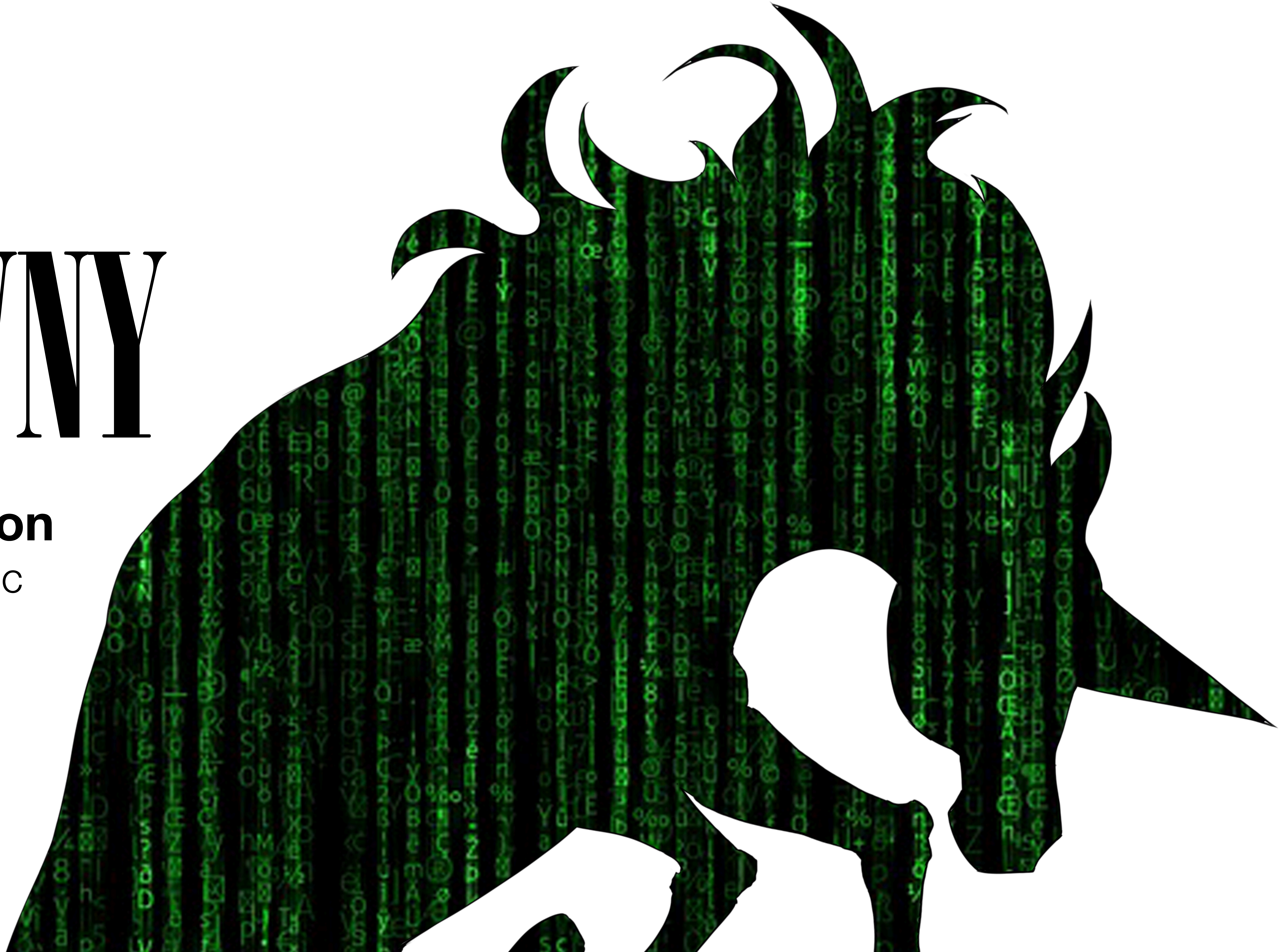
SIGPWNY

Heap Exploitation

Part 1- Intro to Malloc

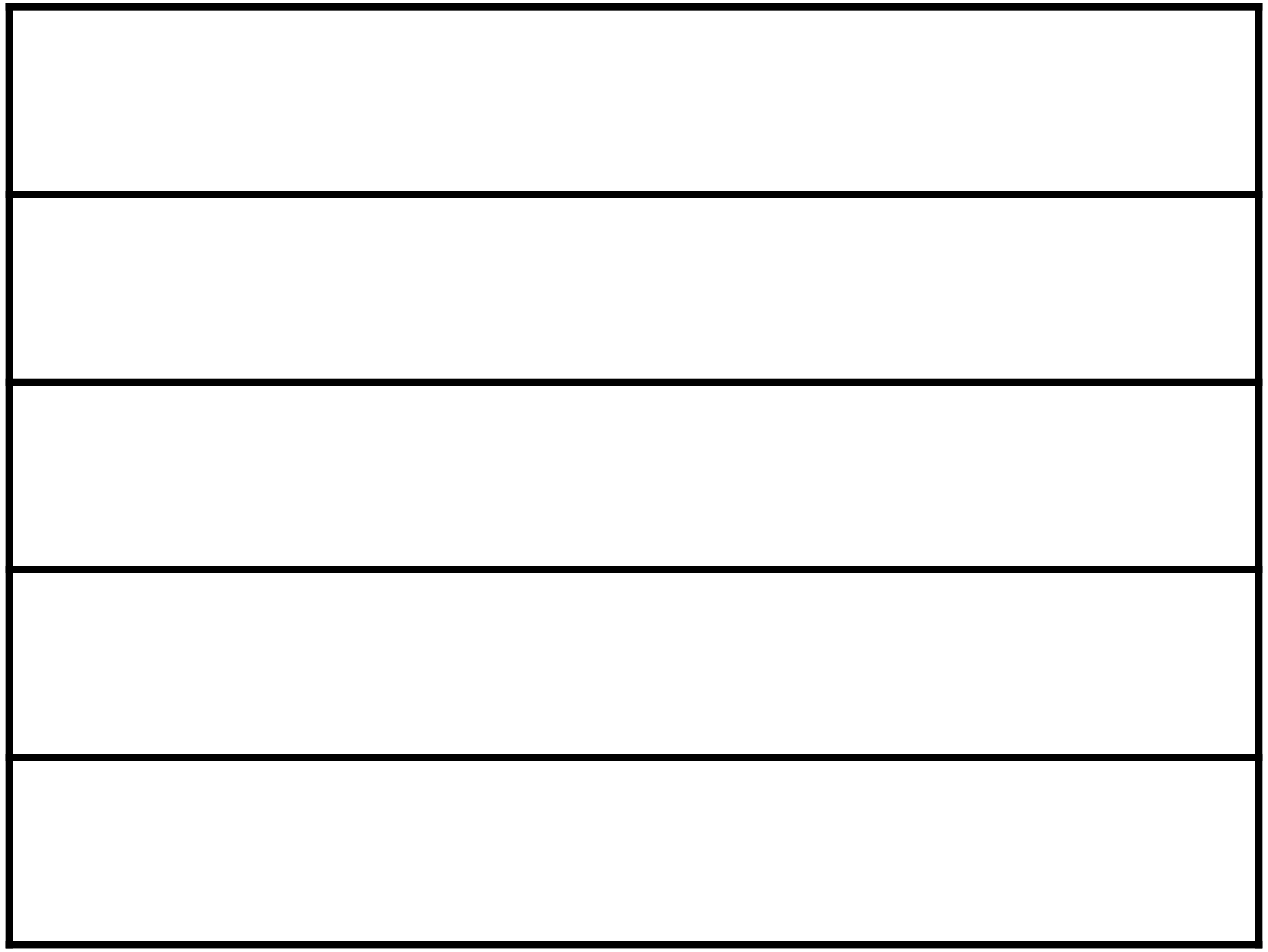


Recap

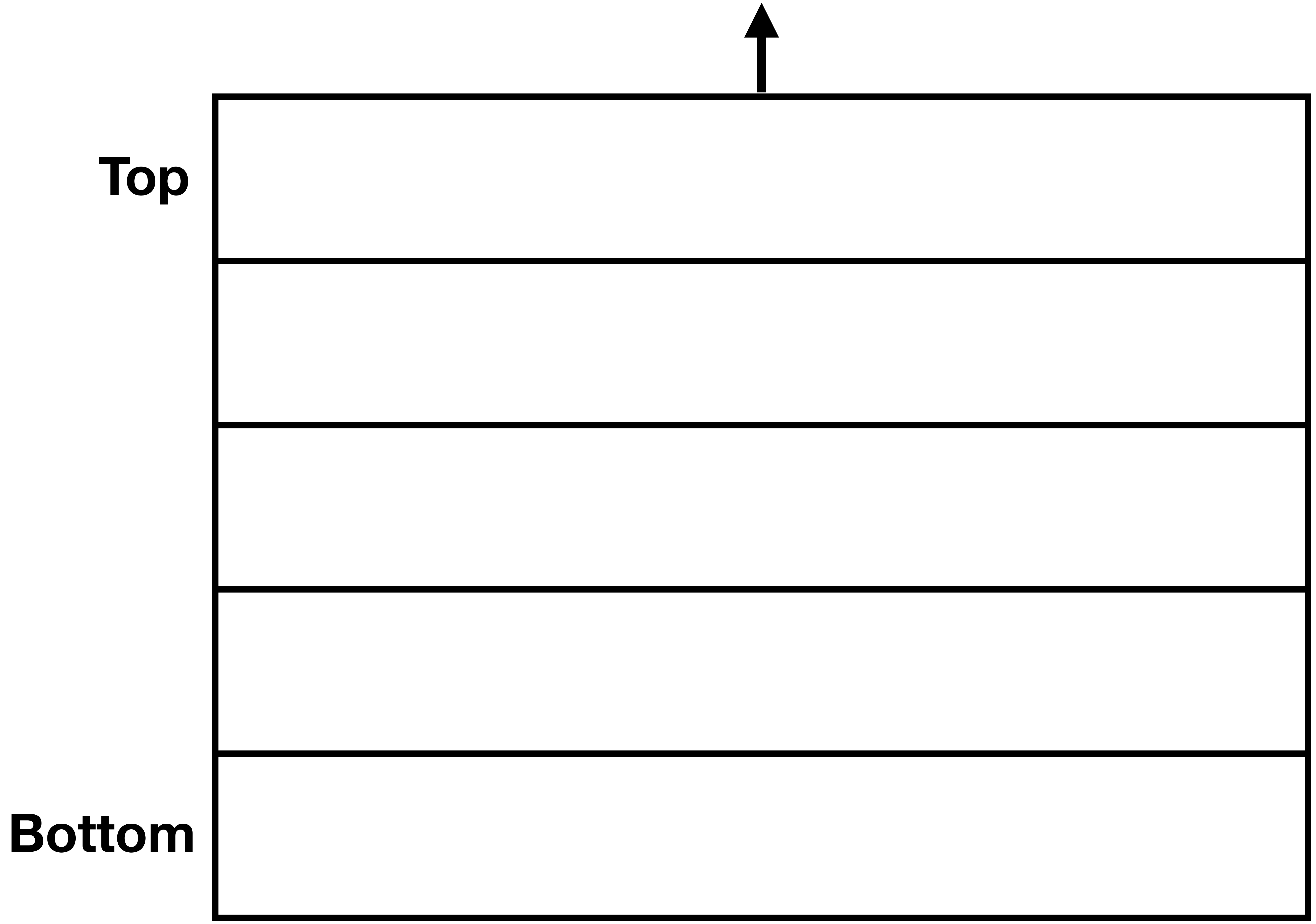
Stack and the Global Offset Table

All programs have a stack.

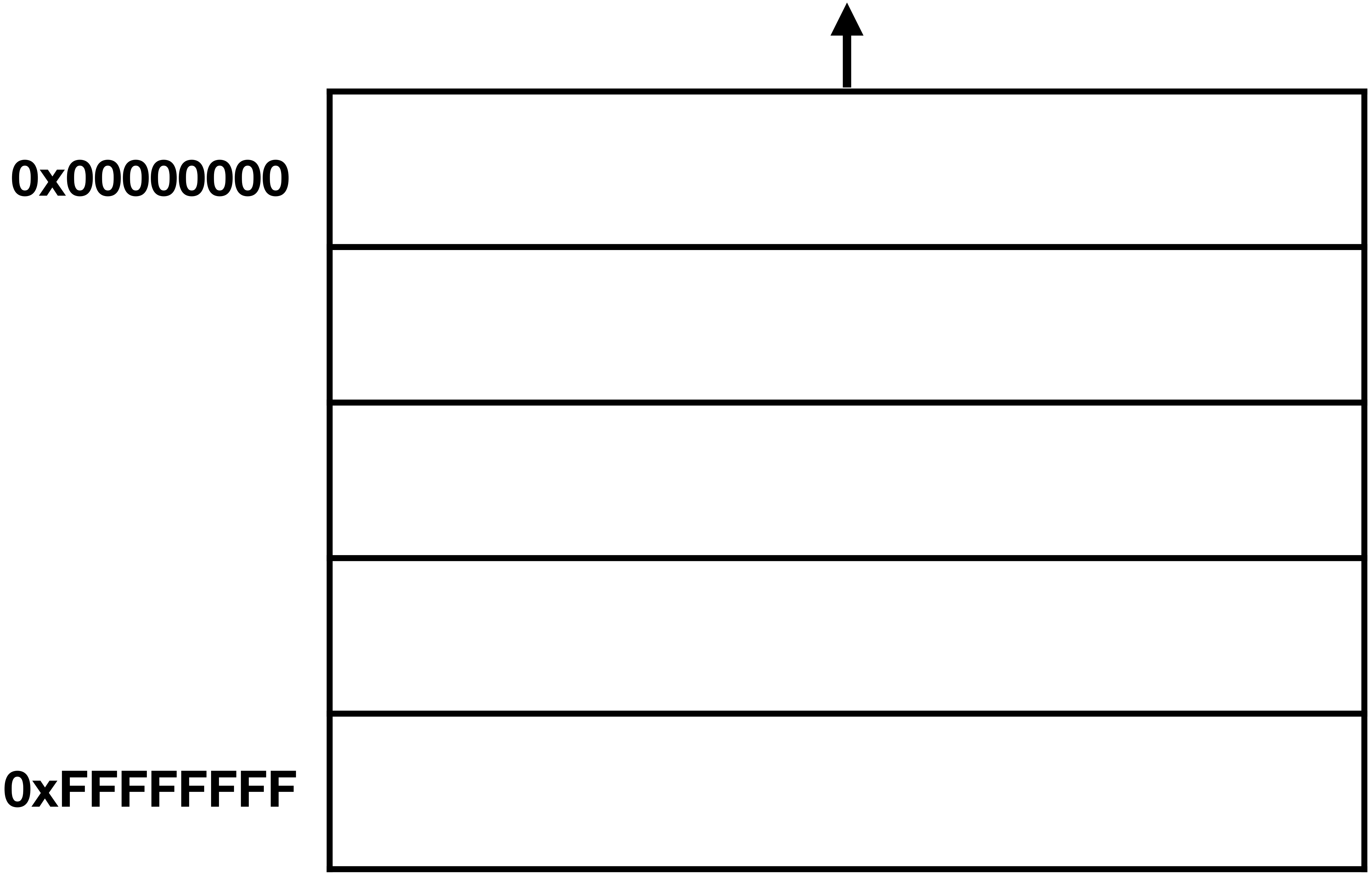
All programs have a stack.



All programs have a stack.



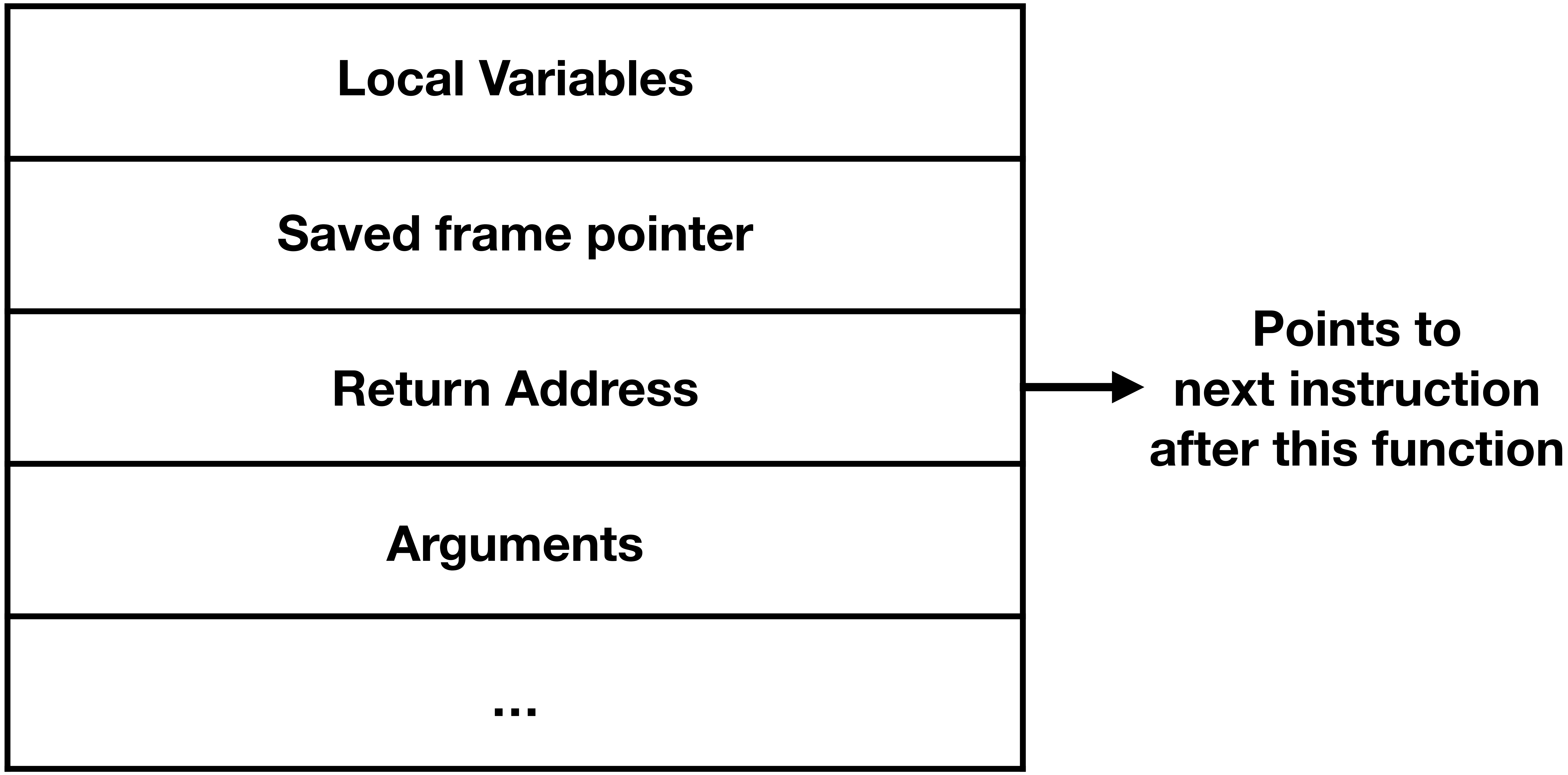
All programs have a stack.



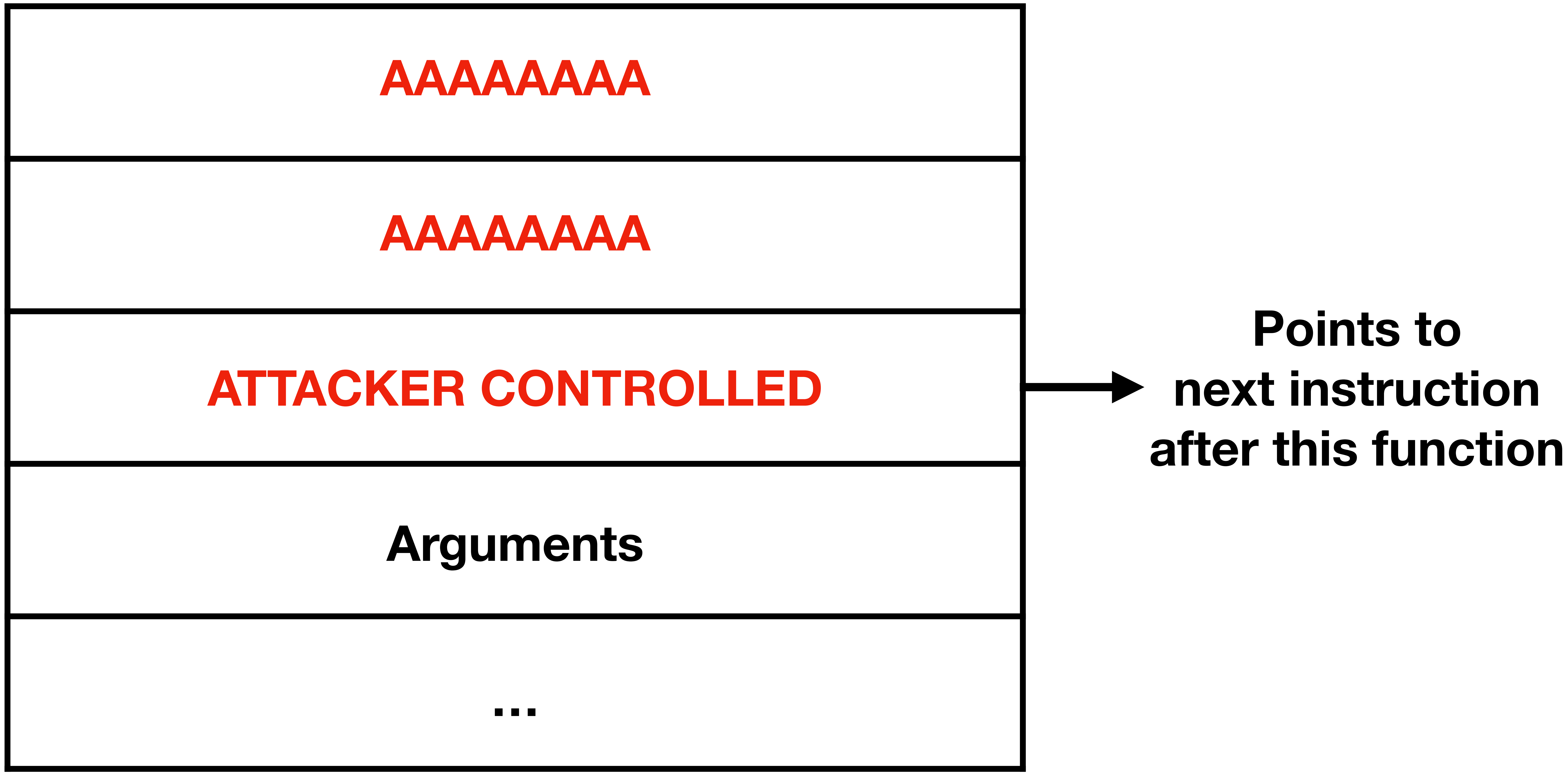
What goes in the stack?

Local Variables
Saved frame pointer
Return Address
Arguments
...

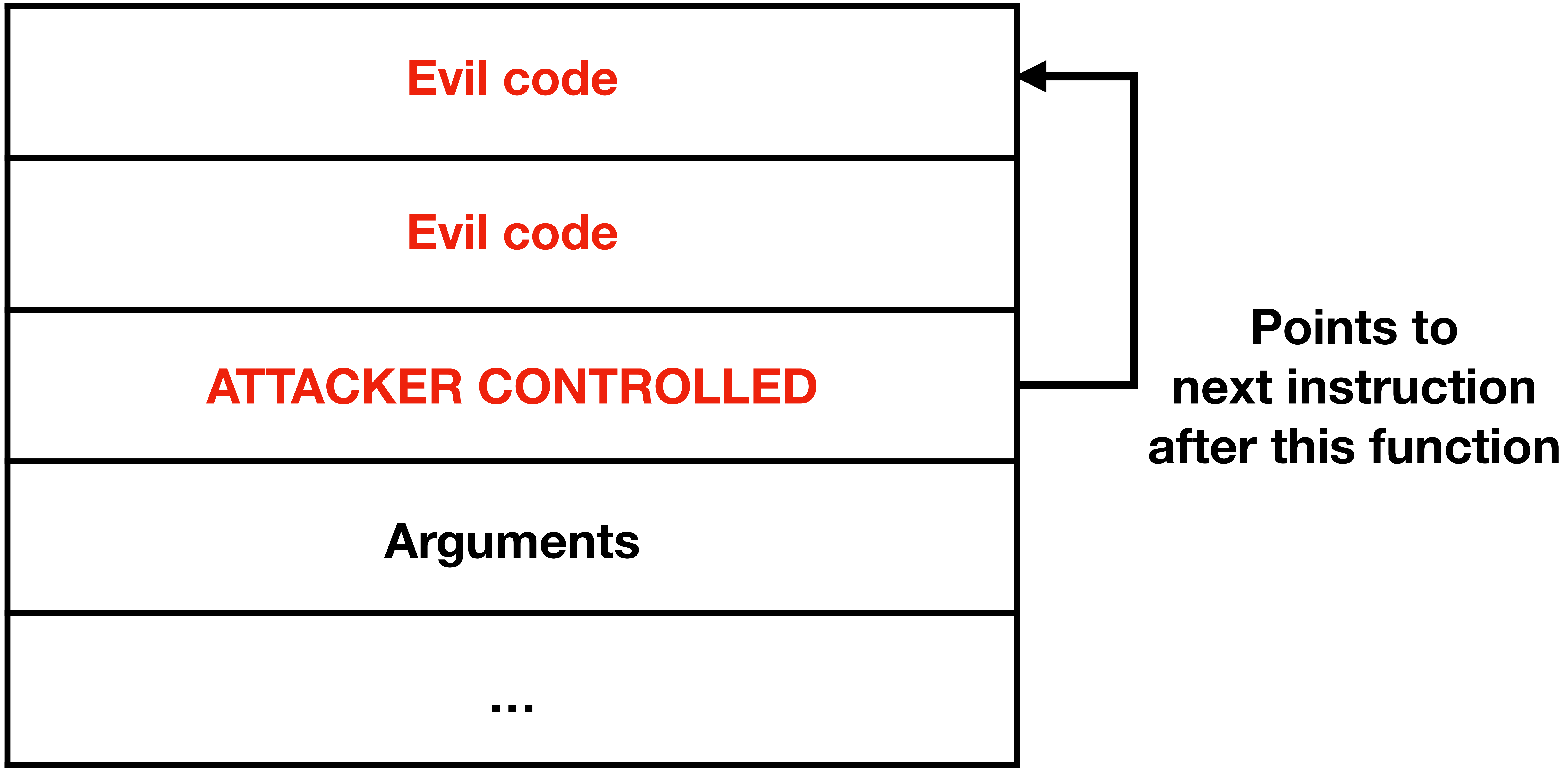
What goes in the stack?



We can overflow the stack



We can overflow the stack



Dynamic Linking

The Global Offset Table

libc

Pronounced “Lib See”

```
⌥⌘1                                /dev/ttys000
     0x401244 <main+32>        call   0x401050 <printf@plt>
                                                                                              threads
[#0] Id 1, Name: "external", stopped 0x401228 in main (), reason: BREAKPOINT
                                                                                                trace
[#0] 0x401228 → main()

gef➤  info proc map
process 20
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /pwn/external
            0x401000           0x402000     0x1000     0x1000 /pwn/external
            0x402000           0x403000     0x1000     0x2000 /pwn/external
            0x403000           0x404000     0x1000     0x2000 /pwn/external
            0x404000           0x405000     0x1000     0x3000 /pwn/external
      0x7ffff7dd0000     0x7ffff7df5000    0x25000        0x0 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7df5000     0x7ffff7f6d000   0x178000    0x25000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7f6d000     0x7ffff7fb7000    0x4a000   0x19d000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7fb7000     0x7ffff7fb8000     0x1000   0x1e7000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7fb8000     0x7ffff7fbb000     0x3000   0x1e7000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7fbb000     0x7ffff7fbe000     0x3000   0x1ea000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7fbe000     0x7ffff7fc4000     0x6000        0x0
      0x7ffff7fca000     0x7ffff7fcd000     0x3000        0x0 [vvar]
      0x7ffff7fcd000     0x7ffff7fcf000     0x2000        0x0 [vdso]
      0x7ffff7fcf000     0x7ffff7fd0000     0x1000        0x0 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7fd0000     0x7ffff7ff3000    0x23000     0x1000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ff3000     0x7ffff7ffb000     0x8000    0x24000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x2c000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x2d000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
gef➤
[0] 0:gdb*                                                       "docker-desktop" 02:58 27-Jan-21
```

libc

Pronounced “Lib See”

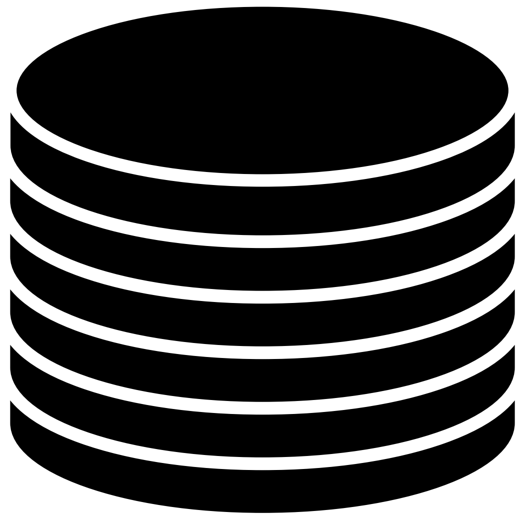
```
⌥⌘1                                /dev/ttys000
     0x401244 <main+32>        call   0x401050 <printf@plt>
                                                                         threads
[#0] Id 1, Name: "external", stopped 0x401228 in main (), reason: BREAKPOINT
                                                                           trace
[#0] 0x401228 → main()

gef➤  info proc map
process 20
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /pwn/external
            0x401000           0x402000     0x1000     0x1000 /pwn/external
      0x7ffff7dd0000     0x7ffff7df5000    0x25000        0x0 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7df5000     0x7ffff7f6d000   0x178000    0x25000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7f6d000     0x7ffff7fb7000    0x4a000   0x19d000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7fb7000     0x7ffff7fb8000     0x1000   0x1e7000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7fb8000     0x7ffff7fbb000     0x3000   0x1e7000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7fbb000     0x7ffff7fbe000     0x3000   0x1ea000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7ffff7fcf000     0x7ffff7fd0000     0x1000        0x0 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7fd0000     0x7ffff7ff3000    0x23000     0x1000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ff3000     0x7ffff7ffb000     0x8000    0x24000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x2c000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x2d000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
gef➤
[0] 0:gdb*                                        "docker-desktop" 02:58 27-Jan-21
```

How do programs know where libc functions are?

Global Offset Table

Mapping between name & address

Name	Address
printf	“0x41414141”
exit	“0x42424242”
puts	“0x43434343"
memset	“0x44444444"



The Heap

libc Dynamic Memory Allocation

Dynamic Memory

`malloc(number of bytes)`

returns a pointer

`free(pointer)`

cleans up the pointer

Dynamic Memory

⌥⌘1 /dev/ttys000

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

int main () {
        int stack_var;
        int *stack_pointer;
        int *heap_pointer;

        // Point stack_pointer to the stack variable
        stack_pointer = &stack_var;

        // Allocate a heap variable and store pointer to it
        // in heap_pointer
        heap_pointer = malloc(sizeof(int));

        printf("stack_pointer points to %p\n", stack_pointer);
        printf("heap_pointer points to %p\n", heap_pointer);

        // Never forget to clean up memory when done!
        free(heap_pointer);
}
```

Dynamic Memory

⌥⌘1 /dev/ttys000

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

int main () {
        int stack_var;
        int *stack_pointer;
        int *heap_pointer;

        // Point stack_pointer to the stack variable
        stack_pointer = &stack_var;

        // Allocate a heap variable and store pointer to it
        // i
pointer
        heap
= malloc(sizeof(int));

        printf("stack_pointer points to %p\n", stack_pointer);
        printf("heap_pointer points to %p\n", heap_pointer);

        // Never forget to clean up memory when done!
        free(heap_pointer);
}
```

Dynamic Memory

⌥⌘1 /dev/ttys000

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

int main () {
        int stack_var;
        int *stack_pointer;
        int *heap_pointer;

        // Point stack_pointer to the stack variable
        stack_pointer = &stack_var;

        // Allocate a heap variable and store pointer to it
        // in heap_pointer
        heap_pointer = malloc(sizeof(int));

        printf("stack_pointer points to %p\n", stack_pointer);
        printf("heap_pointer points to %p\n", heap_pointer);

                                        when done!

}
```

```c
free(heap_pointer);
```

Dynamic Memory

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

int main () {
	int stack_var;
	int *stack_pointer;
	int *heap_pointer;

	// Point stack_pointer to the stack variable
	stack_pointer = &stack_var;

	// Allocate a heap variable and store pointer to it
	// in heap_pointer
	heap_pointer = malloc(sizeof(int));

	printf("stack_pointer points to %p\n", stack_pointer);
	printf("heap_pointer points to %p\n", heap_pointer);

	// Never forget to clean up memory when done!
	free(heap_pointer);
}
```

```
root@docker-desktop:/pwn# ./demo1
stack_pointer points to 0x7ffd021cd084
heap_pointer points to 0x55d45fad82a0
root@docker-desktop:/pwn#
```

Ways to screw up dynamic memory

Not freeing memory

Leads to memory leaks.

(not the information disclosure kind of leak)

Not necessarily exploitable, might be able to crash a program.

Using memory after freeing it

Leads to “Use after Free” (UaF)

Pretty good chance that this is exploitable.

Ways to screw up dynamic memory

Freeing something more than once

Corrupts internal heap data structures.

Can lead to exploits (see Heap Challenge 3).

Main Goal

Get attacker controlled data where it shouldn’t be.

Get attacker controlled data where it shouldn't be.

1. Corrupt objects the program is still using ("Use after Free")
2. Set the heap up such that future allocations return attacker controlled data ("heap spray", "double free", "malloc maleficarum" attacks), or point to structures of interest ("unlink macro" exploit)
3. Overflow heap objects by corrupting size or using buffer overflows (see "Heap Challenge 4")
4. Get creative

Heap can be weird.

Don’t try to memorize all attacks,
develop intuition for why they work.

When glibc patches these attacks, or you’re working on a different allocator, general intuition > memorization.

Next heap meeting:
How malloc internals work.

Recommended Reading

glibc malloc source code: https://github.com/bminor/glibc/blob/master/malloc/malloc.c

Once Upon a Free(): http://phrack.org/issues/57/9.html

Malloc Internals: https://sourceware.org/glibc/wiki/MallocInternals

LiveOverflow Binexp series: https://www.youtube.com/watch?v=HPDBOhiKaD8

Shellphish how2heap: https://github.com/shellphish/how2heap

Heap 1

Your first UaF exploit!