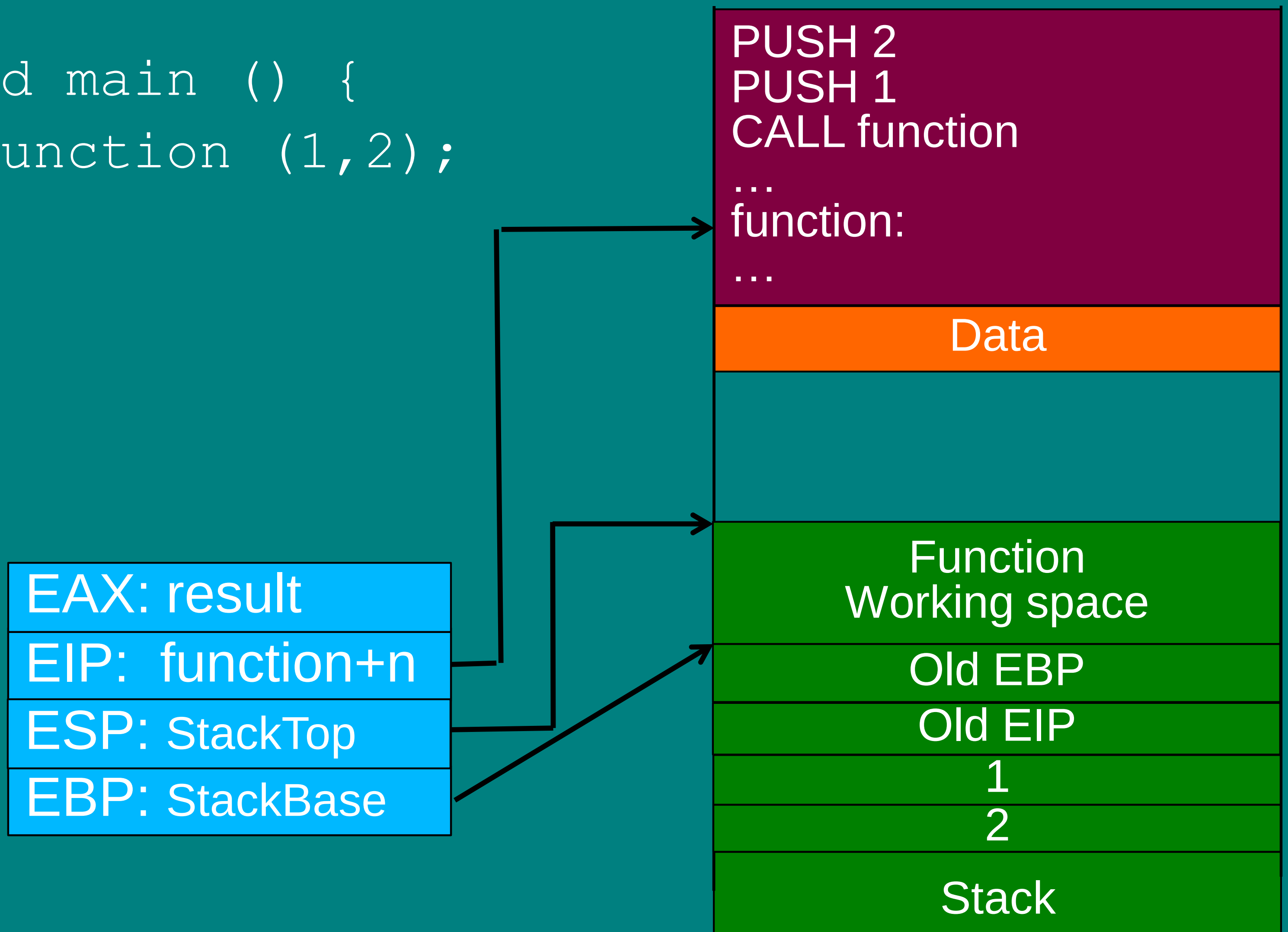
The Stack

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



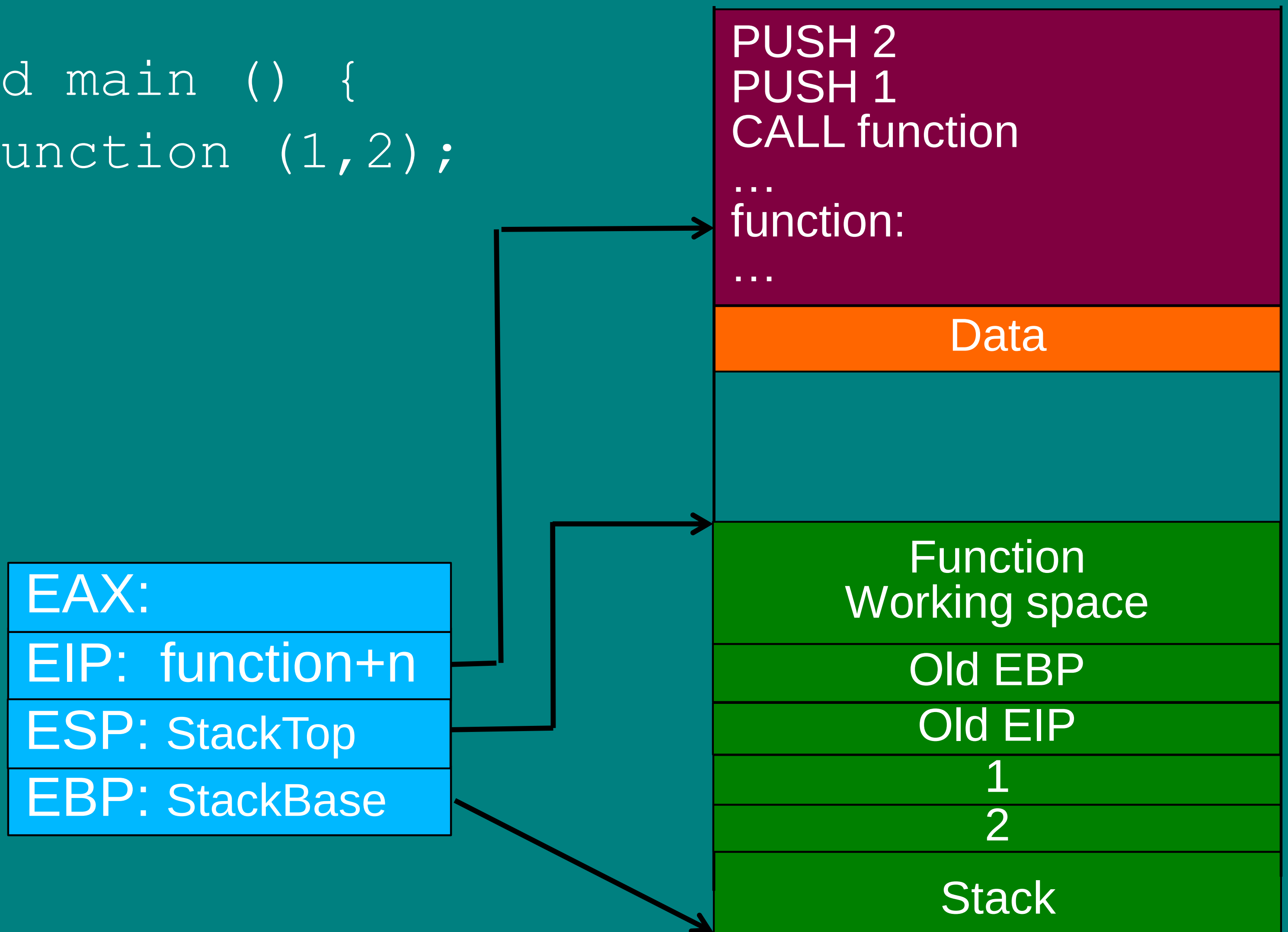
The Stack

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



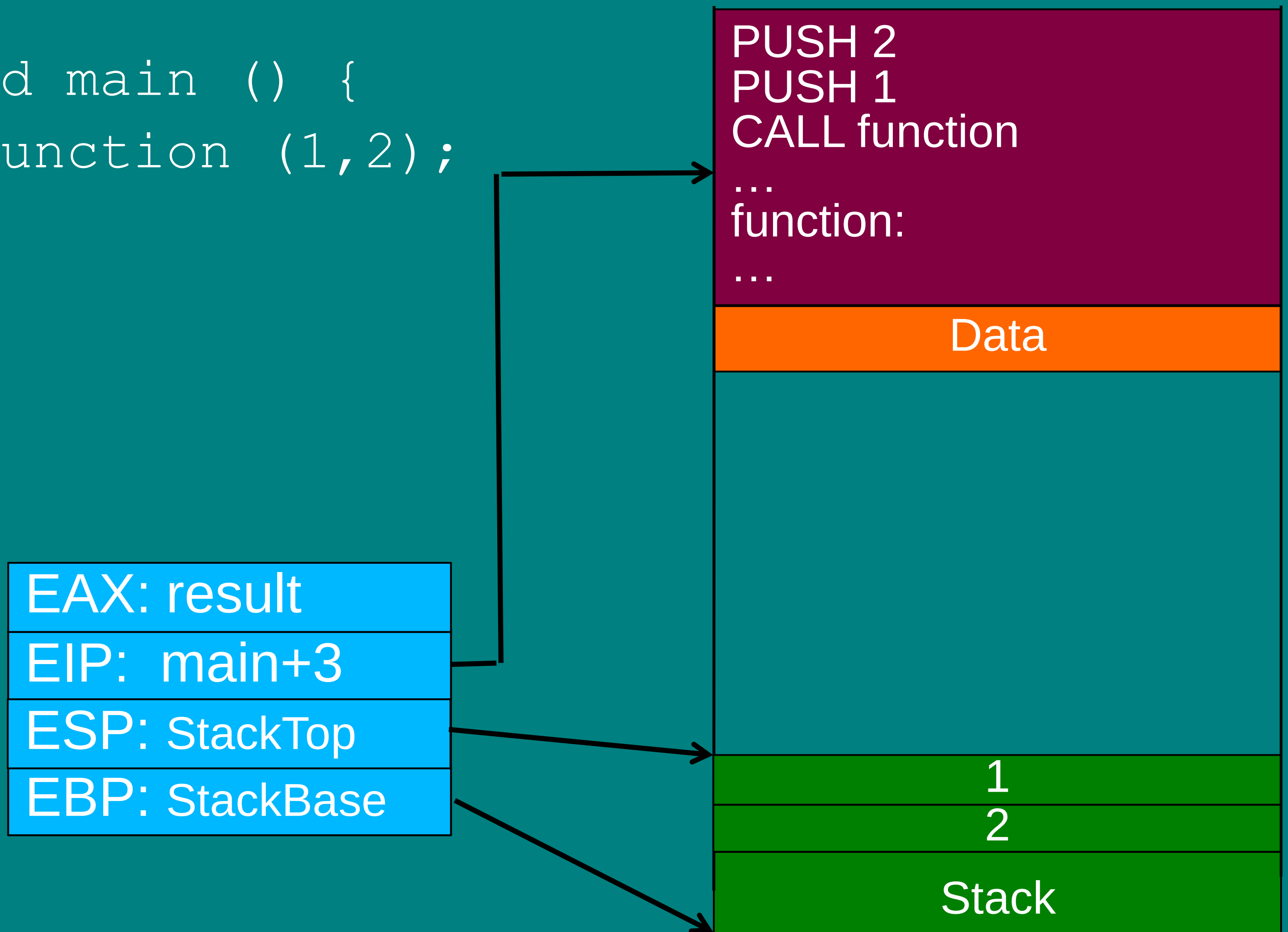
The Stack

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



The Stack

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



Buffer Overflows

- The instruction pointer controls which code executes,

Buffer Overflows

- The instruction pointer controls which code executes,
- The instruction pointer is stored on the stack,

Buffer Overflows

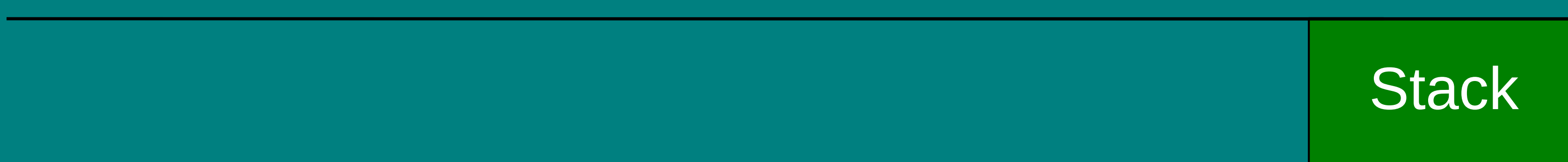
- The instruction pointer controls which code executes,
- The instruction pointer is stored on the stack,
- I can write to the stack …

Buffer Overflows

- The instruction pointer controls which code executes,
- The instruction pointer is stored on the stack,
- I can write to the stack … ☺

Buffers

```
…
getname();
…

getname() {
   char buffer[16];
   gets(buffer);
}
```



Buffers

1. Function called

```
…
getname();
…

getname() {
   char buffer[16];
   gets(buffer);
}
```

Buffers

1. Function called
2. EIP & EBP written to stack

```
...
getname();
...

getname() {
   char buffer[16];
   gets(buffer);
}
```

Buffers

1. Function called
2. EIP & EBP written to stack
3. Function runs

```
...
getname();
...

getname() {
  char buffer[16];
  gets(buffer);
}
```

Buffers

1. Function called
2. EIP & EBP written to stack
3. Function runs
4. Buffer allocated

```
…
getname();
…

getname() {
   char buffer[16];
   gets(buffer);
}
```

	← 16 byte →	Old EBP	Old EIP	Stack

Buffers

1. Function called
2. EIP & EBP written to stack
3. Function runs
4. Buffer allocated
5. User inputs “Hello World”

```
...
getname();
...

getname() {
   char buffer[16];
   gets(buffer);
}
```

	Hello World	Old EBP	Old EIP	Stack

Buffer Overflow

If input is >16 bytes:

Hello World XXXXXXXXXXXX

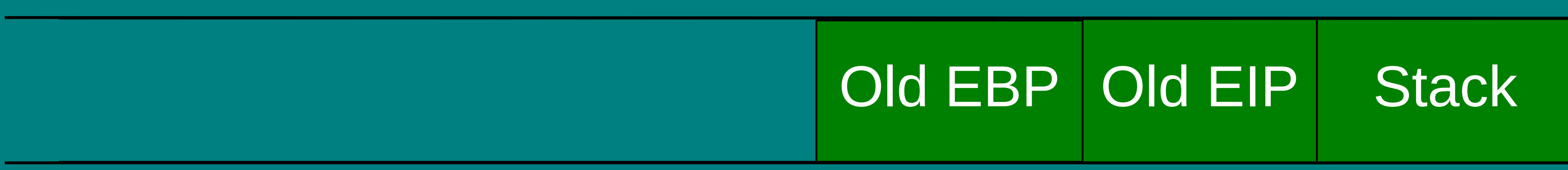
```
…
getname();
…

getname() {
  char buffer[16];
  gets(buffer);
}
```

Buffer Overflow

If input is >16 bytes:
Hello World XXXXXXXXXXXX

1. Runs as before

```
...
getname();
...

getname() {
  char buffer[16];
  gets(buffer);
}
```



Buffer Overflow

If input is >16 bytes:
Hello World XXXXXXXXXXXX

1. Runs as before

```
…
getname();
…

getname() {
  char buffer[16];
  gets(buffer);
}
```

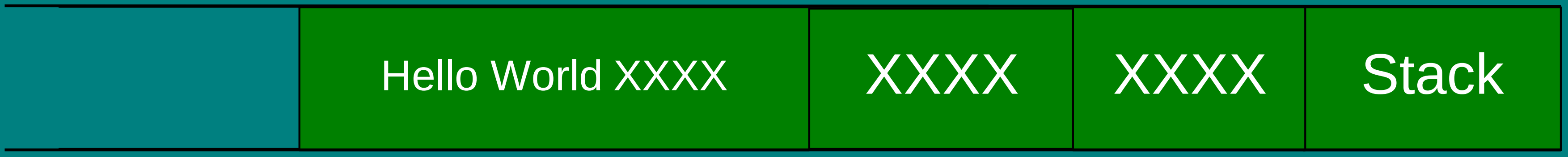
	← 16 byte →	Old EBP	Old EIP	Stack

Buffer Overflow

If input is >16 bytes:
Hello World XXXXXXXXXXXX

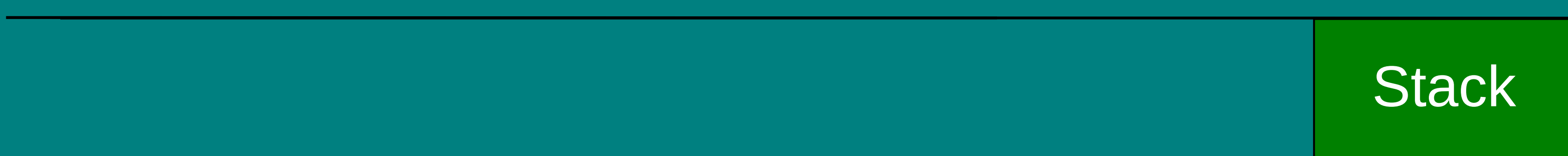
1. Runs as before
2. But the string flows over the end of the buffer
3. EIP corrupted, segmentation fault

```
…
getname();
…

getname() {
   char buffer[16];
   gets(buffer);
}
```



Once more, with malice

1. Runs as before



Once more, with malice

1. Runs as before
2. Attack send a very long message, ending with the address of some code that gives him a shell:

   Hello World XXXX XXXX97F9

Once more, with malice

1. Runs as before
2. Attack send a very long message, ending with the address of some code that gives him a shell:

   Hello World XXXX XXXX97F9
3. The attackers value is copied over the old EIP

Once more, with malice

1. Runs as before
2. Attack send a very long message, ending with the address of some code that gives him a shell:

   Hello World XXXX XXXX97F9

3. The attackers value is copied over the old EIP
4. When the function returns the attacks code (here: at 0x97f9) is run

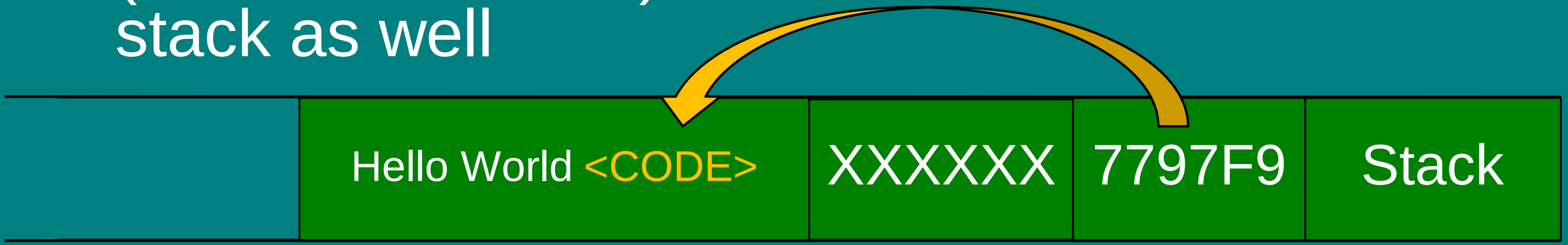
	Hello World XXXX	XXXX	97F9	Stack

Once more, with malice

1. Runs as before
2. Attack send a very long message, ending with the address of some code that gives him a shell:

   Hello World XXXX XXXX97F9

3. The attackers value is copied over the old EIP
4. When the function returns the attacks code (here: at 0x779ff9) is run: this code can be on the stack as well



Live-Demo

“Anything that can go wrong, will go wrong”

Debugging a buffer overflow with gdb

A few remarks

- In the above example we simplified notation by mixing HEX and ASCII
- In 32-bit mode: EIP/EBP are 4 byte each
- In 64-bit mode: RIP/RBP are 8 byte each
- There might be some padding
- Easiest to experimentally find the offset of EIP/RIP on the stack with a debugger

Live-Demo

"Anything that can go wrong, will go wrong"

Exploiting a buffer overflow

What To Inject

Shell code (under Linux) is assembly code for

```
exec("/bin/bash", {NULL}, NULL)
```

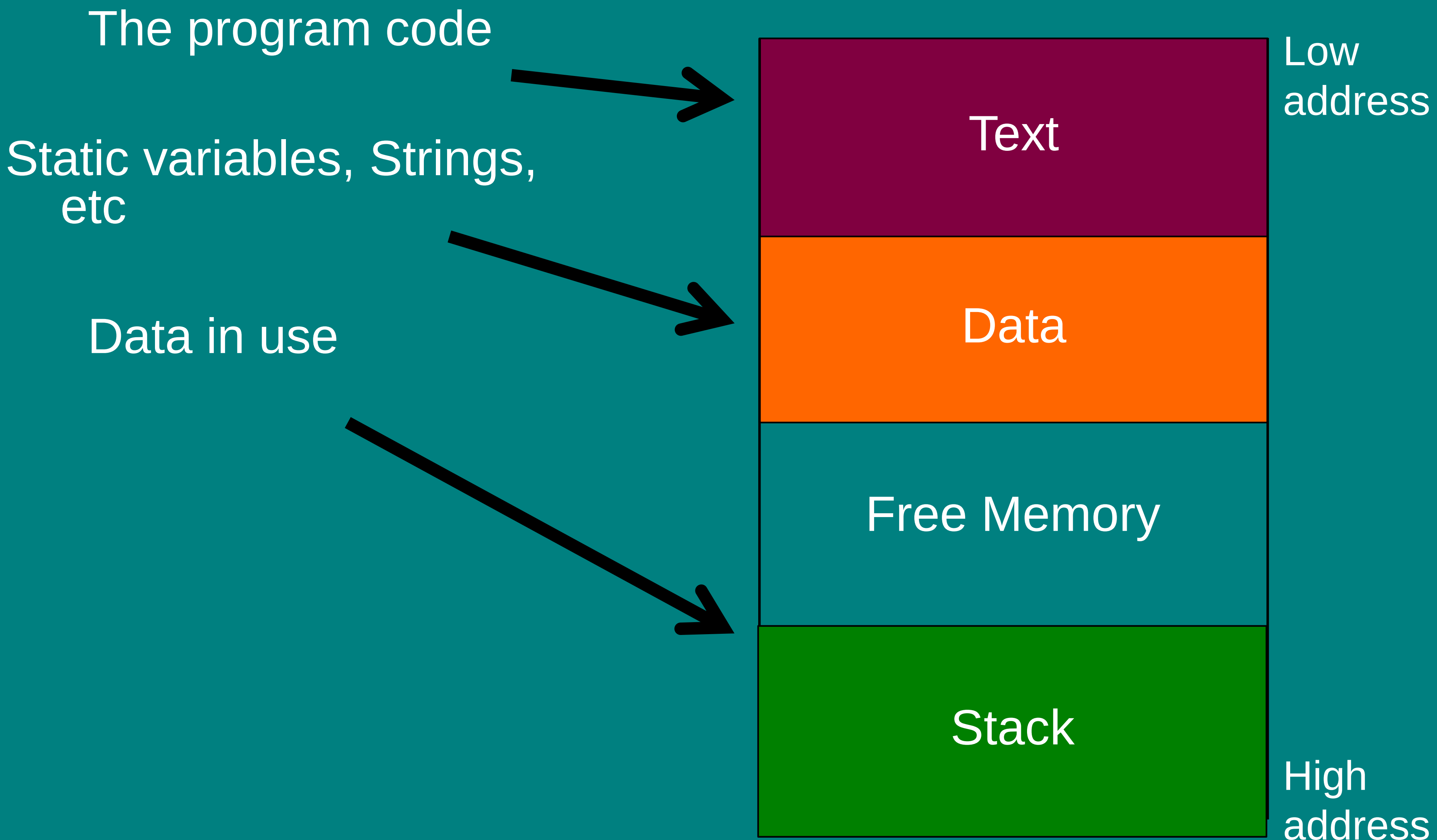
There are some defenses in modern Linux, hence use that to indirectly call a binary that first calls `setuid(0)` and then spawns a shell (see `msh.c` on Canvas)

Live-Demo

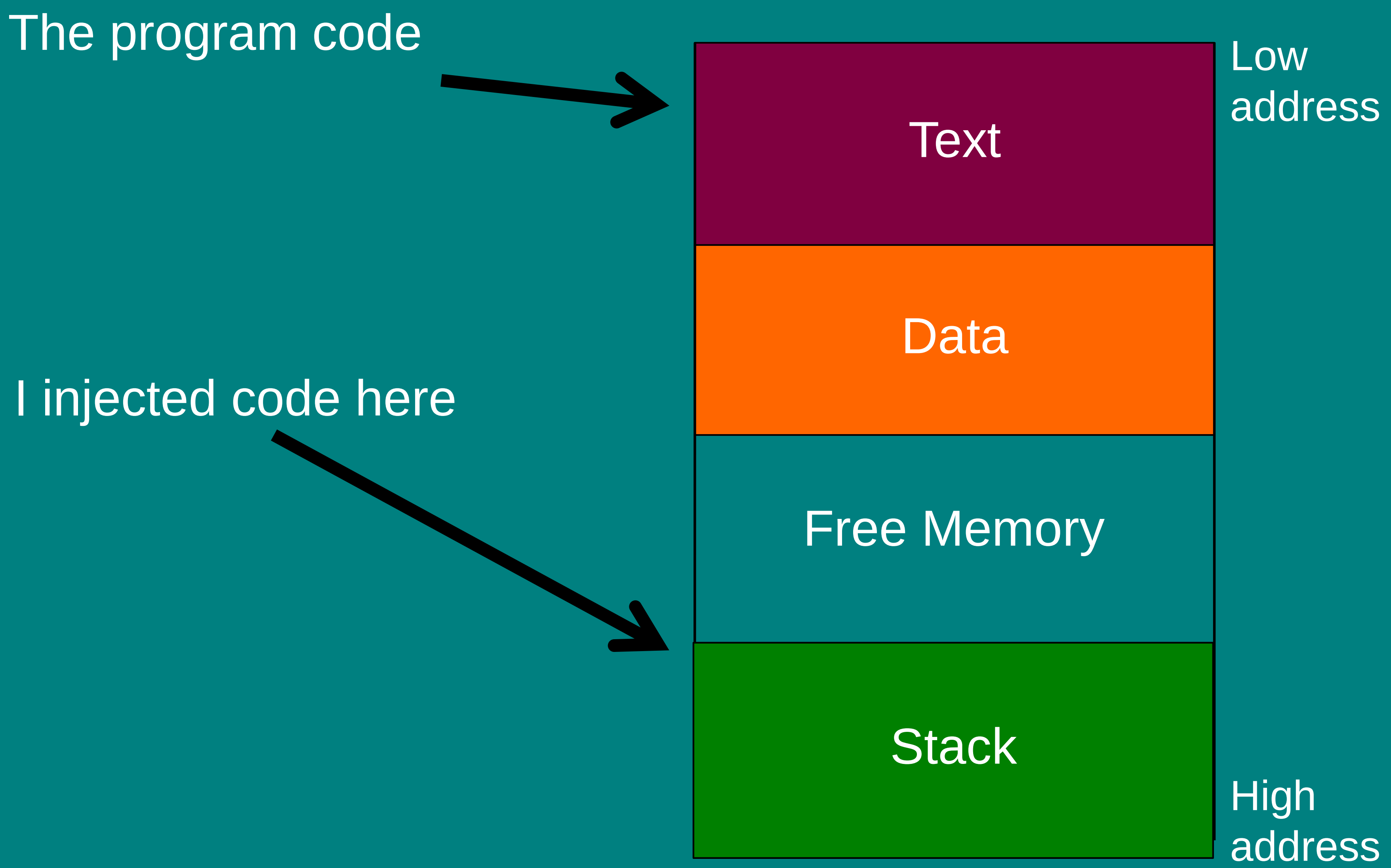
"Anything that can go wrong, will go wrong"

Exploiting a buffer overflow to pop a shell

Defense: The NX-bit

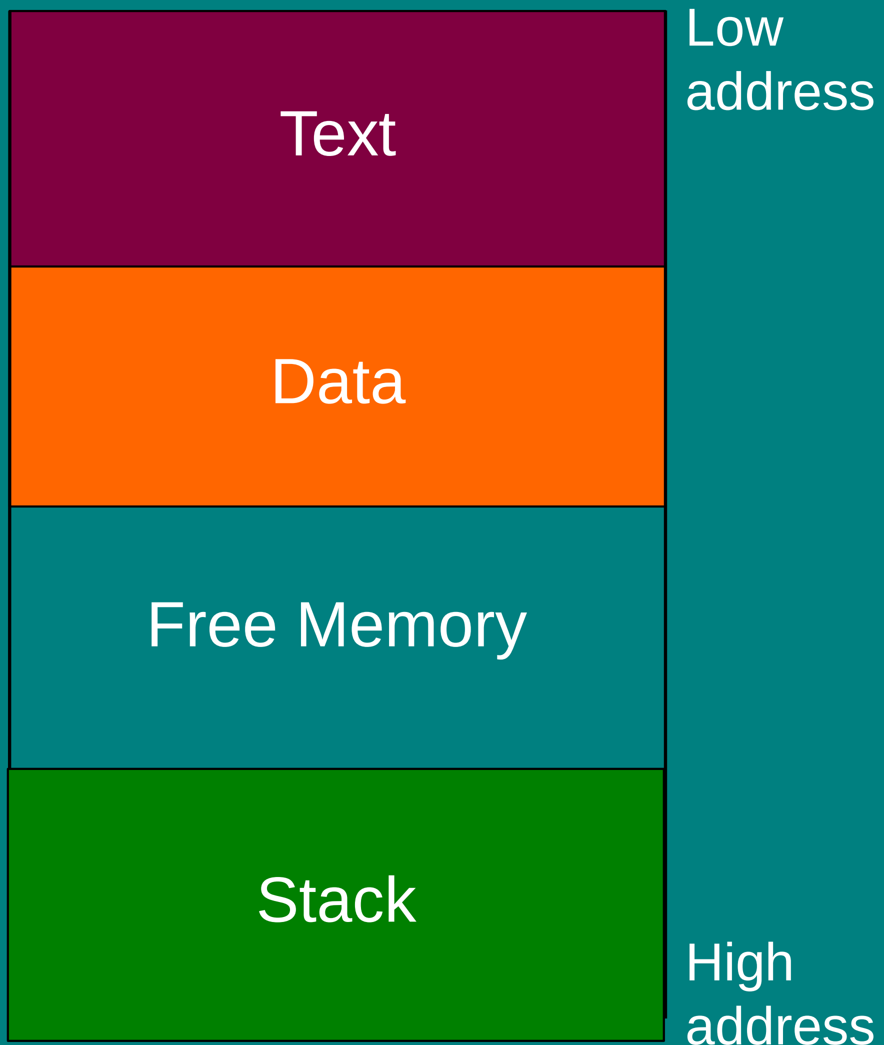


Defense: The NX-bit



Defense: The NX-bit

- Code should be in the text area of the memory. Not on the stack.
- The NX-bit provides a hardware distinction between the text and stack.
- When enabled, the program will crash if the EIP ever points to the stack.



Reuse Code.

The standard attack against the NX-bit is to reuse code from the executable part of memory. E.g.

- Jump to another function in the program.
- Jump to a function from the standard C library (Return to libc)
- String together little pieces of existing code (Return-oriented programming).

Return-to-libc

- Libc is the C standard library.
- It is often packaged with executables to provide a runtime environment.
- It includes lots of useful calls like “system” which runs any command.
- It links to executable memory, therefore bypasses NX-bit protections.

Address space layout randomization.

- ASLR adds a random offset to the stack and codes base each time the program runs.
- Jumps in the program are altered to point to the right line.
- The idea is that its now hard for an attacker to guess the address of where they inject code or the address of particular functions.
- On by default in all OS
  - Off in Linux: sudo echo 0 > /proc/sys/kernel/randomize_va_space
  - On in Linux: sudo echo 1 > /proc/sys/kernel/randomize_va_space

NOP slide

- In x86 the op code assembly instruction 0x90 does nothing.
- If the stack is 2MB, I could inject 999000 bytes of 0x90 followed by the my shell code, after the return pointer.
- I then guess a return address and hope it is somewhere in the 2MB of NOPs.
- If it is, the program slides down the NOPs to my shell code.

- Often used with other methods of guessing the randomness.

Metasploit

- Metasploit is a framework for testing and executing known buffer overflow attacks.
- If a vulnerability in an application is well known their will be a patch for it, but also a Metasploit module for it.
- If an application is unpatched it can probably be taken over with Metasploit.
- Metasploit also includes a library of shell code which can be injected.

Recommend Paper:

- “Smashing the Stack for Fun and Profit”
  Elias Levy (Aleph One)

- A simple introduction to buffer overflows from the mid 90s.

- Standard defenses now stop the attacks in this paper, but it gives an excellent introduction.

Conclusion

Buffer overflows are the result of poor memory management in languages like C: even the “best” programmers **will** make mistakes.

Buffer overflow attacks exploit these to overwrite memory values.

This lets an attack execute arbitrary code.