# Lecture #3: Background for memory-based attacks

UCalgary ENSF619

Elements of Software Security

*Instructor: Lorenzo De Carli (lorenzo.decarli@ucalgary.ca)*

# Content of this lecture

1. Memory-based exploits: what are those?
2. Why is understanding memory management important?
    - (from a security point of view)
3. How memory works in traditional computing architectures
4. What comes next

# Software exploits

- **Software can be attacked in many ways**
  - We discussed possible threats/attacker goals last time
  - But what are the **strategies** being used?
- Typically, an attacker constructs one or more **exploits** to achieve their goal
  - "Exploit: a method or piece of code that takes advantage of vulnerabilities in software" (*https://en.wikipedia.org/wiki/Exploit_(computer_security)*)
  - A successful exploit (or chain of exploit) may result in the attacker **gaining control of execution**, **accessing sensitive data**, and/or **crashing the program**

# Memory-based software exploits

- Some exploits take advantage of high-level design flaws, human weaknesses, misconfigurations, etc.
- Other (the oldest and arguably most pernicious form of attack) take advantage of **flaws in program binaries** themselves
- These exploits **flaws** in the way programs **manage their memory**
  - (thus called memory-based attacks/exploits)

# Are these exploits relevant?

- Let's look at CWE Top 25 2024

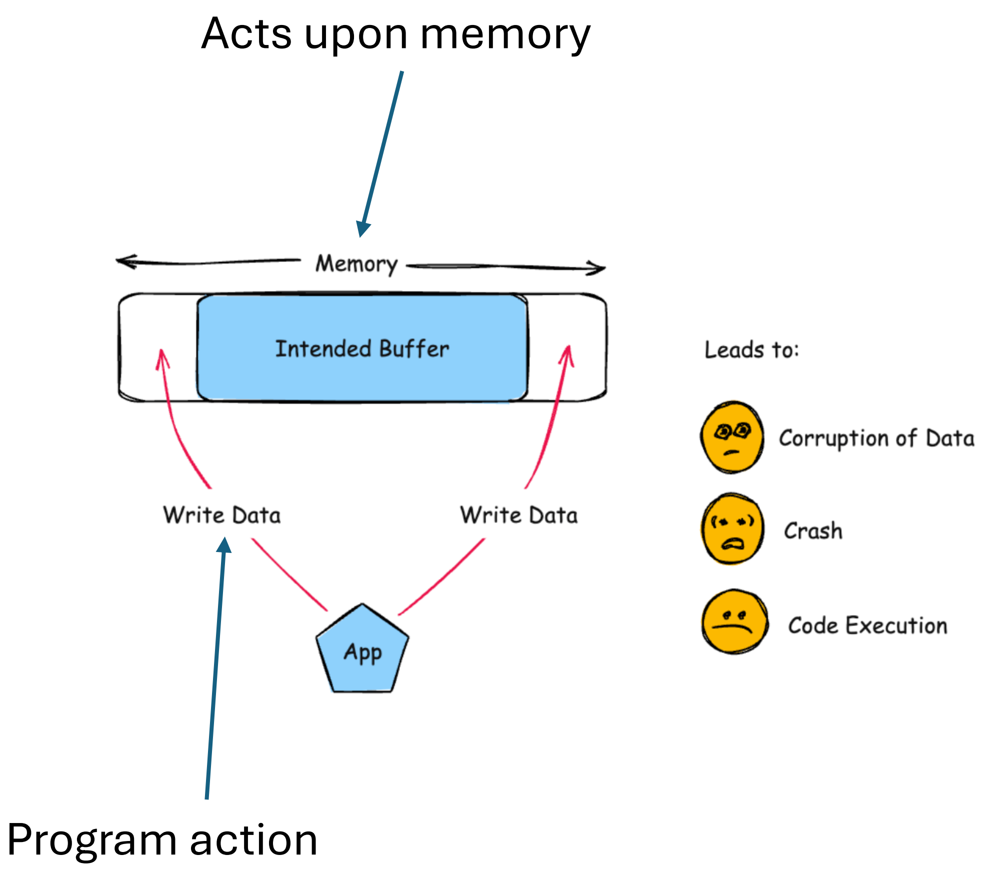
| | |
|---|---|
| 1 | Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')<br>**CWE-79** \| CVEs in KEV: 3 \| Rank Last Year: 2 (up 1) ▲ |
| 2 | Out-of-bounds Write<br>**CWE-787** \| CVEs in KEV: 18 \| Rank Last Year: 1 (down 1) ▼ |
| 3 | Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')<br>**CWE-89** \| CVEs in KEV: 4 \| Rank Last Year: 3 |
| 4 | Cross-Site Request Forgery (CSRF)<br>**CWE-352** \| CVEs in KEV: 0 \| Rank Last Year: 9 (up 5) ▲ |
| 5 | Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')<br>**CWE-22** \| CVEs in KEV: 4 \| Rank Last Year: 8 (up 3) ▲ |

- (from *https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html*)

# Let's dig deeper!

The product writes data past the end, or before the beginning, of the intended buffer.

*https://cwe.mitre.org/data/definitions/787.html*



# How can one exploit program memory?

- This is typically accomplished by feeding the program **malformed/incorrect input**
- Programs use their working memory to:
  - **Store input**
  - **Process input**
- Both tasks can be commandeered if the program exhibit **specific types of bugs**
- **The end result is that the attacker can control the control-flow of the program**

# Which kinds of software can be exploited?

- Programs can be **hardened** against these exploits by incorporating various kind of **checks**
  - E.g., checking that a memory object has enough capacity to accommodate data being written to it
- These checks may be **expensive** to incorporate  in runtimes as they need to be **performed frequently**
- **Interpreted languages** will oftentimes incorporate these checks, while **native programs** may not for efficiency reasons
- Thus, while exceptions abound, these issues typically affect **native (binary) code**

# Types of software exploits

- **Overflows**
  - In the stack
  - In the heap
- **Return-oriented programming**
  - More advanced form that bypasses some defenses against overflows

# Understanding memory management

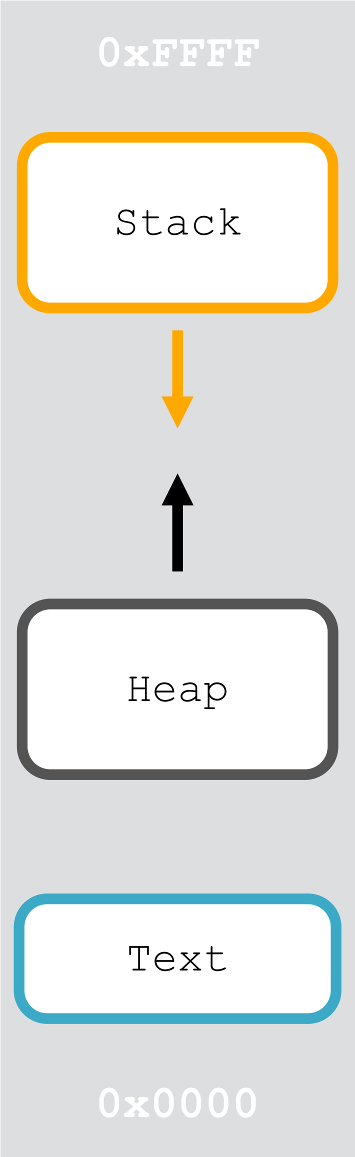
- Memory-based attacks can be simpler or complex, but typically exploits **low-level details of how memory is managed**
- They cannot be understood without a basic grasps of **how programs manage memory**
- In the rest of this lecture, we'll review **basic concepts of memory management**

# A brief review of memory management

# Caveat

- We are going to keep the discussion as **architecture-independent** as possible
- The principles discussed here apply to a **broad range** of **computer architectures** (x86, ARM, etc.) and **OS'es** (Windows, Linux etc.)
- To make the discussion more concrete, we are going to refer to **Linux on x86**

# Virtual address space




- The **virtual address space** is abstraction of the physical memory that makes memory simple for the process, e.g., a byte stream.
- Each byte in memory is associated with an **address**, allowing the process to access the memory location.
- We've divided the address space into three segments:
  - stack: used to support function calls and local variables, grows and shrinks during execution.
  - heap: used for dynamically-allocated, user-managed memory.
  - text: the instructions of the program
- We also need to set aside some space for the **operating system** and for **libraries**.

# Who creates/manages the virtual address space

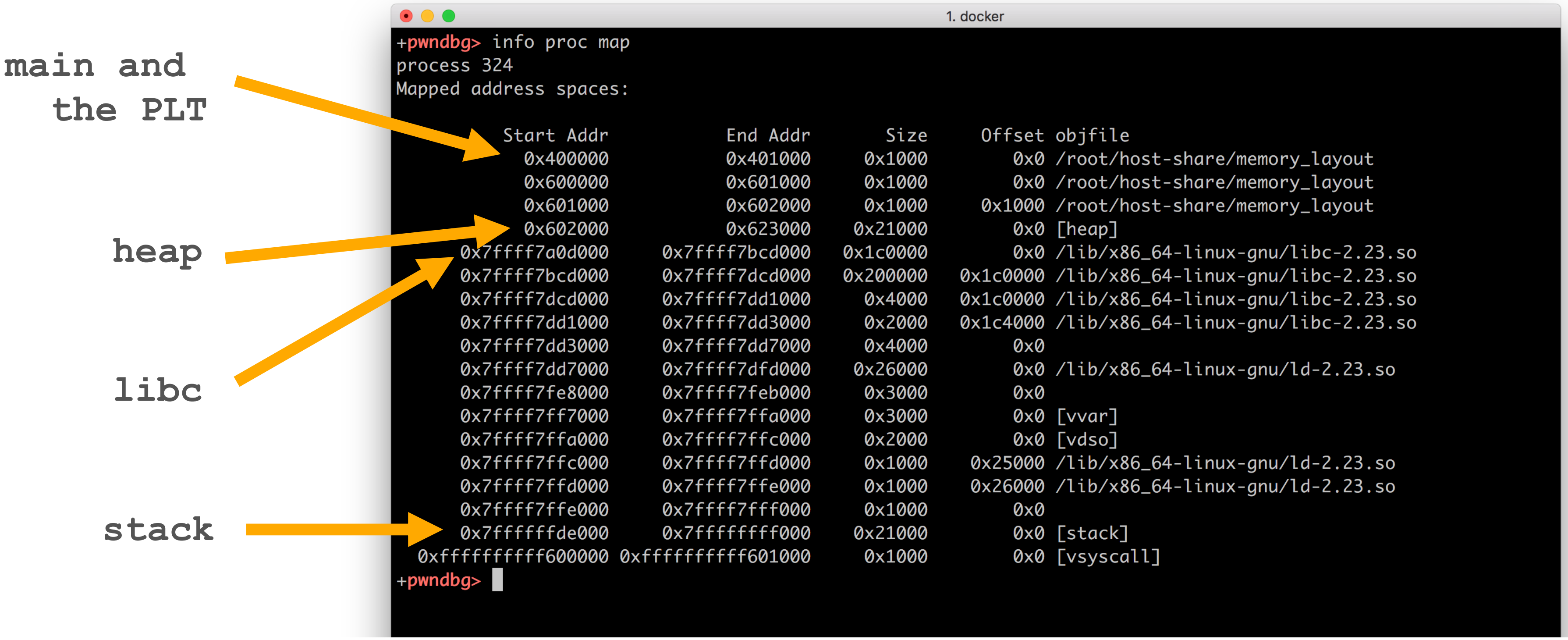
- **The OS!** Whenever a new process is created, the OS initialize the relevant **data structures** and **hardware controls**
- Together, OS+HW to provide the **virtual address space abstraction**
- After the address space is initialized by the kernel, memory management of that space is largely **up to the process**
- OS only intervene in case of **memory errors**!

# Example: memory map

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int x = 777;
  printf("location of code: %p\n", (void *) main);
  printf("location of heap: %p\n", (void *) malloc(1));
  printf("location of stack: %p\n", (void *) &x);
  printf("location of printf: %p\n", (void *) printf);
  printf("location of malloc: %p\n", (void *) malloc);
  return 0;
}
```

> Let’s run this code and make some observations.




- We can also view the **memory map** in GDB.
- Every address matches the previous printout, except for the stack. This is due to **Address Space Layout Randomization** (ASLR)
- Note, the **heap** won’t appear in this map until after the call to malloc.

# The stack

> The stack is used for local variables and all of the data needed to make function calling work:

- function arguments, return addresses, saved stack pointers, saved frame pointers.

> The stack is an example of **implicitly managed** memory, also known as **automatic** memory.

- This means that the programmer doesn't need to explicitly allocate and deallocate memory on the stack.

> Every change to the stack pointer is either an **allocation** or **deallocation** of memory.

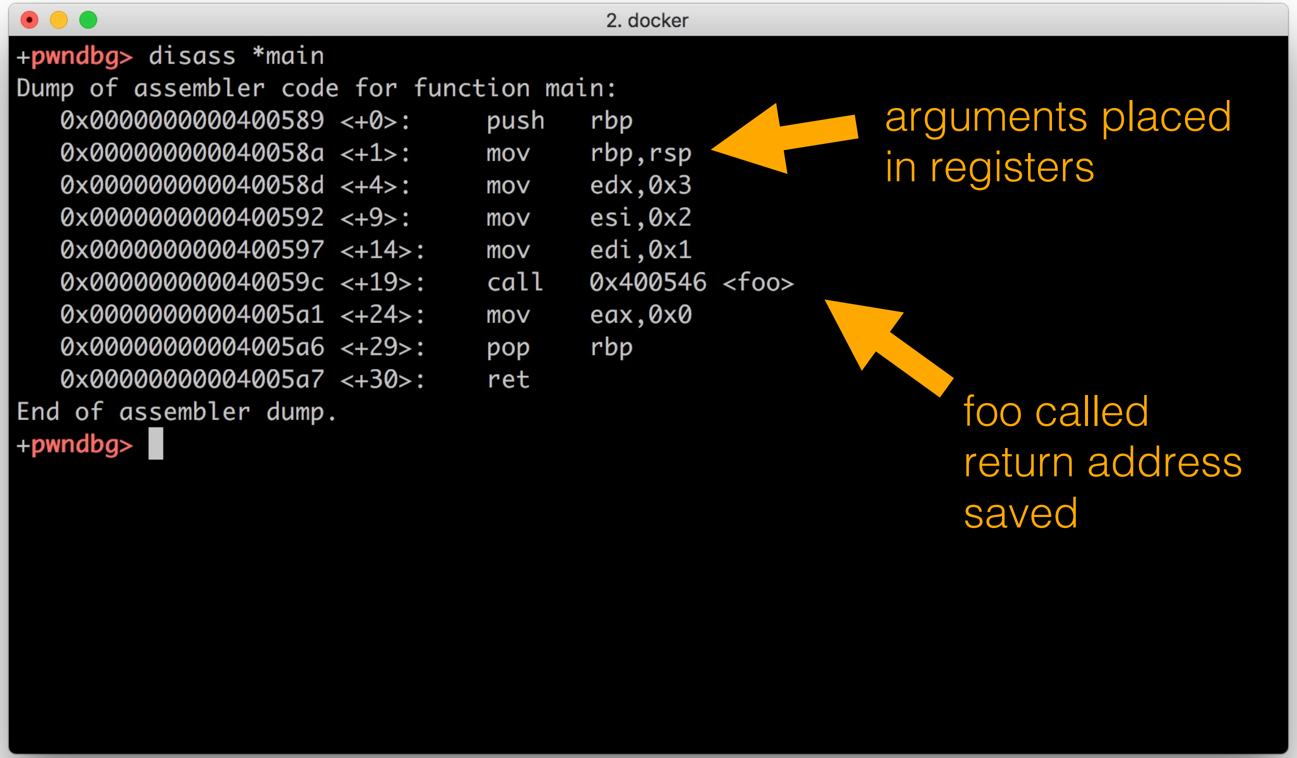
> Let's look at a simple example…

```c
void foo(int a, int b, int c) {
  char* buf[16];
}

int main() {
  foo(1, 2, 3);
}
```

"stack.c" 8L, 84C written

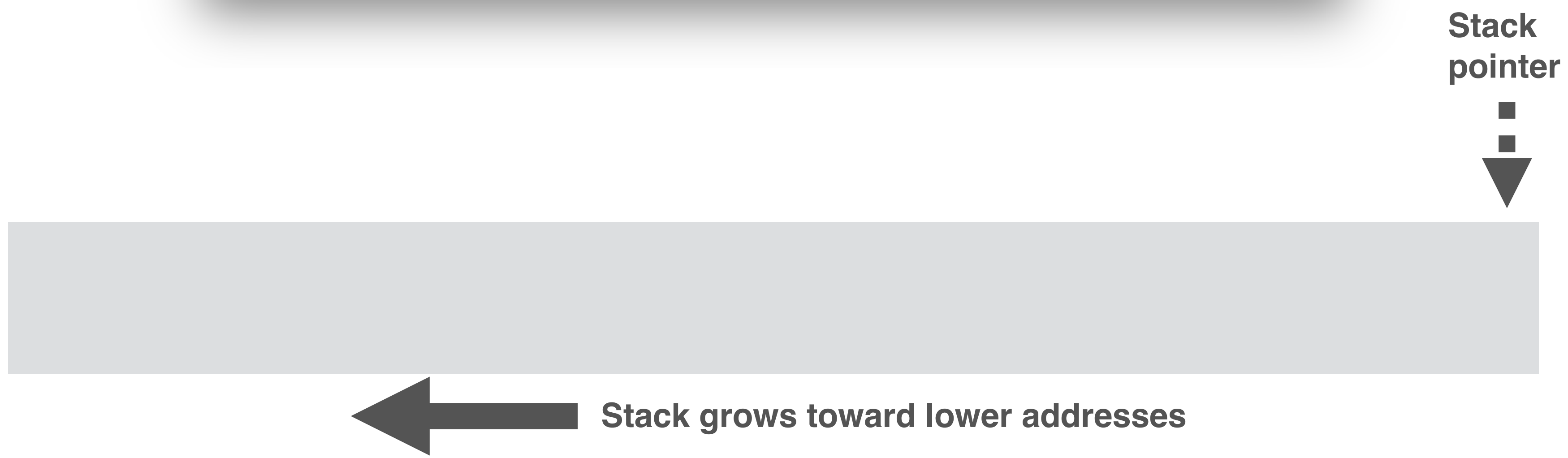
- Consider how the **stack** supports this **function call**.
- The **compiler** must allocate memory for the **arguments** to foo (a, b, c), the local variable buf, and control metadata.




> The call instruction pushes the **return address** to the stack. This push is a **memory allocation**.

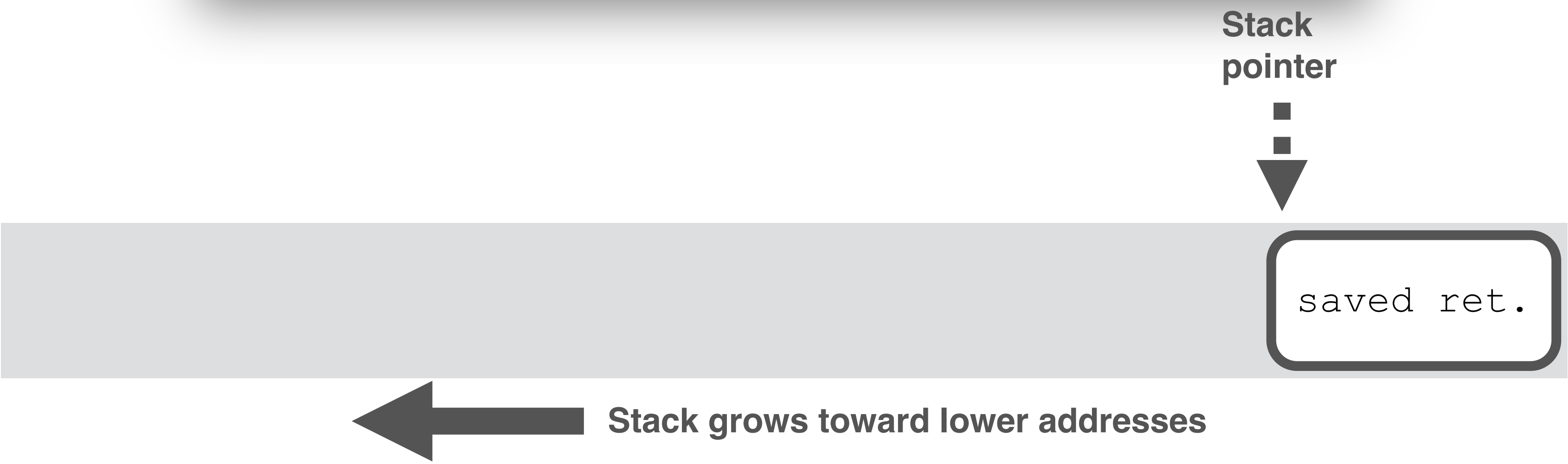
```
+pwndbg> disass *main
Dump of assembler code for function main:
   0x0000000000400589 <+0>:     push   rbp
   0x000000000040058a <+1>:     mov    rbp,rsp
   0x000000000040058d <+4>:     mov    edx,0x3
   0x0000000000400592 <+9>:     mov    esi,0x2
   0x0000000000400597 <+14>:    mov    edi,0x1
   0x000000000040059c <+19>:    call   0x400546 <foo>
   0x00000000004005a1 <+24>:    mov    eax,0x0
   0x00000000004005a6 <+29>:    pop    rbp
   0x00000000004005a7 <+30>:    ret
End of assembler dump.
+pwndbg>
```

foo called
return address
saved



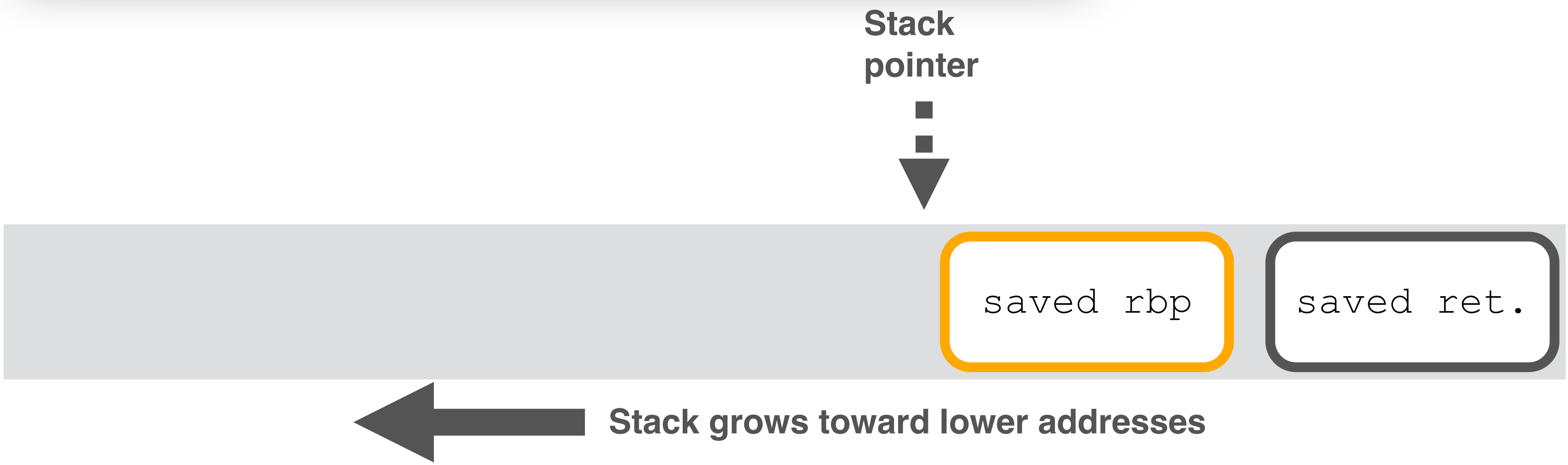
```
+pwndbg> disass *main
Dump of assembler code for function main:
   0x0000000000400589 <+0>:     push   rbp
   0x000000000040058a <+1>:     mov    rbp,rsp
   0x000000000040058d <+4>:     mov    edx,0x3
   0x0000000000400592 <+9>:     mov    esi,0x2
   0x0000000000400597 <+14>:    mov    edi,0x1
   0x000000000040059c <+19>:    call   0x400546 <foo>
   0x00000000004005a1 <+24>:    mov    eax,0x0
   0x00000000004005a6 <+29>:    pop    rbp
   0x00000000004005a7 <+30>:    ret
End of assembler dump.
+pwndbg>
```

foo called
return address
saved



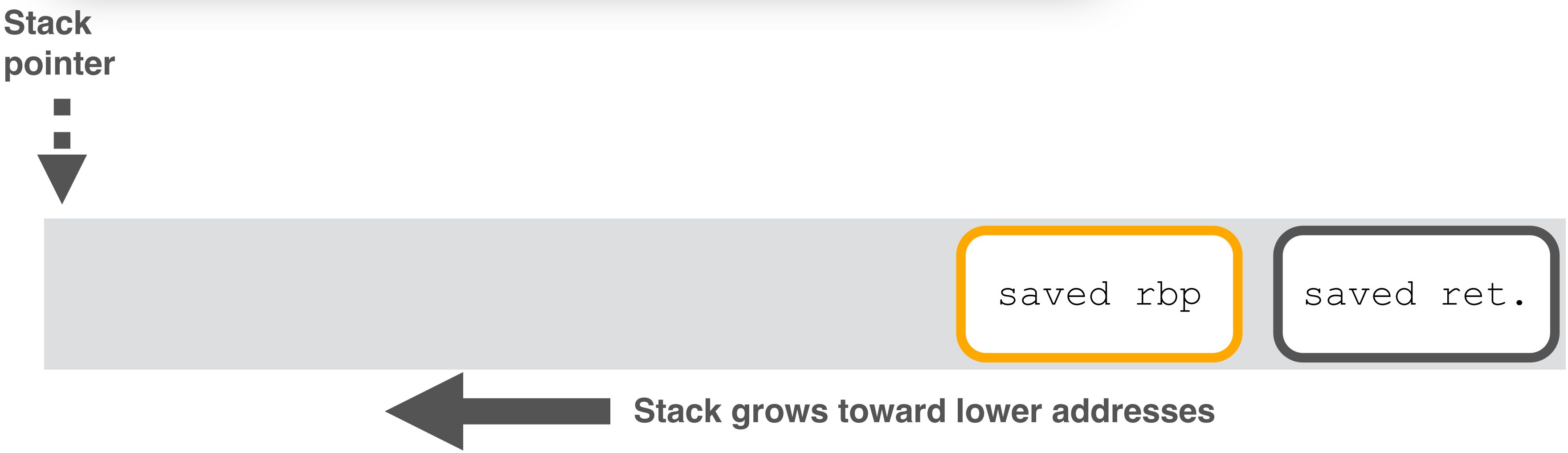
```
+pwndbg> disass *foo
Dump of assembler code for function foo:
   0x0000000000400546 <+0>:     push   rbp
   0x0000000000400547 <+1>:     mov    rbp,rsp
   0x000000000040054a <+4>:     sub    rsp,0xa0
   0x0000000000400551 <+11>:    mov    DWORD PTR [rbp-0x94],edi
   0x0000000000400557 <+17>:    mov    DWORD PTR [rbp-0x98],esi
   0x000000000040055d <+23>:    mov    DWORD PTR [rbp-0x9c],edx
   0x0000000000400563 <+29>:    mov    rax,QWORD PTR fs:0x28
   0x000000000040056c <+38>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000400570 <+42>:    xor    eax,eax
   0x0000000000400572 <+44>:    nop
   0x0000000000400573 <+45>:    mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000400577 <+49>:    xor    rax,QWORD PTR fs:0x28
   0x0000000000400580 <+58>:    je     0x400587 <foo+65>
   0x0000000000400582 <+60>:    call   0x400420 <__stack_chk_fail@plt>
   0x0000000000400587 <+65>:    leave
   0x0000000000400588 <+66>:    ret
End of assembler dump.
+pwndbg>
```

Saving main's frame pointer



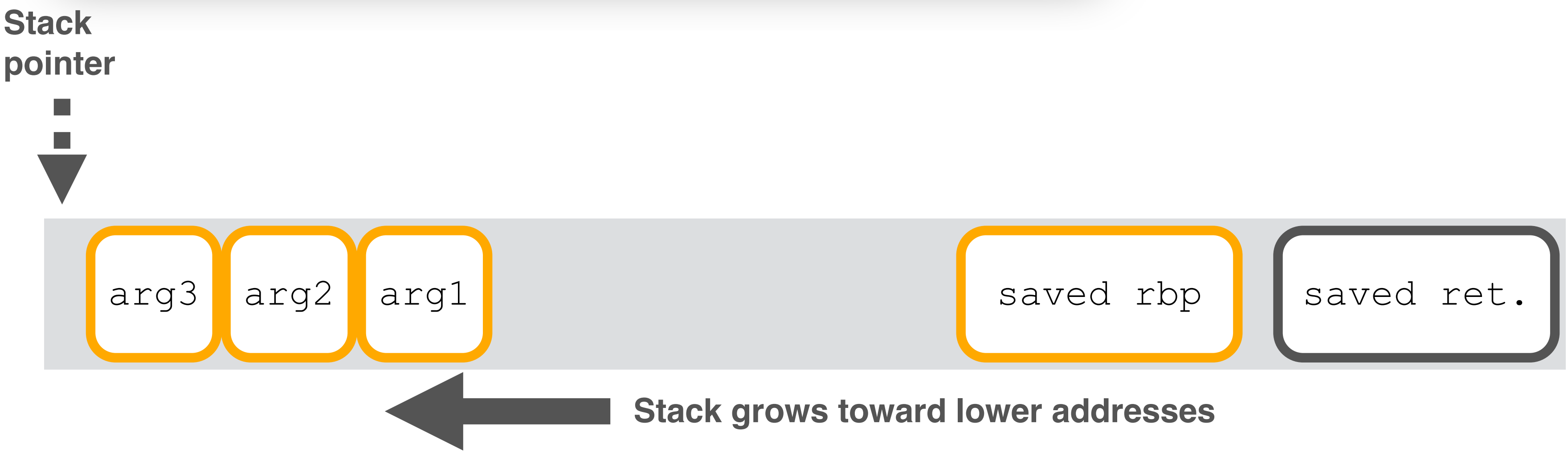
```
+pwndbg> disass *foo
Dump of assembler code for function foo:
   0x0000000000400546 <+0>:     push   rbp
   0x0000000000400547 <+1>:     mov    rbp,rsp
   0x000000000040054a <+4>:     sub    rsp,0xa0
   0x0000000000400551 <+11>:    mov    DWORD PTR [rbp-0x94],edi
   0x0000000000400557 <+17>:    mov    DWORD PTR [rbp-0x98],esi
   0x000000000040055d <+23>:    mov    DWORD PTR [rbp-0x9c],edx
   0x0000000000400563 <+29>:    mov    rax,QWORD PTR fs:0x28
   0x000000000040056c <+38>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000400570 <+42>:    xor    eax,eax
   0x0000000000400572 <+44>:    nop
   0x0000000000400573 <+45>:    mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000400577 <+49>:    xor    rax,QWORD PTR fs:0x28
   0x0000000000400580 <+58>:    je     0x400587 <foo+65>
   0x0000000000400582 <+60>:    call   0x400420 <__stack_chk_fail@plt>
   0x0000000000400587 <+65>:    leave
   0x0000000000400588 <+66>:    ret
End of assembler dump.
+pwndbg>
```

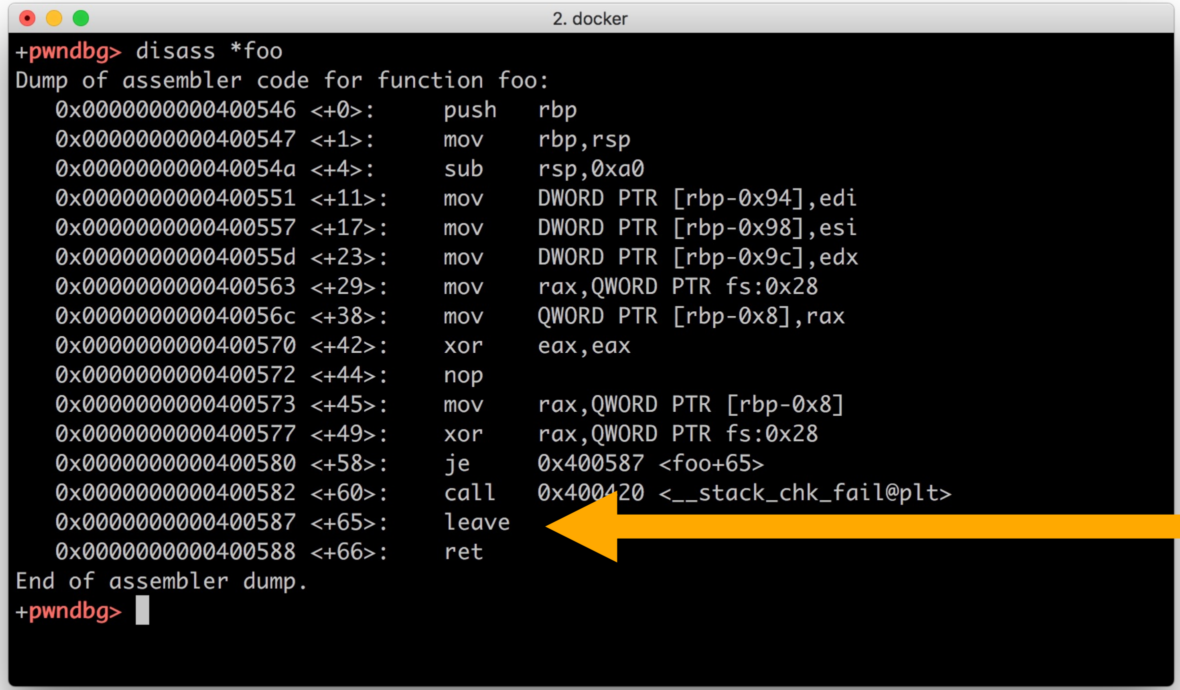
Allocating memory on the stack for args, and local vars



```
2. docker
+pwndbg> disass *foo
Dump of assembler code for function foo:
   0x0000000000400546 <+0>:     push   rbp
   0x0000000000400547 <+1>:     mov    rbp,rsp
   0x000000000040054a <+4>:     sub    rsp,0xa0
   0x0000000000400551 <+11>:    mov    DWORD PTR [rbp-0x94],edi
   0x0000000000400557 <+17>:    mov    DWORD PTR [rbp-0x98],esi
   0x000000000040055d <+23>:    mov    DWORD PTR [rbp-0x9c],edx
   0x0000000000400563 <+29>:    mov    rax,QWORD PTR fs:0x28
   0x000000000040056c <+38>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000400570 <+42>:    xor    eax,eax
   0x0000000000400572 <+44>:    nop
   0x0000000000400573 <+45>:    mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000400577 <+49>:    xor    rax,QWORD PTR fs:0x28
   0x0000000000400580 <+58>:    je     0x400587 <foo+65>
   0x0000000000400582 <+60>:    call   0x400420 <__stack_chk_fail@plt>
   0x0000000000400587 <+65>:    leave
   0x0000000000400588 <+66>:    ret
End of assembler dump.
+pwndbg>
```

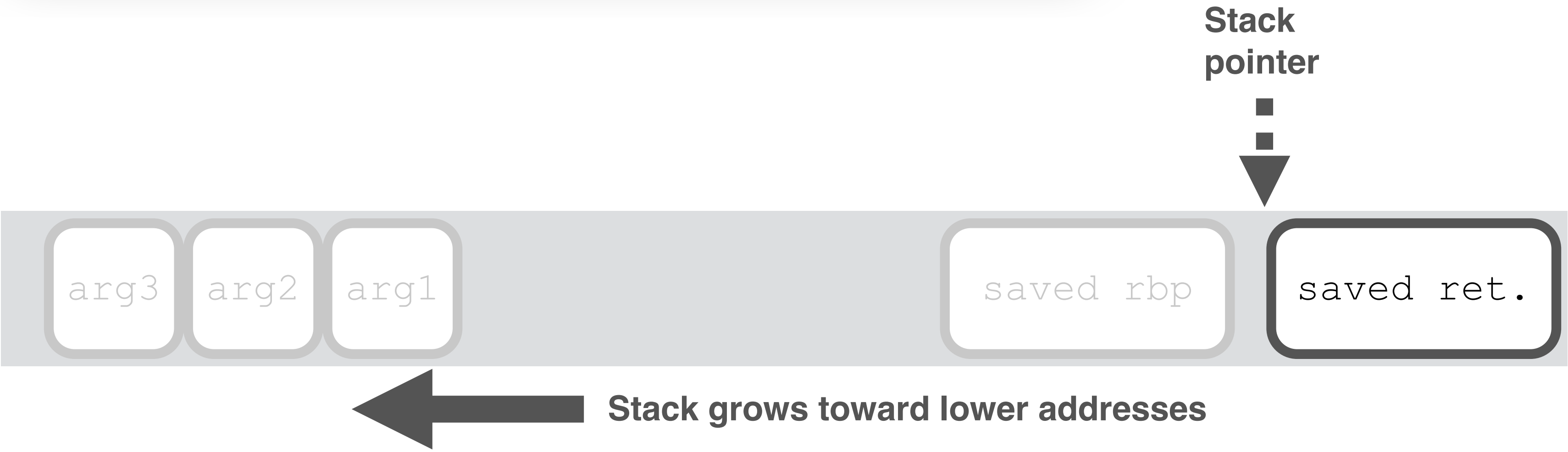
Moving args onto the stack

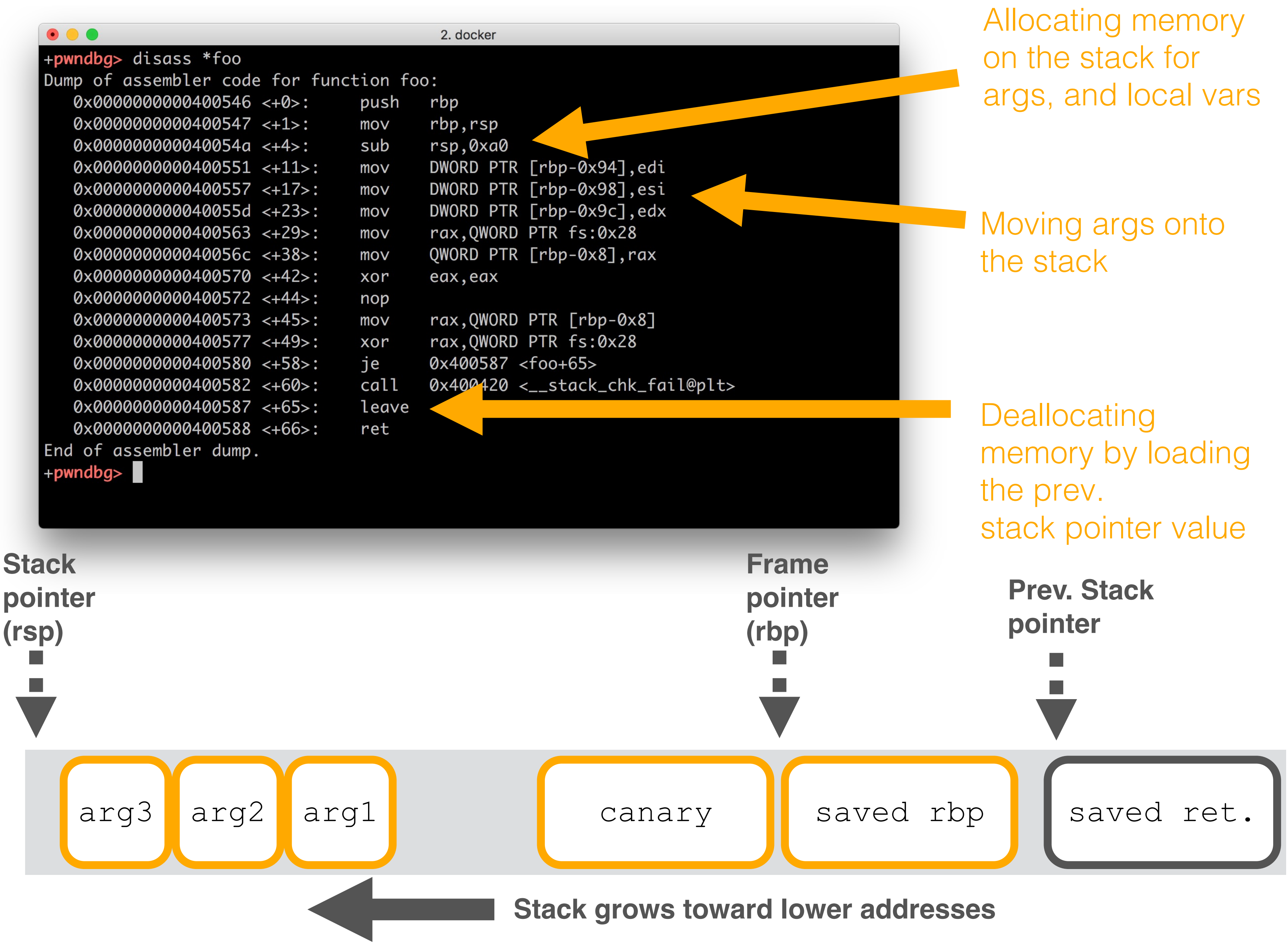




```
+pwndbg> disass *foo
Dump of assembler code for function foo:
   0x0000000000400546 <+0>:     push   rbp
   0x0000000000400547 <+1>:     mov    rbp,rsp
   0x000000000040054a <+4>:     sub    rsp,0xa0
   0x0000000000400551 <+11>:    mov    DWORD PTR [rbp-0x94],edi
   0x0000000000400557 <+17>:    mov    DWORD PTR [rbp-0x98],esi
   0x000000000040055d <+23>:    mov    DWORD PTR [rbp-0x9c],edx
   0x0000000000400563 <+29>:    mov    rax,QWORD PTR fs:0x28
   0x000000000040056c <+38>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000400570 <+42>:    xor    eax,eax
   0x0000000000400572 <+44>:    nop
   0x0000000000400573 <+45>:    mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000400577 <+49>:    xor    rax,QWORD PTR fs:0x28
   0x0000000000400580 <+58>:    je     0x400587 <foo+65>
   0x0000000000400582 <+60>:    call   0x400420 <__stack_chk_fail@plt>
   0x0000000000400587 <+65>:    leave
   0x0000000000400588 <+66>:    ret
End of assembler dump.
+pwndbg>
```

Deallocating memory by loading the prev. stack pointer value




2. docker
+pwndbg> disass *foo
Dump of assembler code for function foo:
0x0000000000400546 <+0>: push rbp
0x0000000000400547 <+1>: mov rbp,rsp
0x000000000040054a <+4>: sub rsp,0xa0
0x0000000000400551 <+11>: mov DWORD PTR [rbp-0x94],edi
0x0000000000400557 <+17>: mov DWORD PTR [rbp-0x98],esi
0x000000000040055d <+23>: mov DWORD PTR [rbp-0x9c],edx
0x0000000000400563 <+29>: mov rax,QWORD PTR fs:0x28
0x000000000040056c <+38>: mov QWORD PTR [rbp-0x8],rax
0x0000000000400570 <+42>: xor eax,eax
0x0000000000400572 <+44>: nop
0x0000000000400573 <+45>: mov rax,QWORD PTR [rbp-0x8]
0x0000000000400577 <+49>: xor rax,QWORD PTR fs:0x28
0x0000000000400580 <+58>: je 0x400587 <foo+65>
0x0000000000400582 <+60>: call 0x400420 <__stack_chk_fail@plt>
0x0000000000400587 <+65>: leave
0x0000000000400588 <+66>: ret
End of assembler dump.
+pwndbg>
Allocating memory on the stack for args, and local vars
Moving args onto the stack
Deallocating memory by loading the prev. stack pointer value
Stack pointer (rsp)
Frame pointer (rbp)
Prev. Stack pointer
arg3
arg2
arg1
canary
saved rbp
saved ret.
Stack grows toward lower addresses

# The heap

> The **heap** is used for user-managed, dynamically allocated memory. One common interface to the heap is Libc's `malloc()` and `free()` functions.

> `malloc(size)` allocates a size number of bytes from the heap and returns a void pointer to those bytes.

- `int *x = (int *) malloc(sizeof(int));`

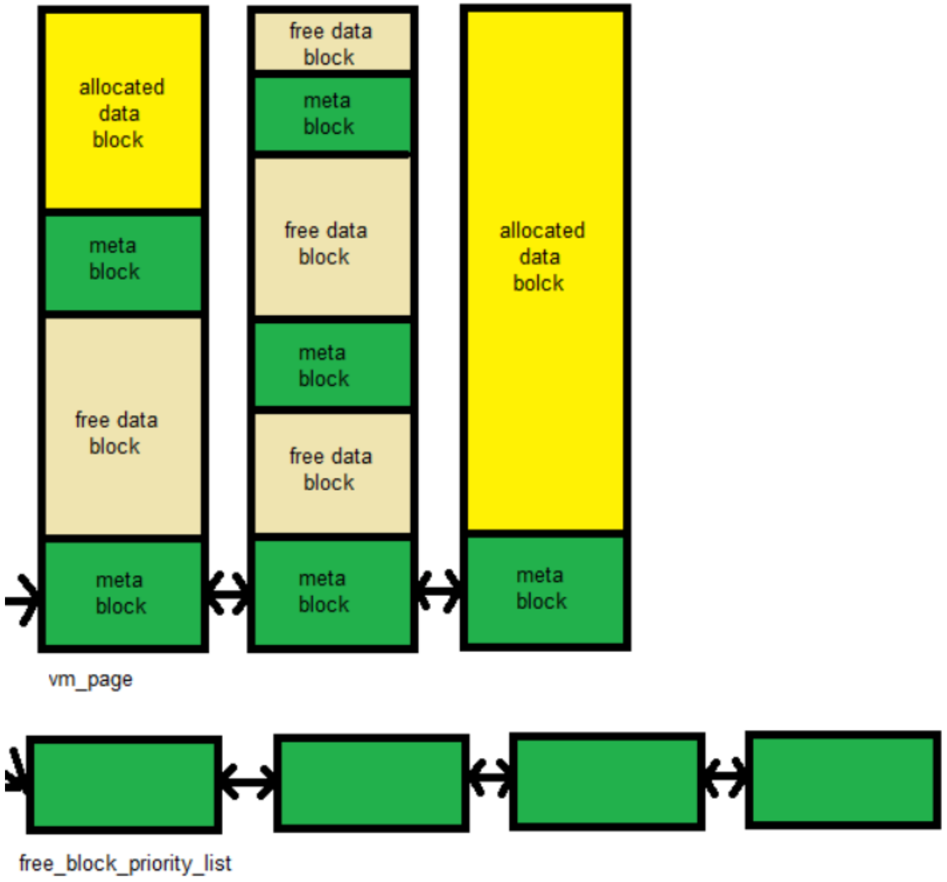
> `free()` takes a pointer to some previously allocated heap memory and deallocates that memory, making available for future use.

- `free(x);`

> Under the hood, these functions using system calls (e.g., `sbrk`) to request memory from the OS.

# How does the heap internally work?

- In general, functions such as malloc will request a bunch of memory from the OS, and then each call to allocate memory will reserve a block within this memory
- Thus, the heap must be explicitly managed using a dedicated data structure




by Debarshi Maitra

*https://github.com/artiam99/Linux-Heap-Memory-Manager*

# Heap usage example

- Calls to **malloc()** results in **new blocks being allocated** in the area managed as the heap

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
int* array = (int*)malloc(sizeof(int)*10);

array[0] = 24;
array[9] = 42;

printf("Location of array pointer: %p\n", &array);
printf("Location pointed by pointer: %p\n", array);
printf("Location of first element of array: %p\n",
&(array[0]));
printf("Content of first element of array: %d\n", array[0]);
printf("Location of last element of array: %p\n", &(array[9]));
printf("Content of last element of array: %d\n", array[9]);
}
```

# What comes next?

> We are going to discuss memory exploits **with increasing level of complexity:**
  - **Stack overflow** (next lecture)
  - **Heap exploits** (two lectures from now)
  - **Return-oriented** programming (three lectures from now)

See you next week!