Buffer Overflow

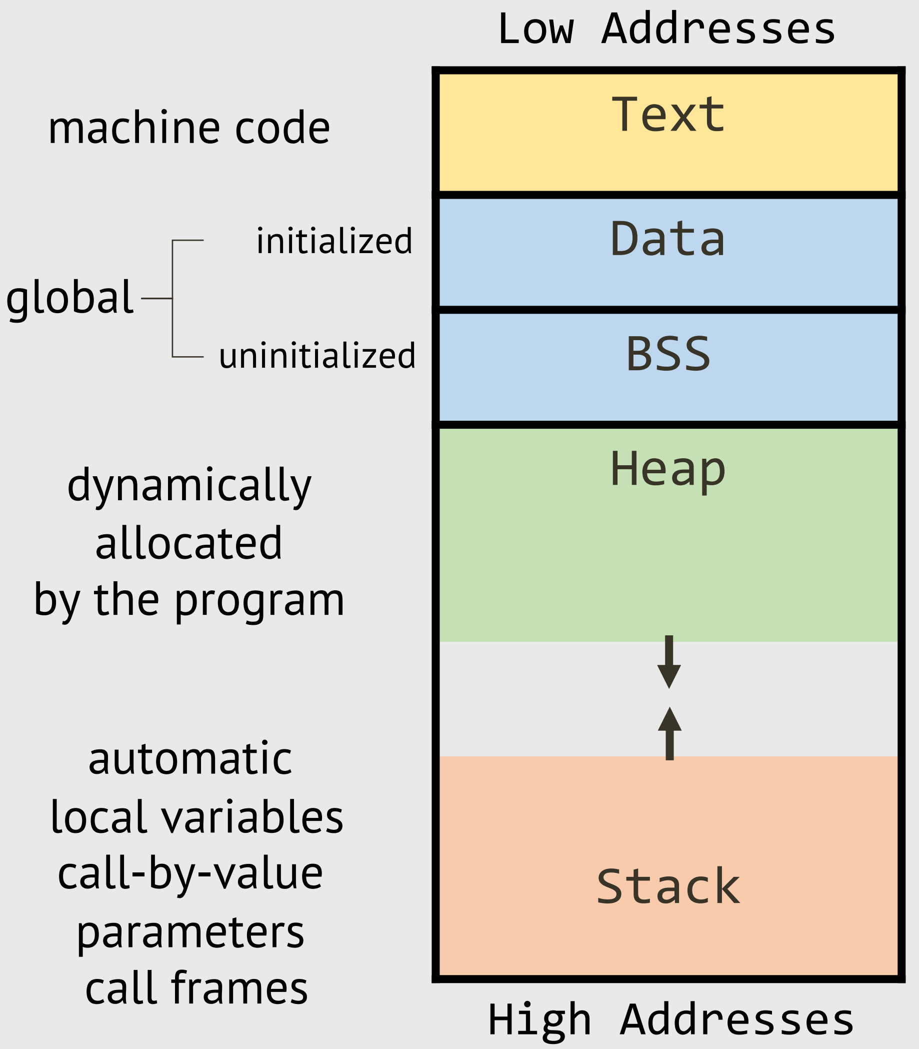
Buffer Overflows

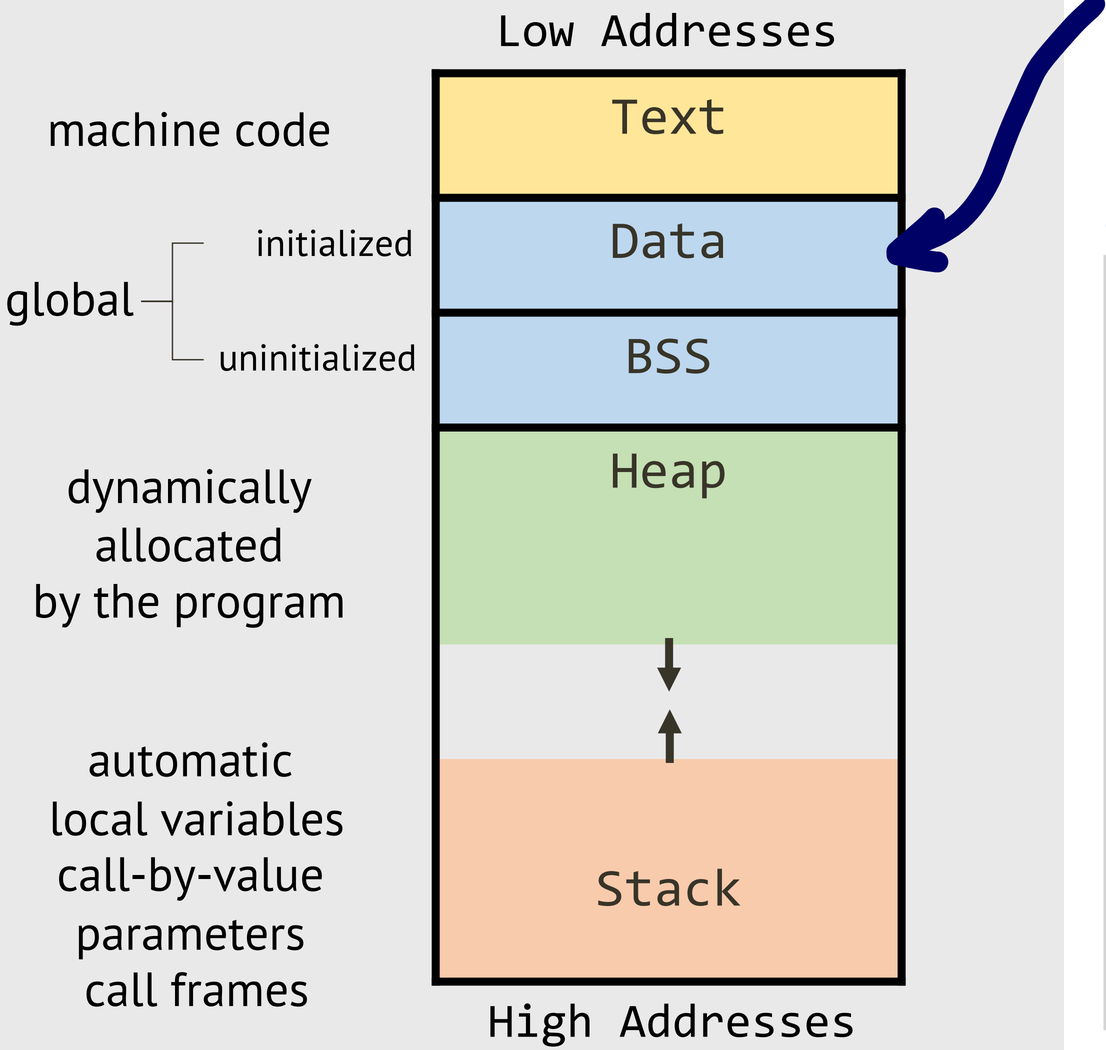
1. How they work
2. Countermeasures
3. Shellcode

Buffer Overflows

1. How they work
2. Countermeasures
3. Shellcode

C/C++ Program Memory Layout


Low Addresses
machine code
Text
global
initialized
Data
uninitialized
BSS
dynamically allocated by the program
Heap
automatic local variables call-by-value parameters call frames
Stack
High Addresses



Low Addresses

machine code — Text

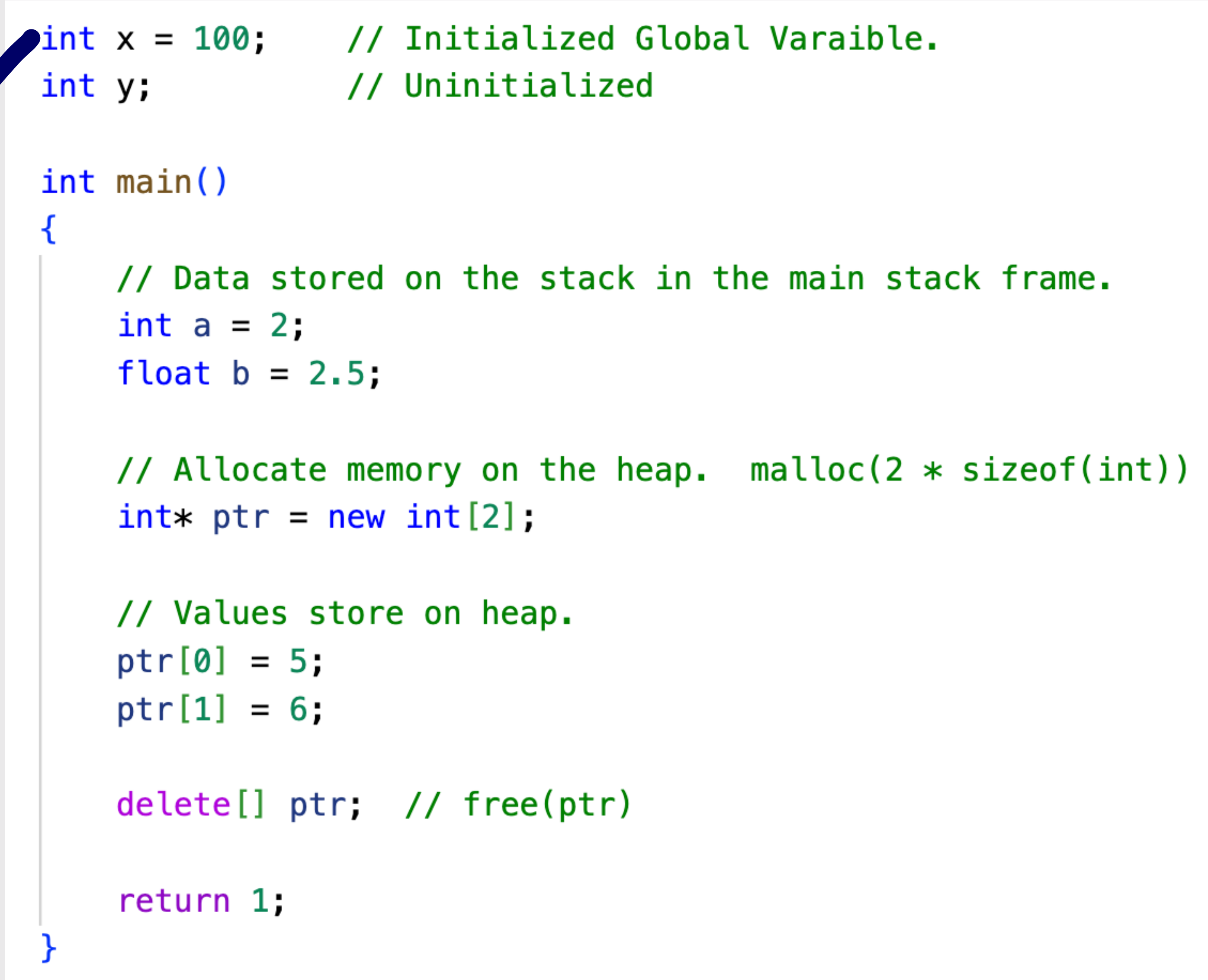
global — initialized — Data

global — uninitialized — BSS

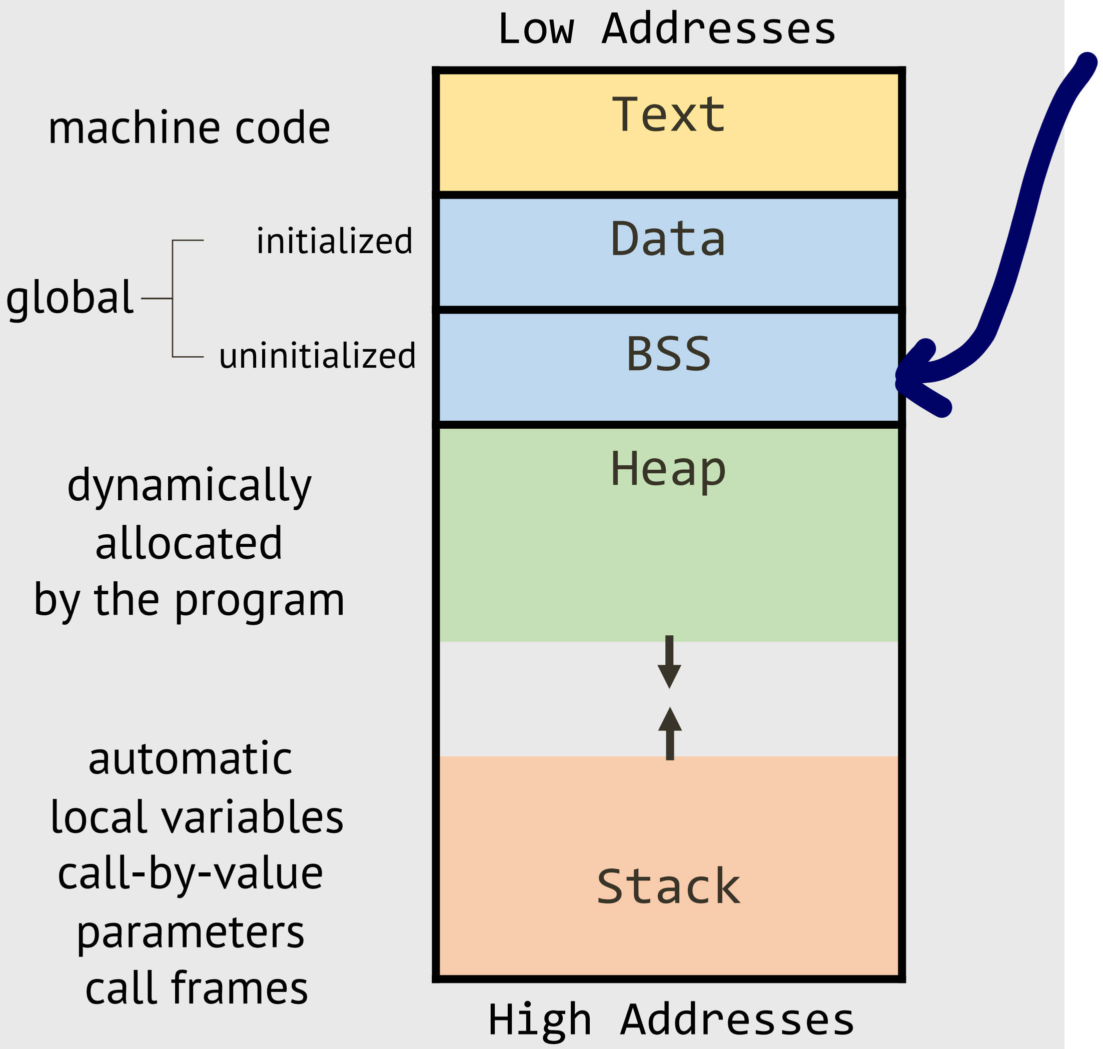
dynamically allocated by the program — Heap

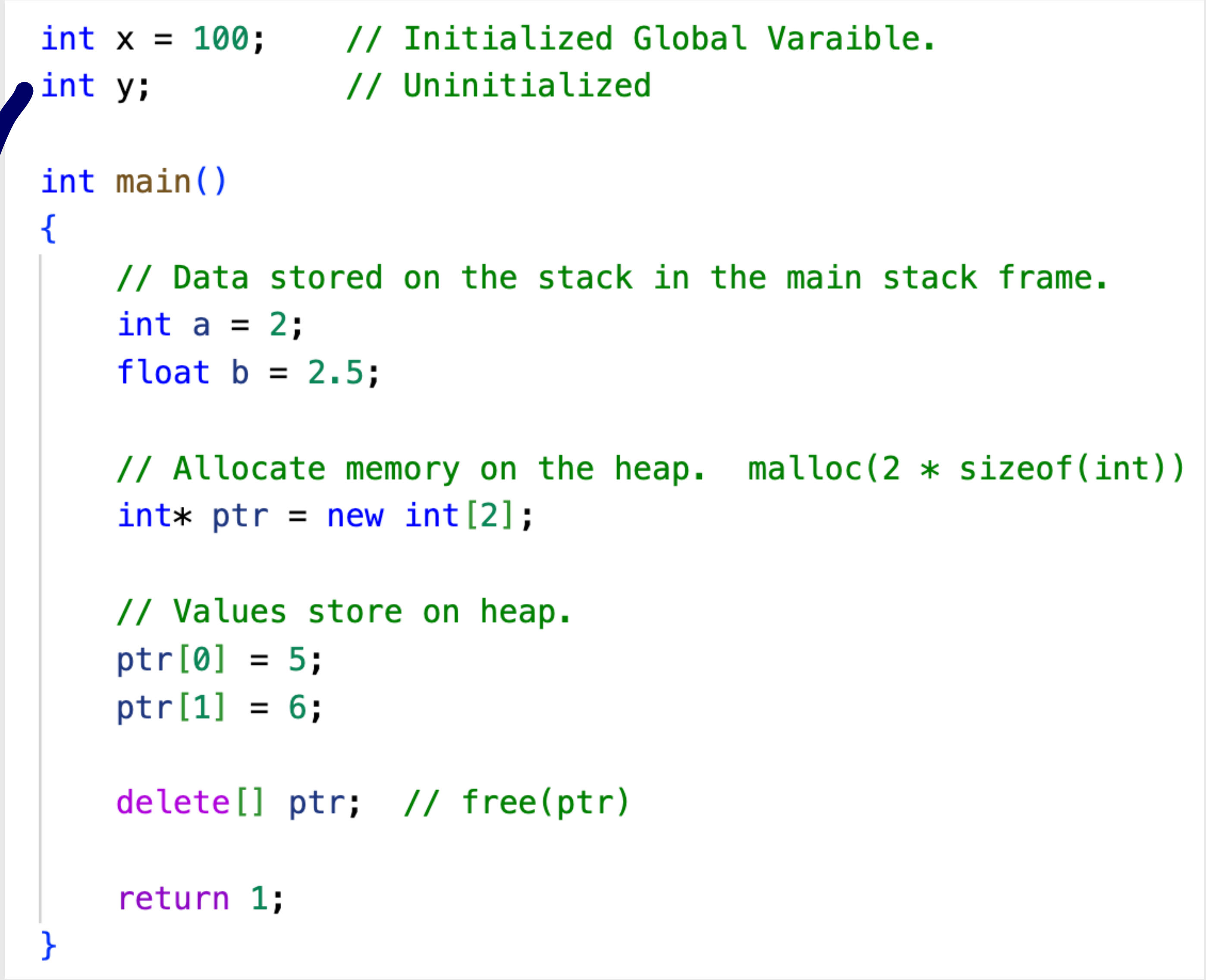
automatic local variables call-by-value parameters call frames — Stack

High Addresses

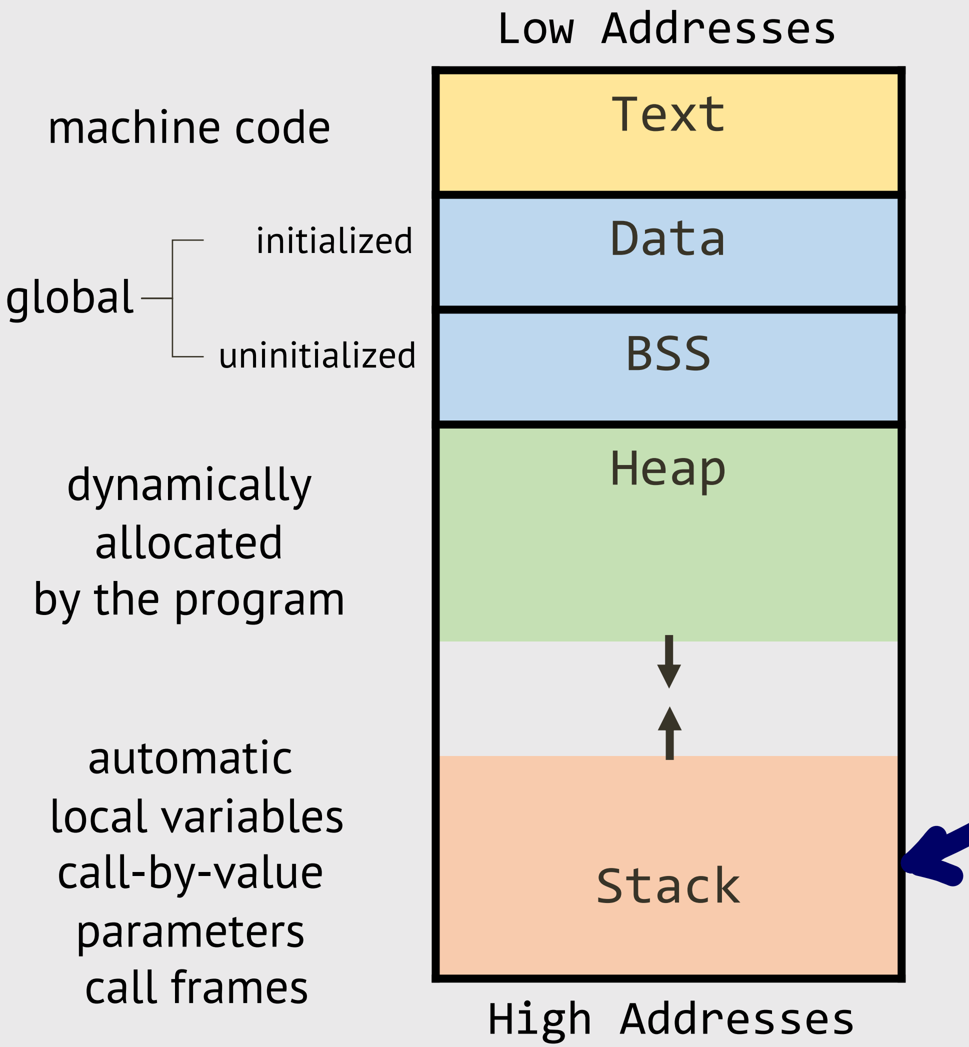


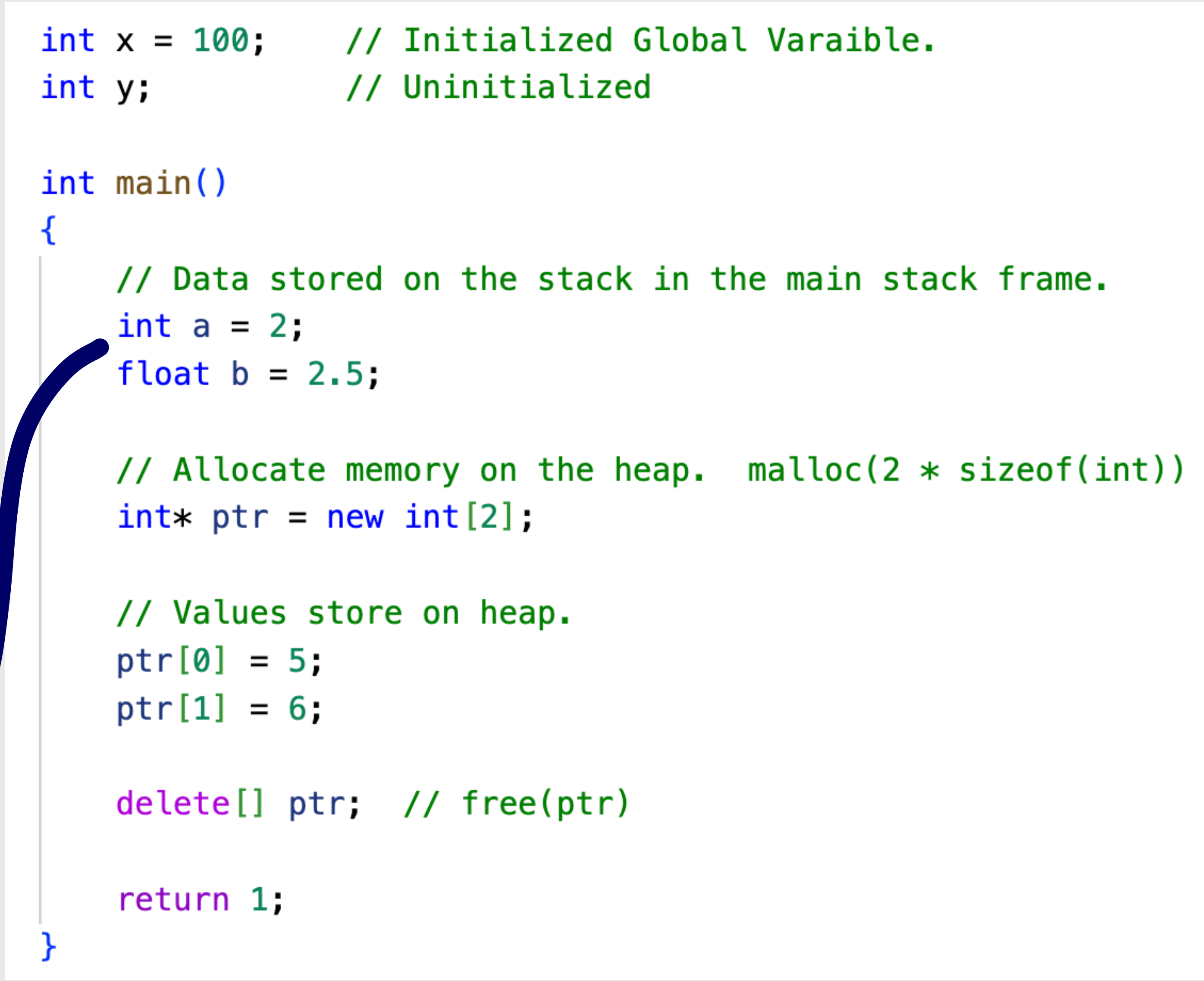
```
int x = 100;    // Initialized Global Varaible.
int y;          // Uninitialized

int main()
{
    // Data stored on the stack in the main stack frame.
    int a = 2;
    float b = 2.5;

    // Allocate memory on the heap.  malloc(2 * sizeof(int))
    int* ptr = new int[2];

    // Values store on heap.
    ptr[0] = 5;
    ptr[1] = 6;

    delete[] ptr;  // free(ptr)

    return 1;
}
```


Low Addresses
machine code
Text
global
initialized
Data
uninitialized
BSS
dynamically allocated by the program
Heap
automatic local variables call-by-value parameters call frames
Stack
High Addresses

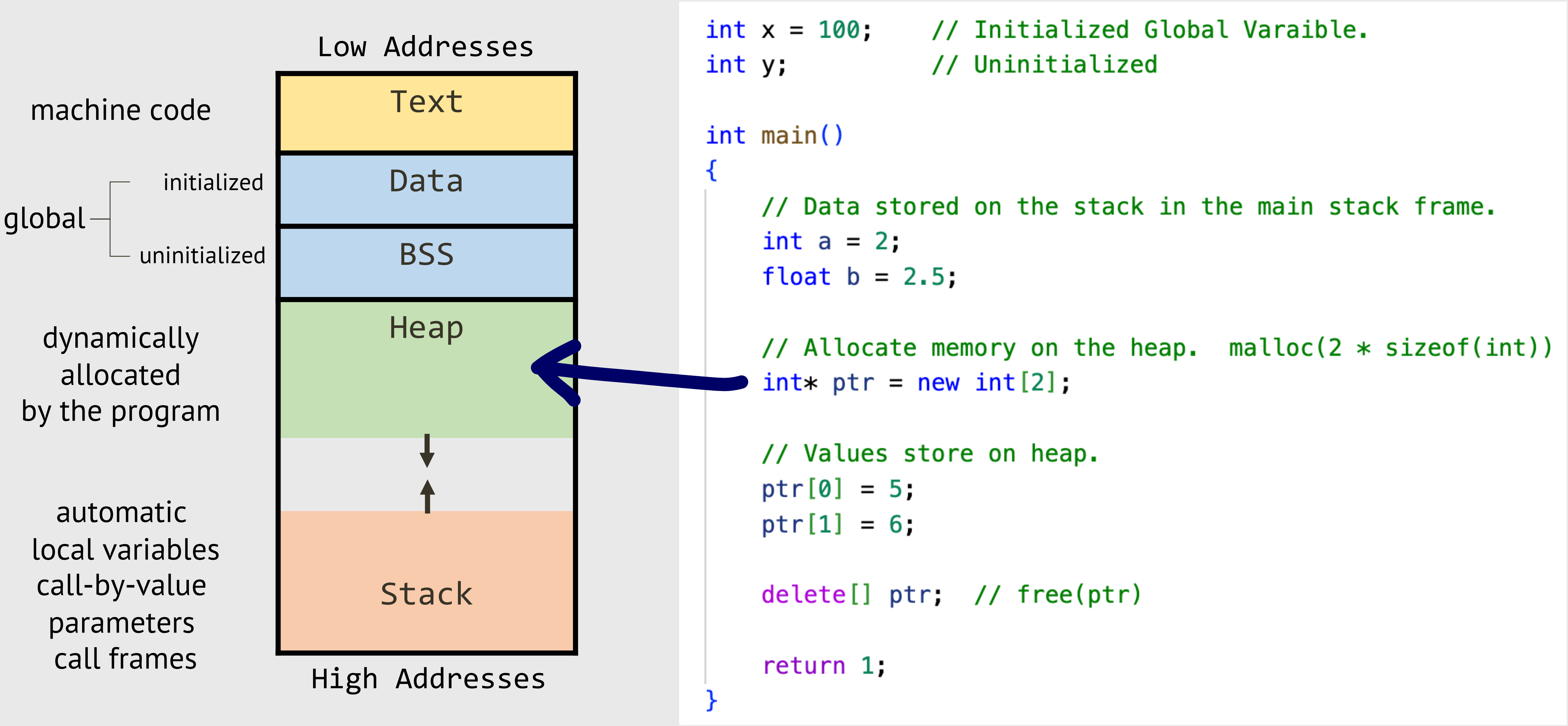


```
int x = 100;     // Initialized Global Varaible.
int y;           // Uninitialized

int main()
{
    // Data stored on the stack in the main stack frame.
    int a = 2;
    float b = 2.5;

    // Allocate memory on the heap.  malloc(2 * sizeof(int))
    int* ptr = new int[2];

    // Values store on heap.
    ptr[0] = 5;
    ptr[1] = 6;

    delete[] ptr;  // free(ptr)

    return 1;
}
```

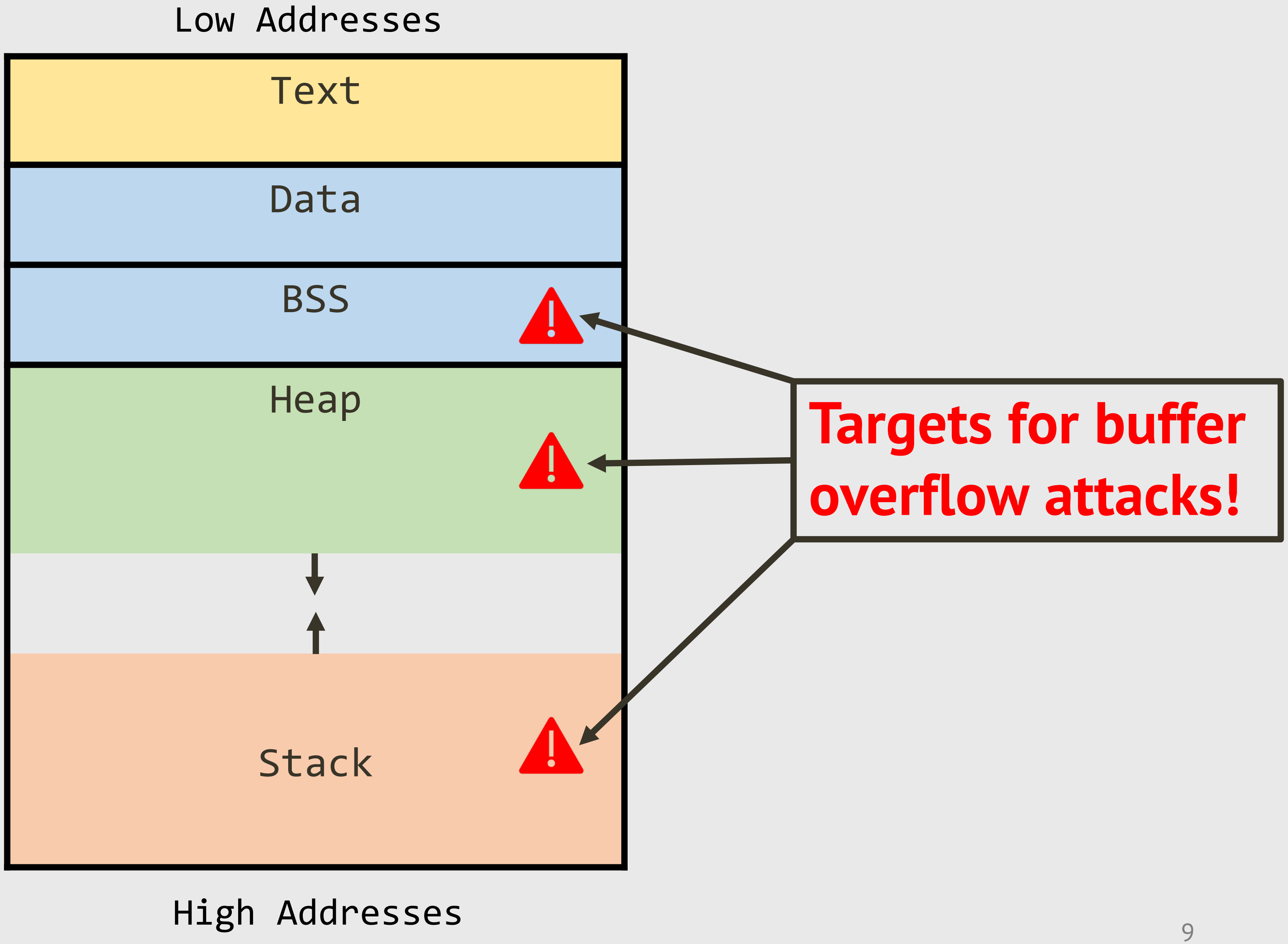

Low Addresses
machine code
Text
global
initialized
Data
uninitialized
BSS
dynamically allocated by the program
Heap
automatic local variables call-by-value parameters call frames
Stack
High Addresses



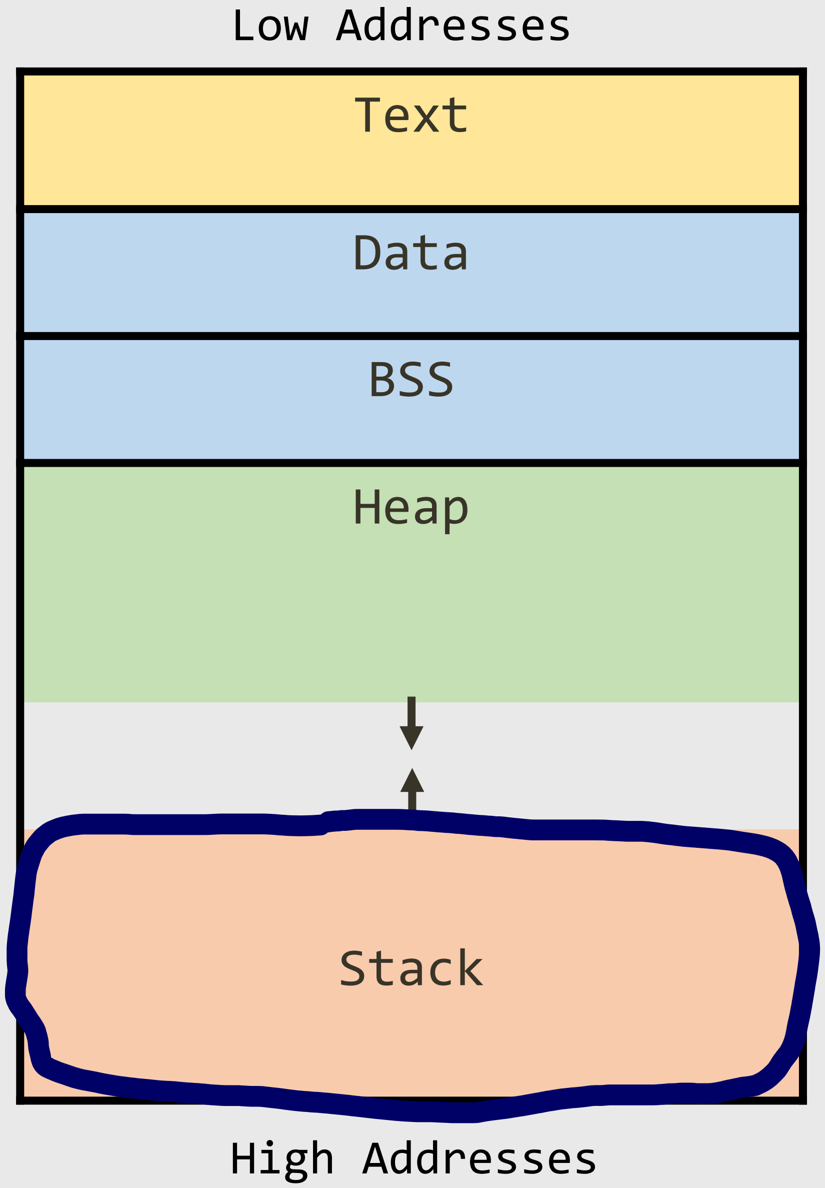
```
int x = 100;    // Initialized Global Varaible.
int y;          // Uninitialized

int main()
{
    // Data stored on the stack in the main stack frame.
    int a = 2;
    float b = 2.5;

    // Allocate memory on the heap.  malloc(2 * sizeof(int))
    int* ptr = new int[2];

    // Values store on heap.
    ptr[0] = 5;
    ptr[1] = 6;

    delete[] ptr;  // free(ptr)

    return 1;
}
```



```
int x = 100;    // Initialized Global Varaible.
int y;          // Uninitialized

int main()
{
    // Data stored on the stack in the main stack frame.
    int a = 2;
    float b = 2.5;

    // Allocate memory on the heap.  malloc(2 * sizeof(int))
    int* ptr = new int[2];

    // Values store on heap.
    ptr[0] = 5;
    ptr[1] = 6;

    delete[] ptr;  // free(ptr)

    return 1;
}
```

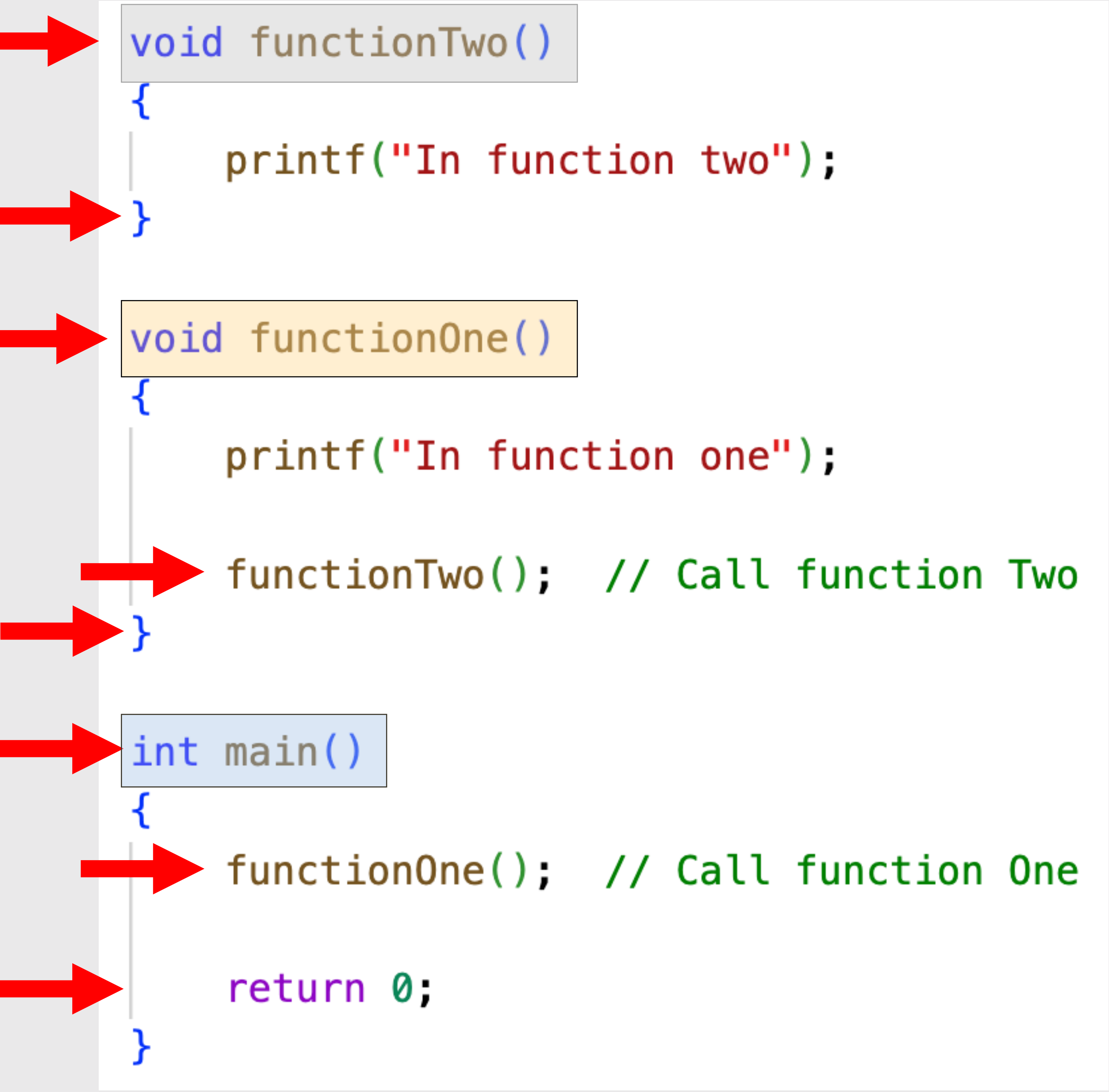


Targets for buffer overflow attacks!

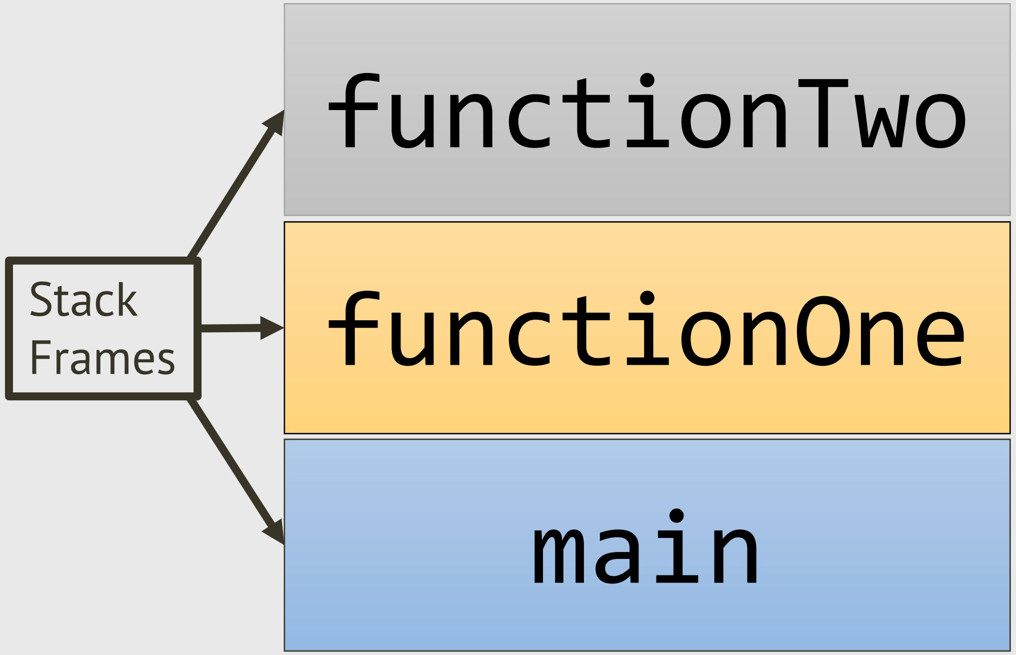

Low Addresses
Text
Data
BSS
Heap
Stack
High Addresses

Call Stack

consisting of stack frames

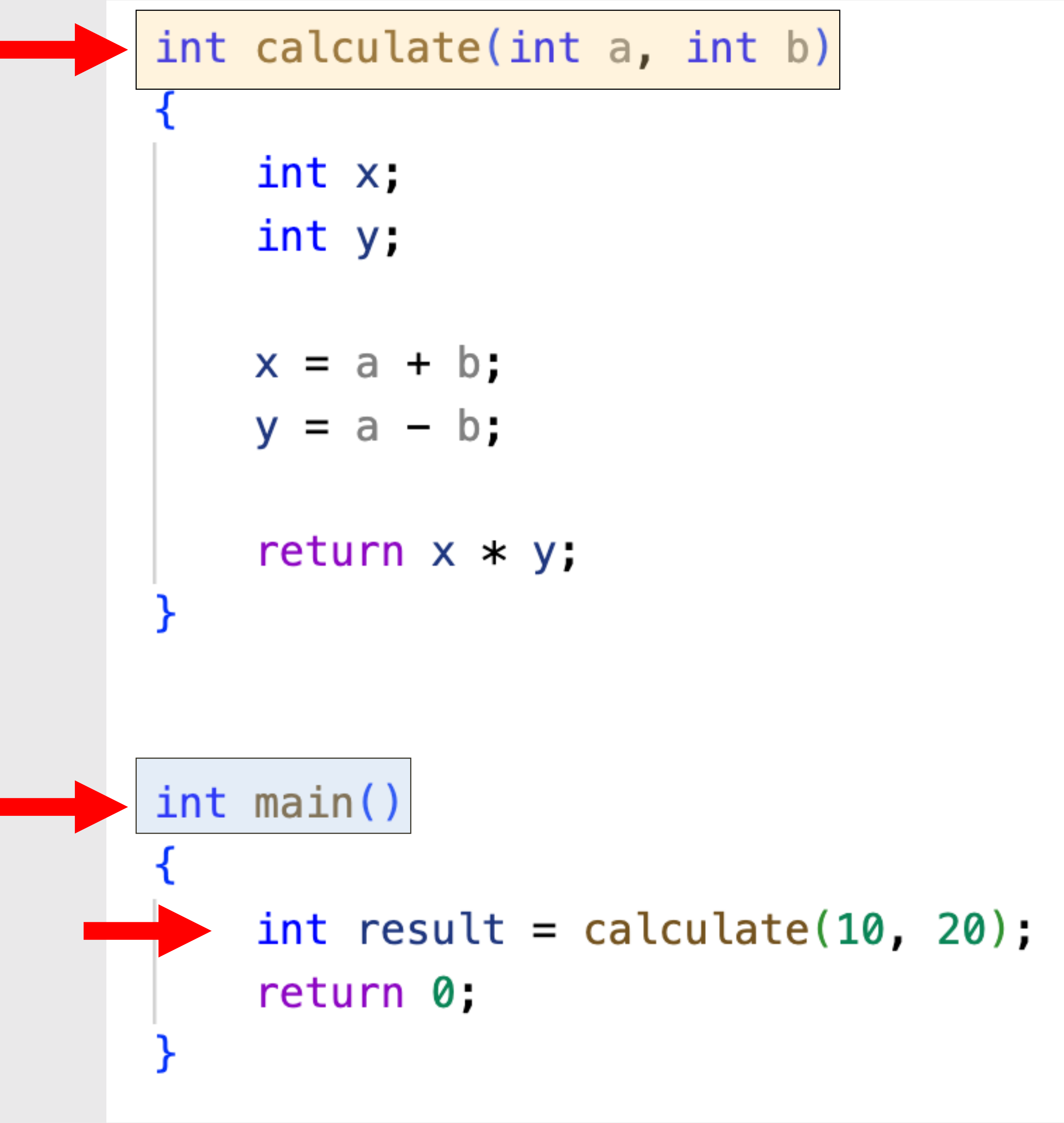


```
void functionTwo()
{
    printf("In function two");
}

void functionOne()
{
    printf("In function one");

    functionTwo();  // Call function Two
}

int main()
{
    functionOne();  // Call function One

    return 0;
}
```

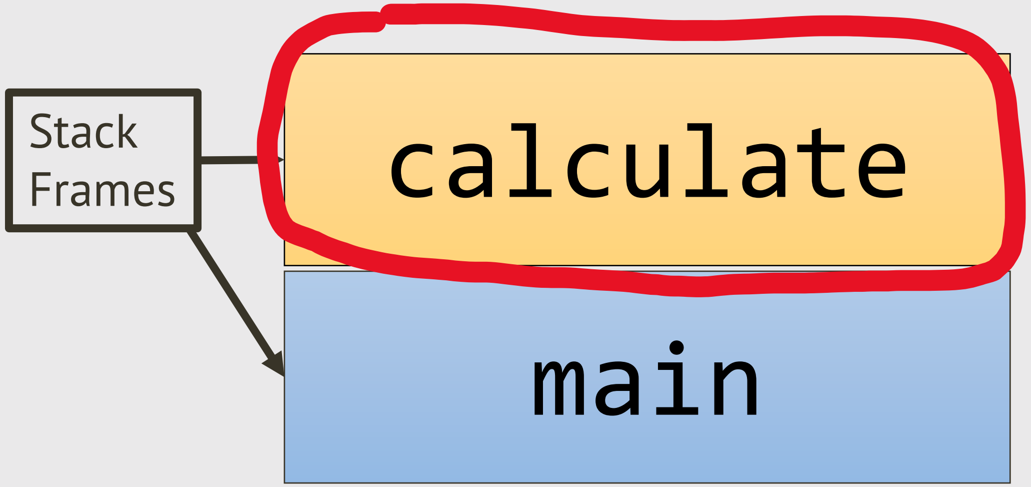


Call Stack

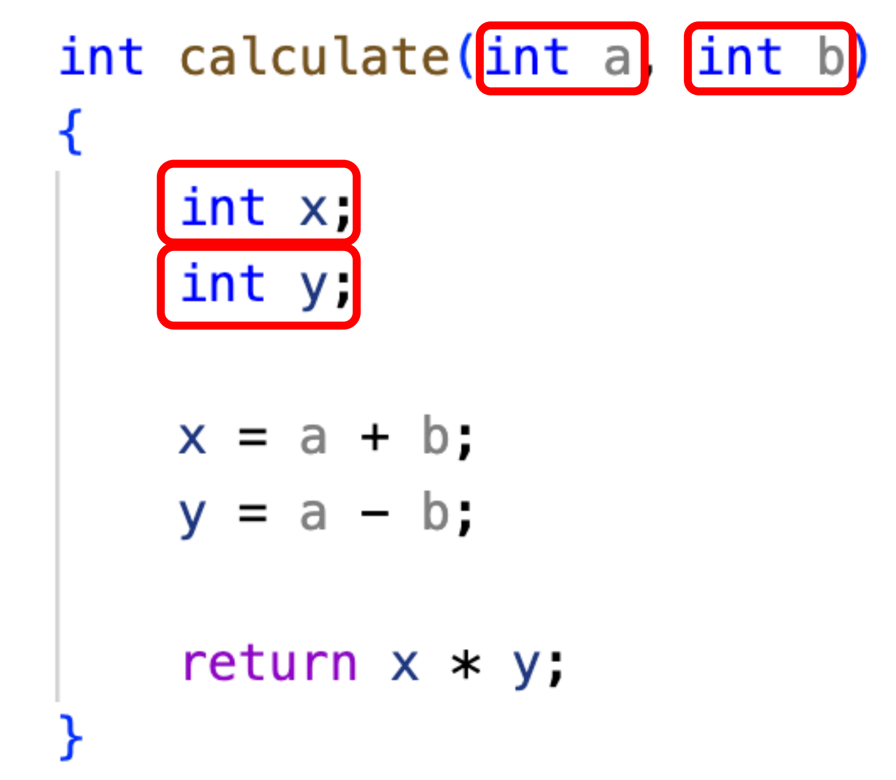
consisting of stack frames



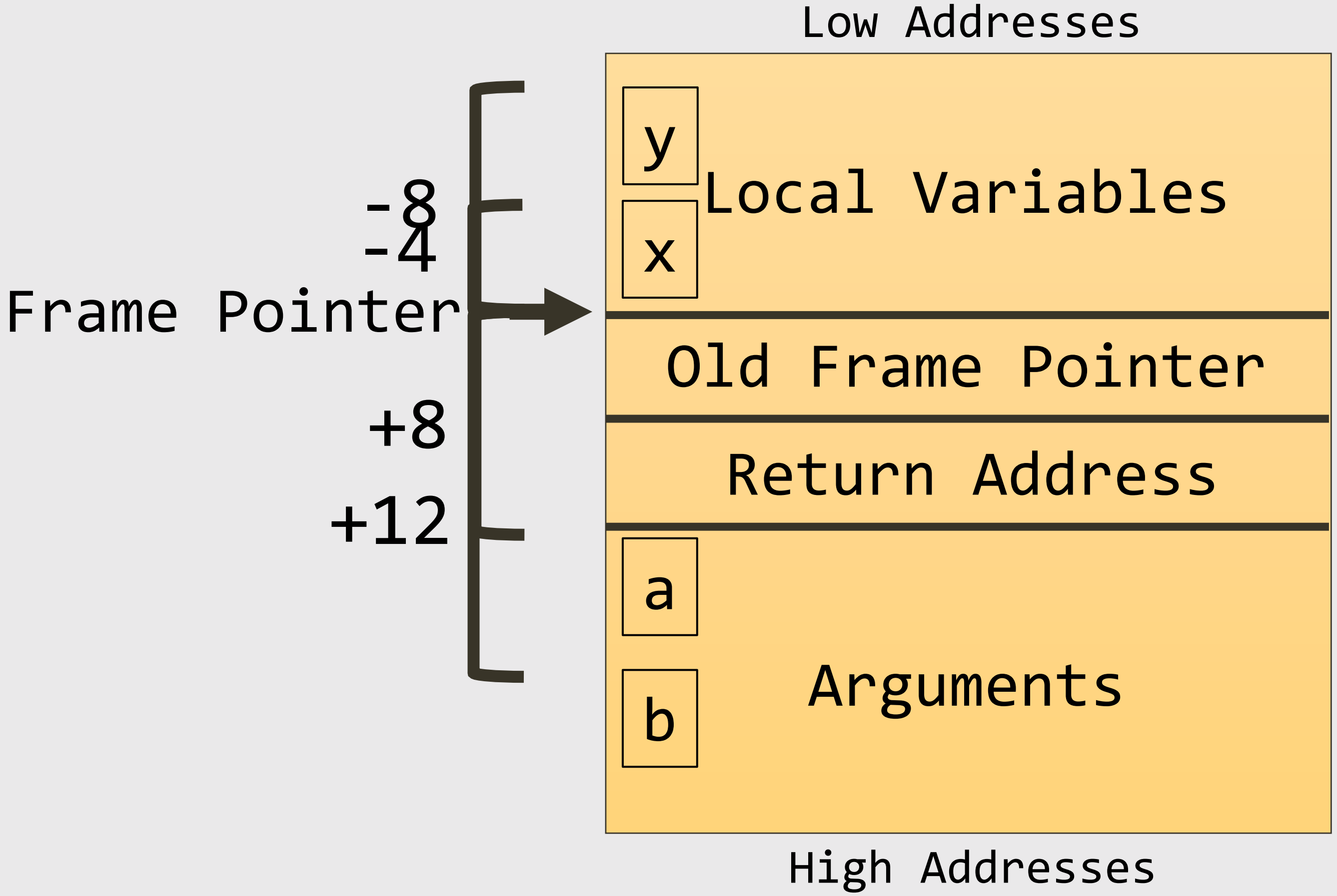
```
int calculate(int a, int b)
{
    int x;
    int y;

    x = a + b;
    y = a - b;

    return x * y;
}


int main()
{
    int result = calculate(10, 20);
    return 0;
}
```



Stack Frame



```
int calculate(int a, int b)
{
    int x;
    int y;

    x = a + b;
    y = a - b;

    return x * y;
}
```



Low Addresses

y

x

Local Variables

-8

-4

Frame Pointer

Old Frame Pointer

+8

Return Address

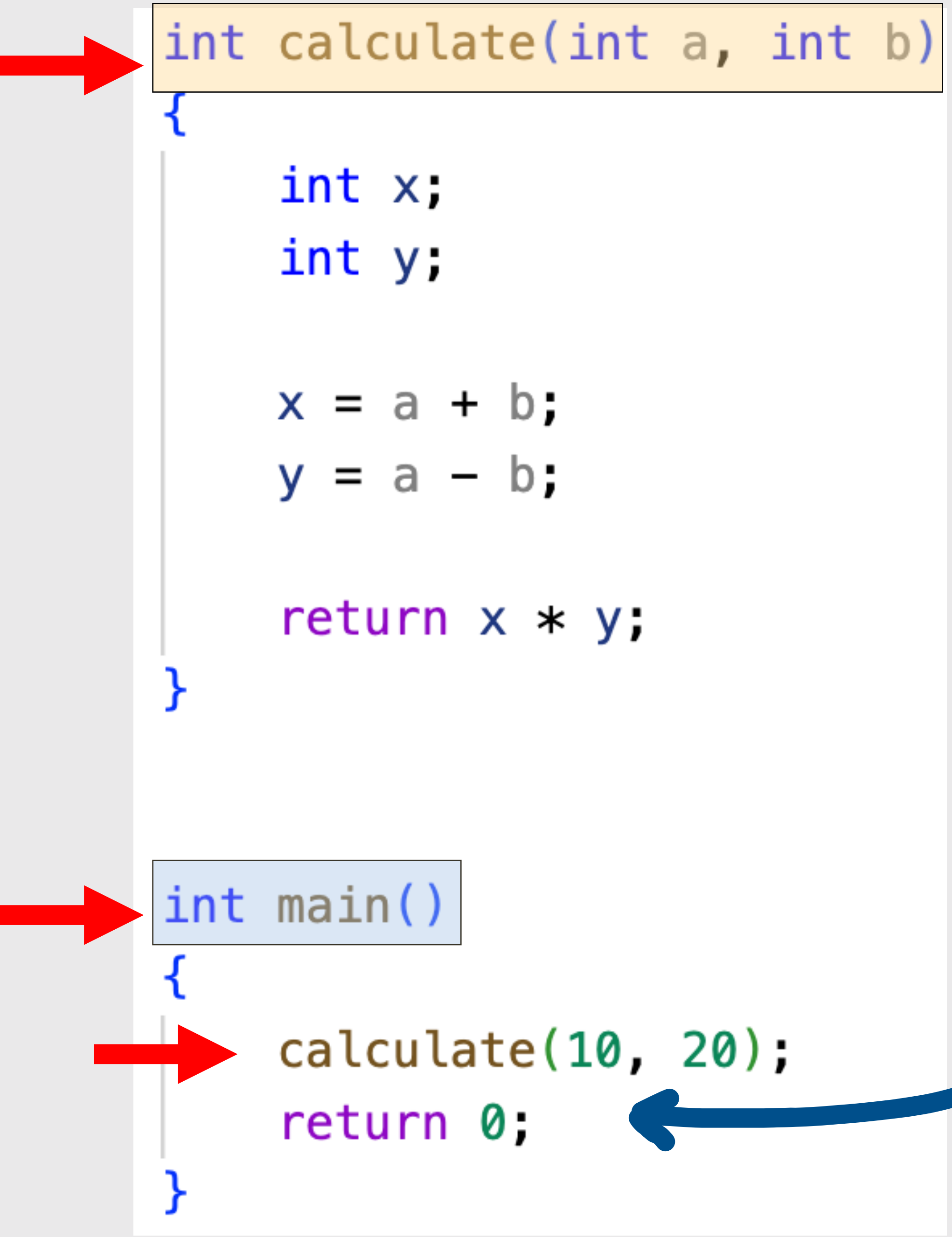
+12

a

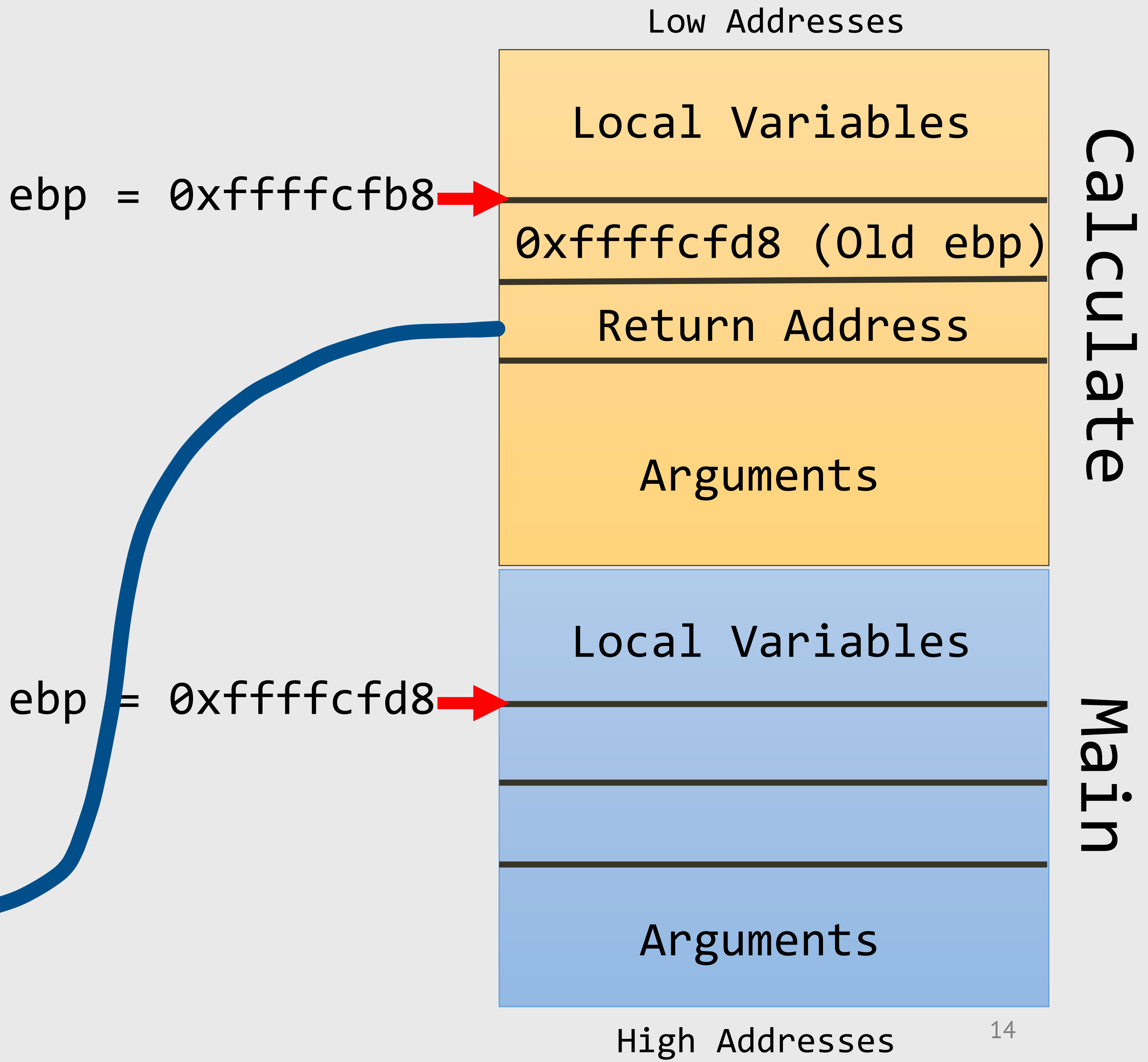
b

Arguments

High Addresses



```
int calculate(int a, int b)
{
    int x;
    int y;

    x = a + b;
    y = a - b;

    return x * y;
}


int main()
{
    calculate(10, 20);
    return 0;
}
```



Low Addresses

Local Variables

ebp = 0xffffcfb8

0xffffcfd8 (Old ebp)

Return Address

Arguments

Calculate

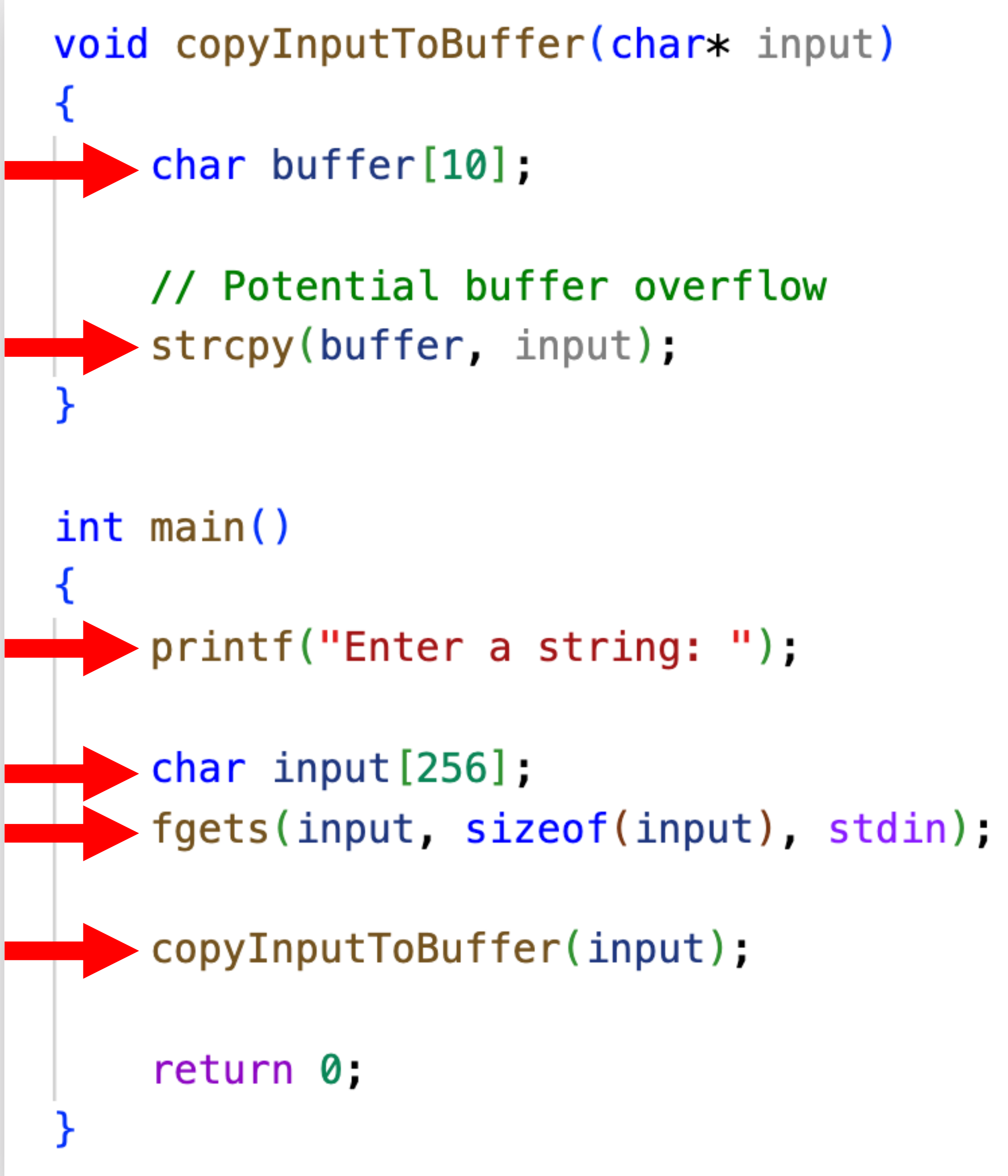
Local Variables

ebp = 0xffffcfd8

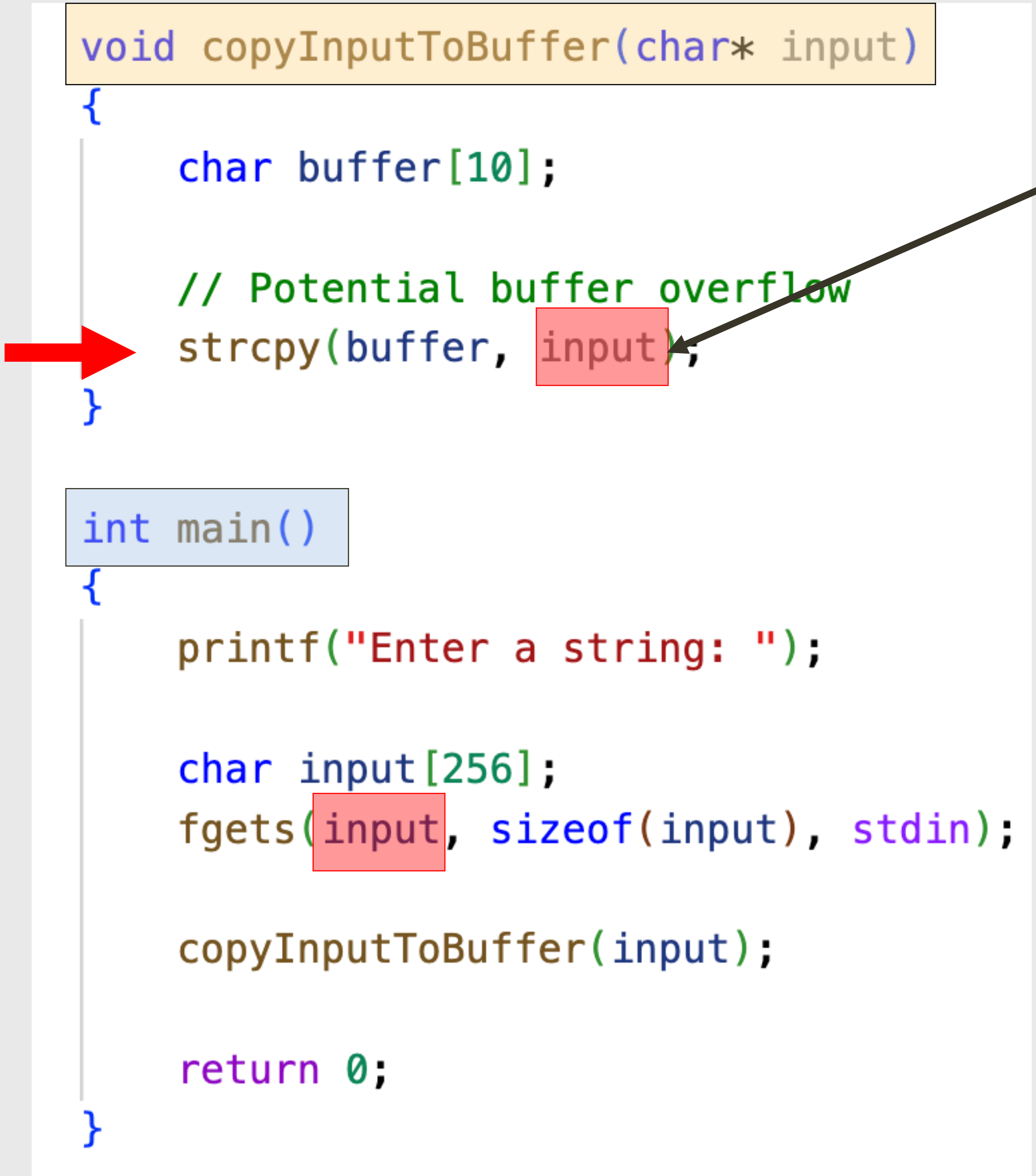
Arguments

Main

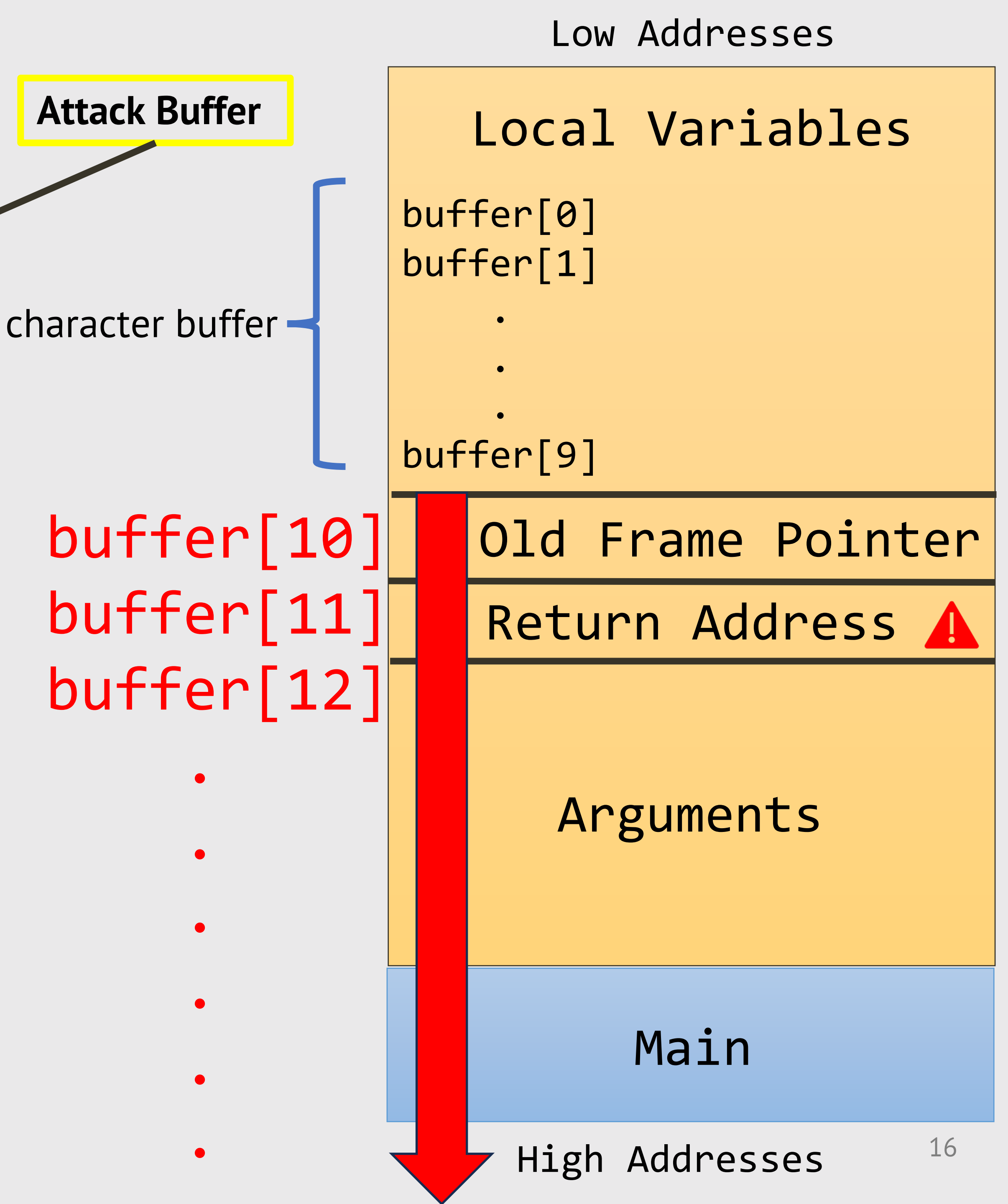
High Addresses



```
void copyInputToBuffer(char* input)
{
    char buffer[10];

    // Potential buffer overflow
    strcpy(buffer, input);
}

int main()
{
    printf("Enter a string: ");

    char input[256];
    fgets(input, sizeof(input), stdin);

    copyInputToBuffer(input);

    return 0;
}
```



```c
void copyInputToBuffer(char* input)
{
    char buffer[10];

    // Potential buffer overflow
    strcpy(buffer, input);
}

int main()
{
    printf("Enter a string: ");

    char input[256];
    fgets(input, sizeof(input), stdin);

    copyInputToBuffer(input);

    return 0;
}
```



What can attacker do with a buffer overflow?

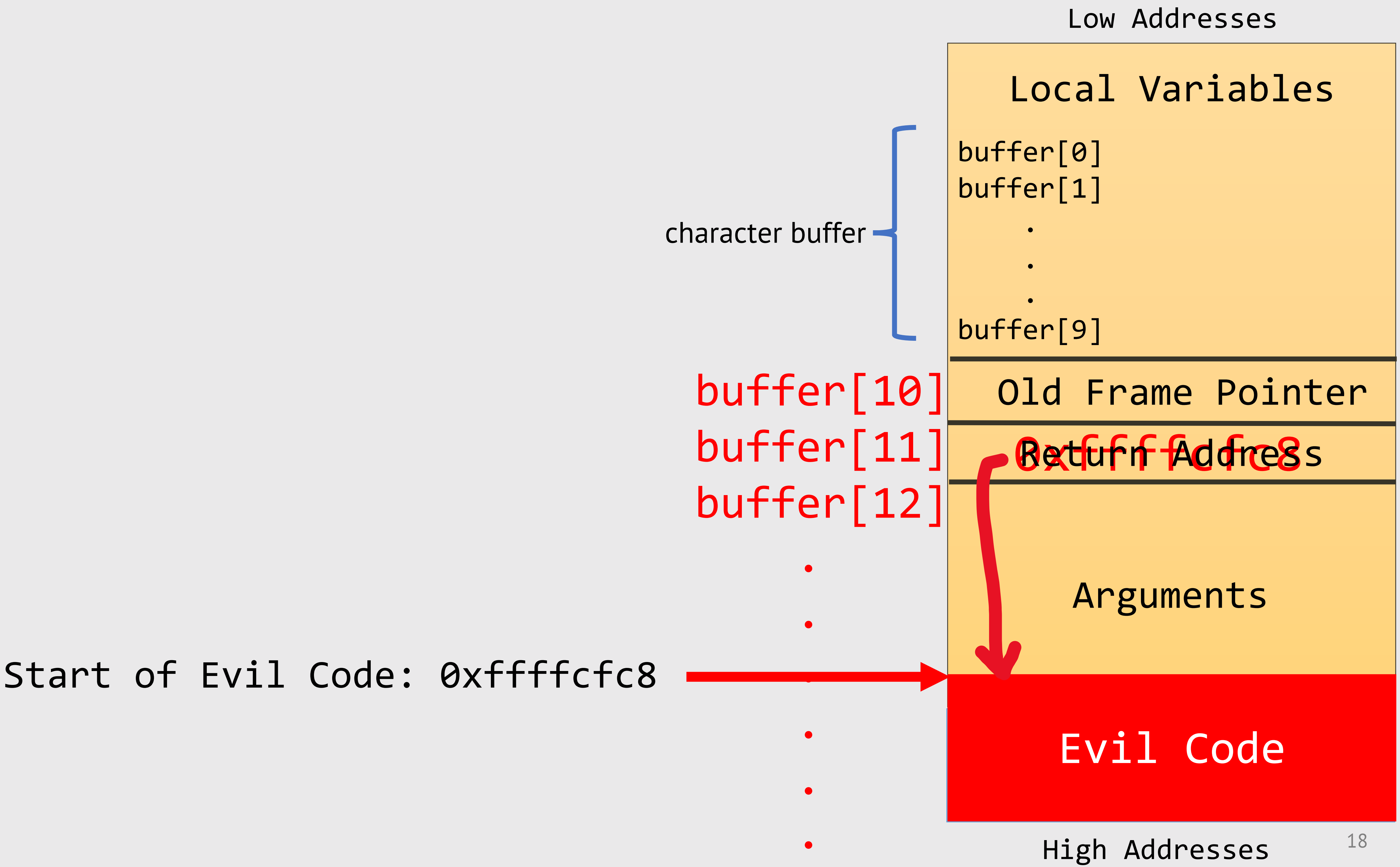
1. Modify data on the stack
   - variables
   - return address

2. Crash the program

3. Inject malicious code on the stack
   - change the return address to point to this code

4. Change the return address to point somewhere else in the program

5. Change the return address to point somewhere in a library



Low Addresses

Local Variables

buffer[0]

buffer[1]

.

.

.

buffer[9]

character buffer

buffer[10]

buffer[11]

buffer[12]

Old Frame Pointer

Return Address

0xffffcfc8

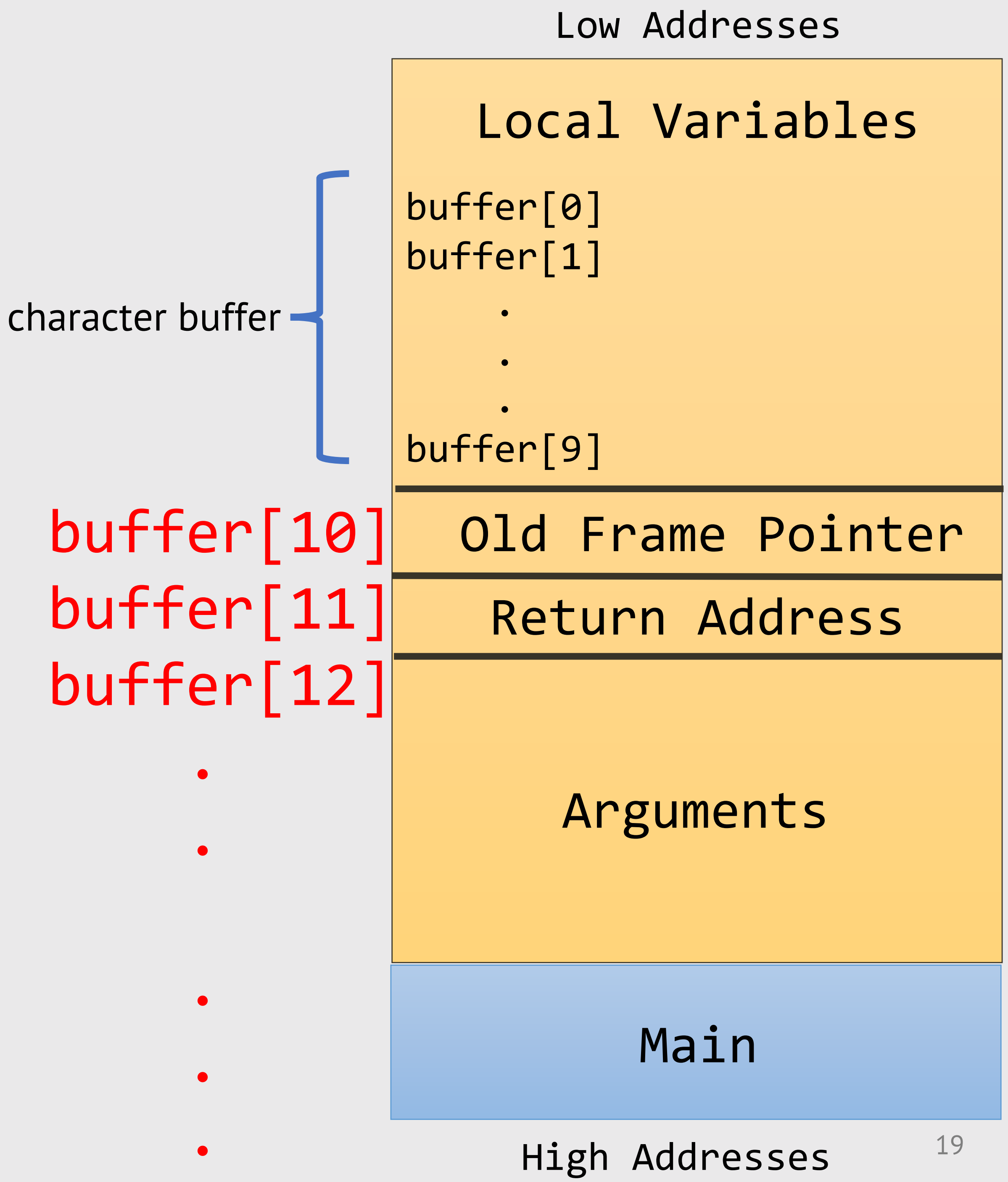
Arguments

Start of Evil Code: 0xffffcfc8

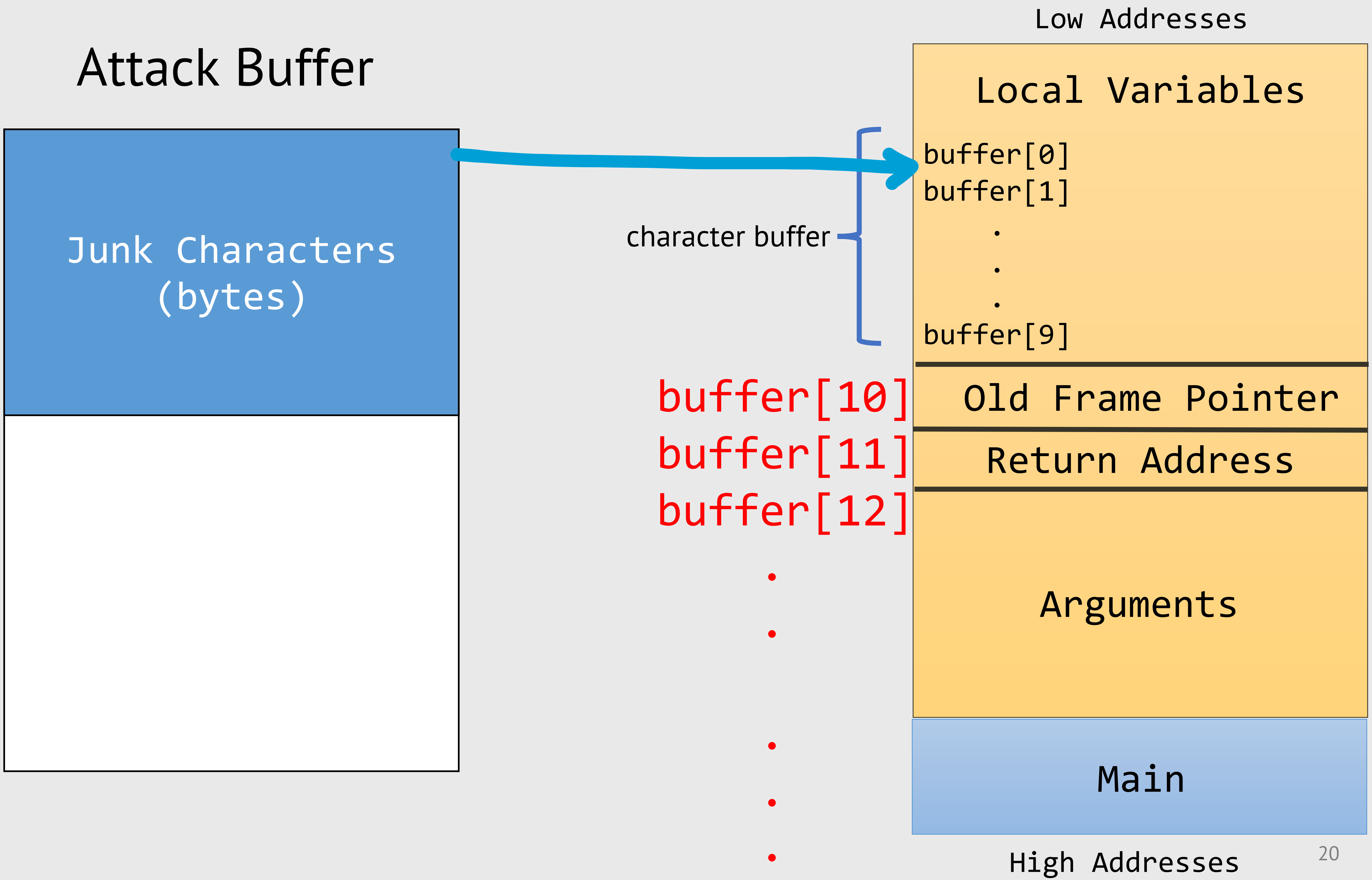
Evil Code

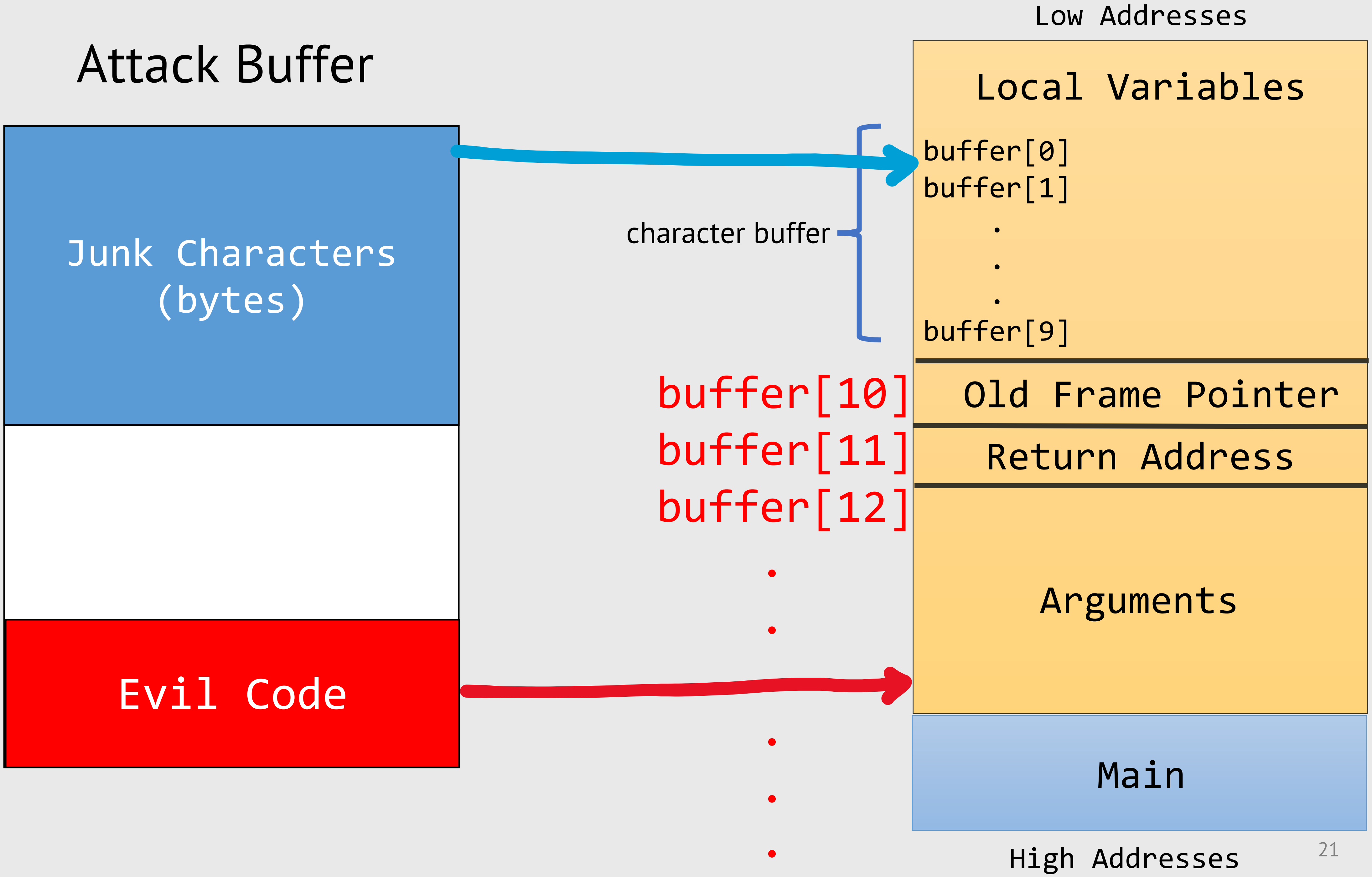
High Addresses

Attack buffer including payload



Attack Buffer





Attack Buffer

Junk Characters (bytes)

Evil Code

character buffer

Low Addresses

Local Variables

buffer[0]
buffer[1]
.
.
.
buffer[9]

buffer[10]
buffer[11]
buffer[12]
.
.
.
.
.

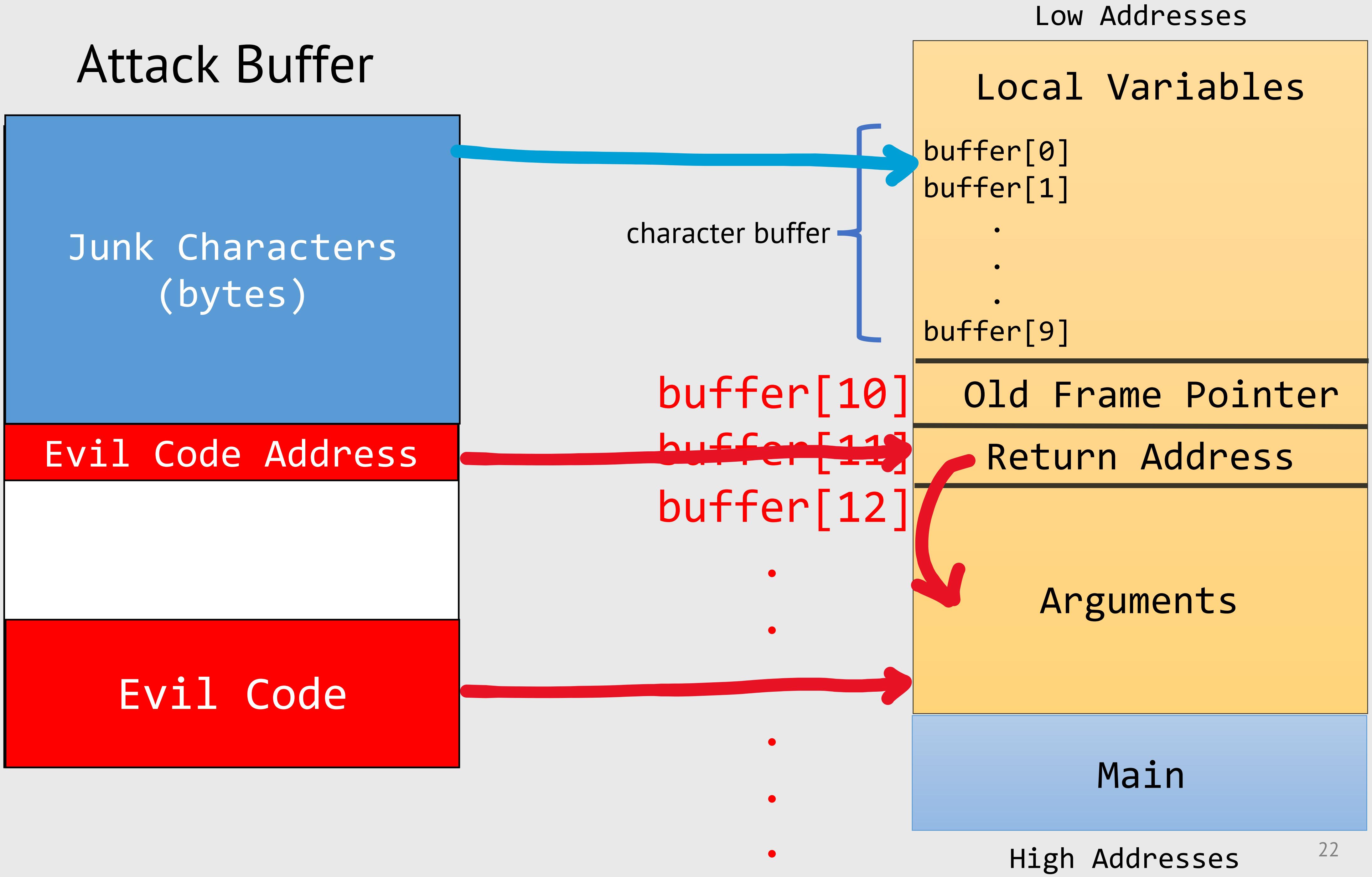
Old Frame Pointer

Return Address

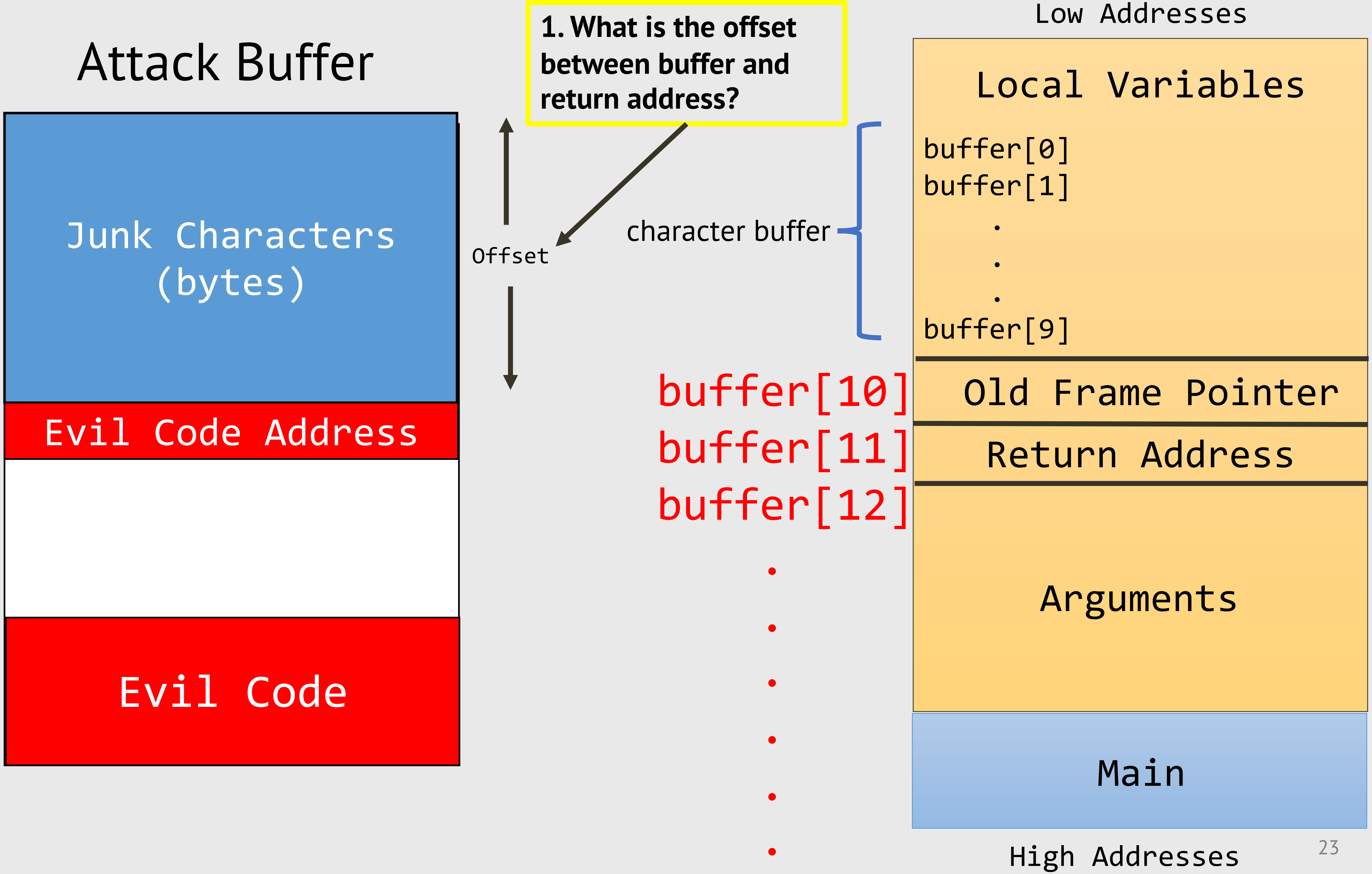
Arguments

Main

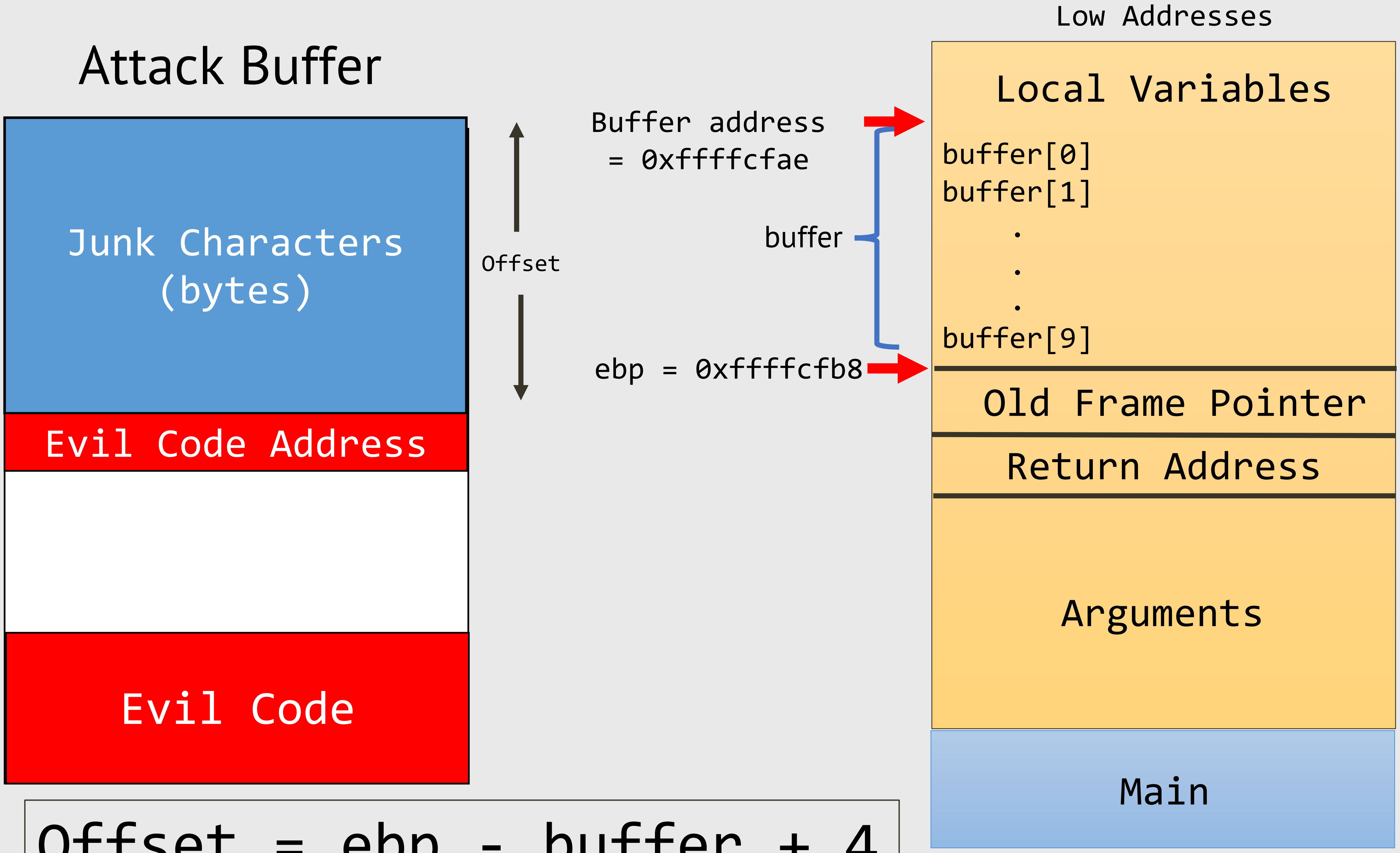
High Addresses


Attack Buffer
Junk Characters (bytes)
Evil Code Address
Evil Code
character buffer
buffer[10]
buffer[11]
buffer[12]
Low Addresses
Local Variables
buffer[0]
buffer[1]
.
.
.
buffer[9]
Old Frame Pointer
Return Address
Arguments
Main
High Addresses

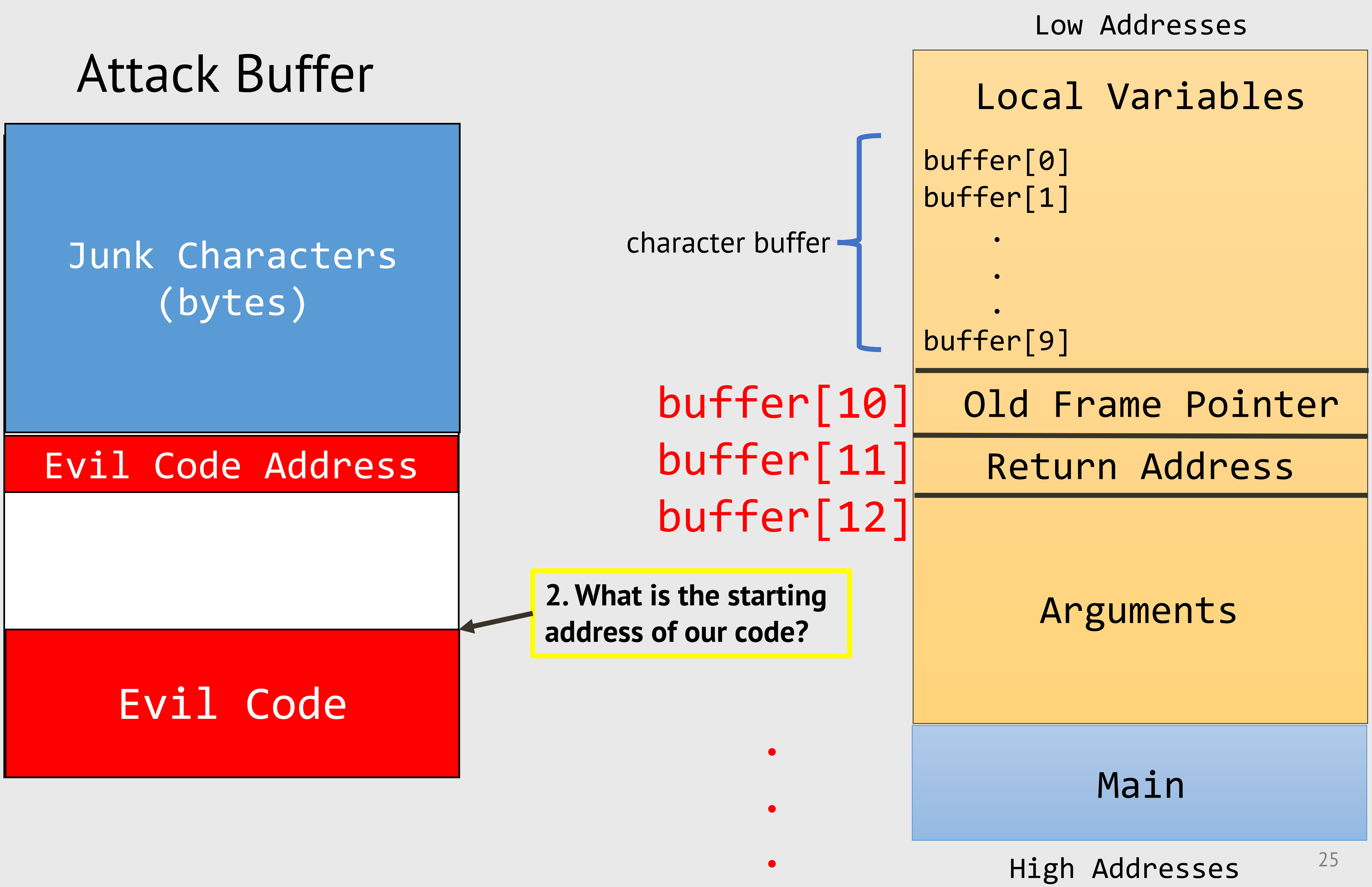
1. What is the offset between buffer and return address?



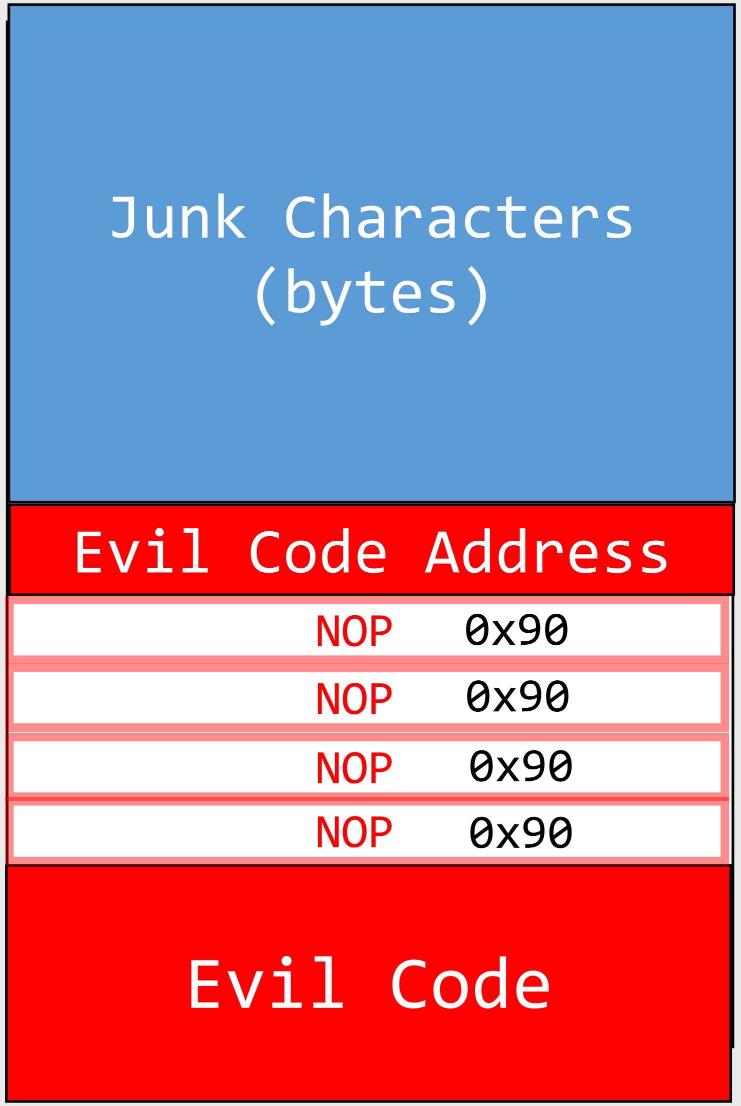
Attack Buffer

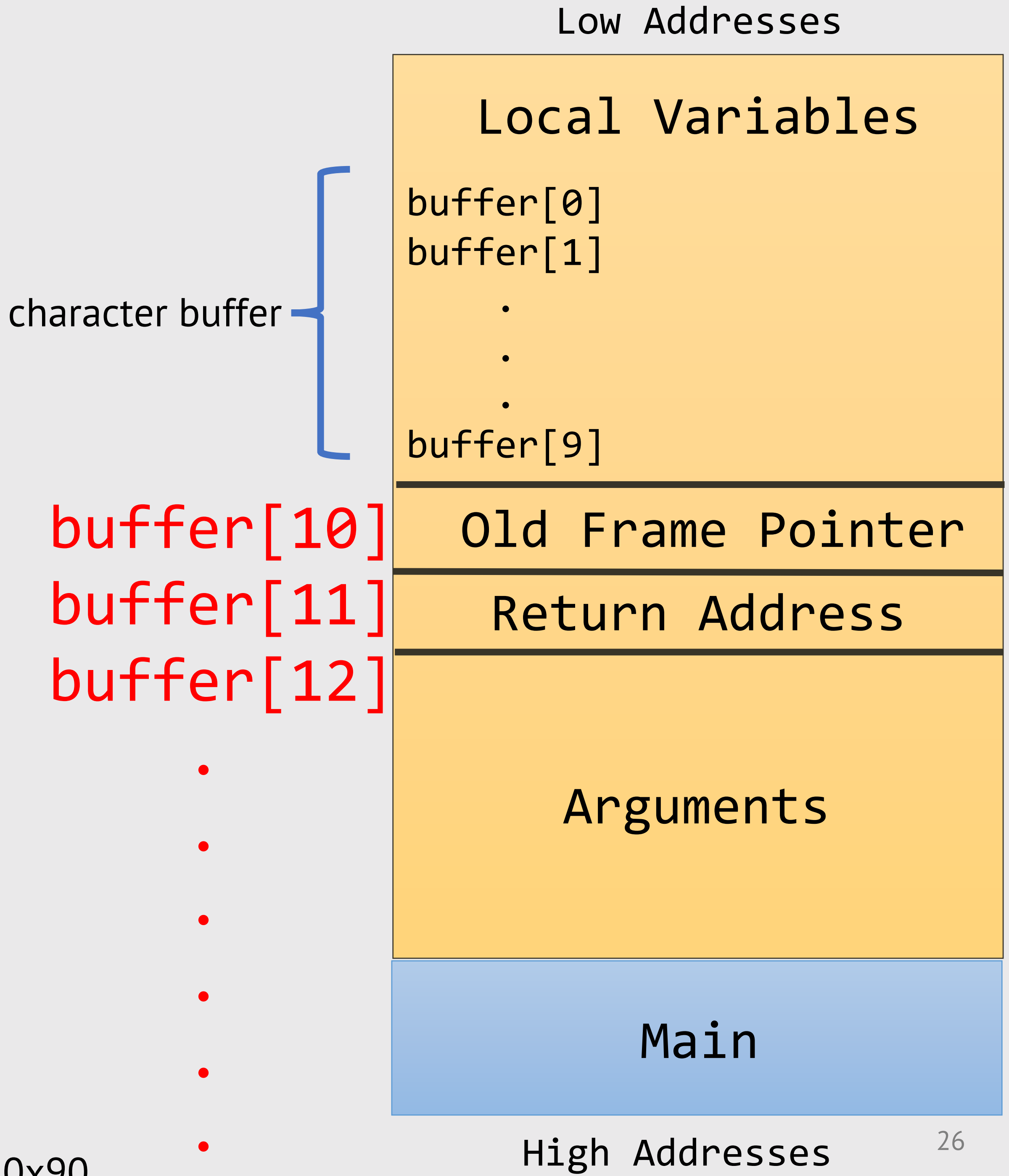


Offset = ebp - buffer + 4


Attack Buffer
Junk Characters (bytes)
Evil Code Address
Evil Code
2. What is the starting address of our code?
Low Addresses
Local Variables
character buffer
buffer[0]
buffer[1]
.
.
.
buffer[9]
buffer[10]
buffer[11]
buffer[12]
Old Frame Pointer
Return Address
Arguments
Main
High Addresses

Attack Buffer



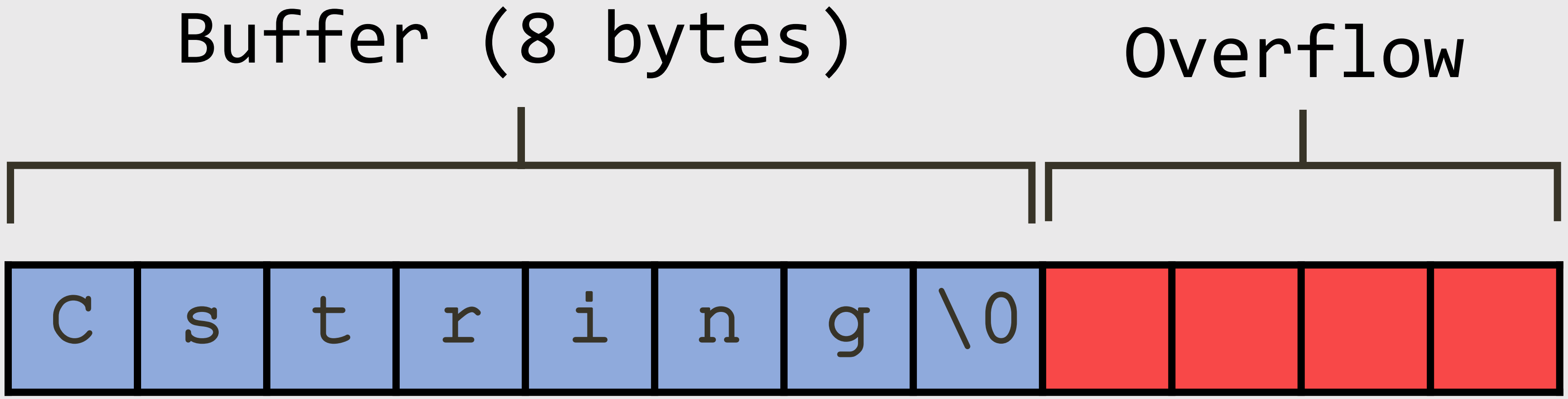


Create multiple entry points! (**NOP Sled**)

In the x86 architecture, the NOP instruction number is 0x90

Problem with **zeros** in the attack buffer

- In C, the NUL character is a special character with the value zero used to signify the end of a string

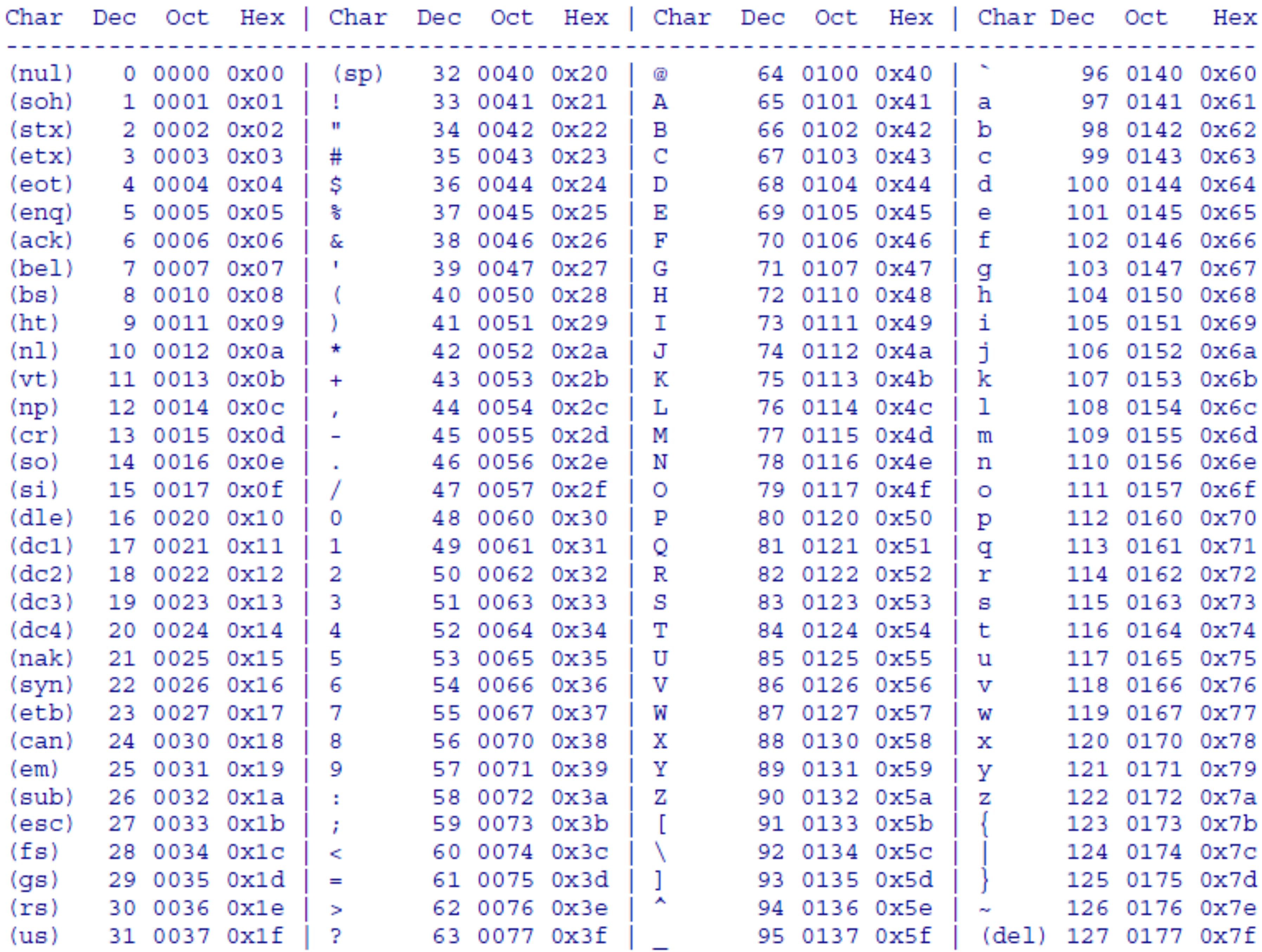


Buffer (8 bytes)

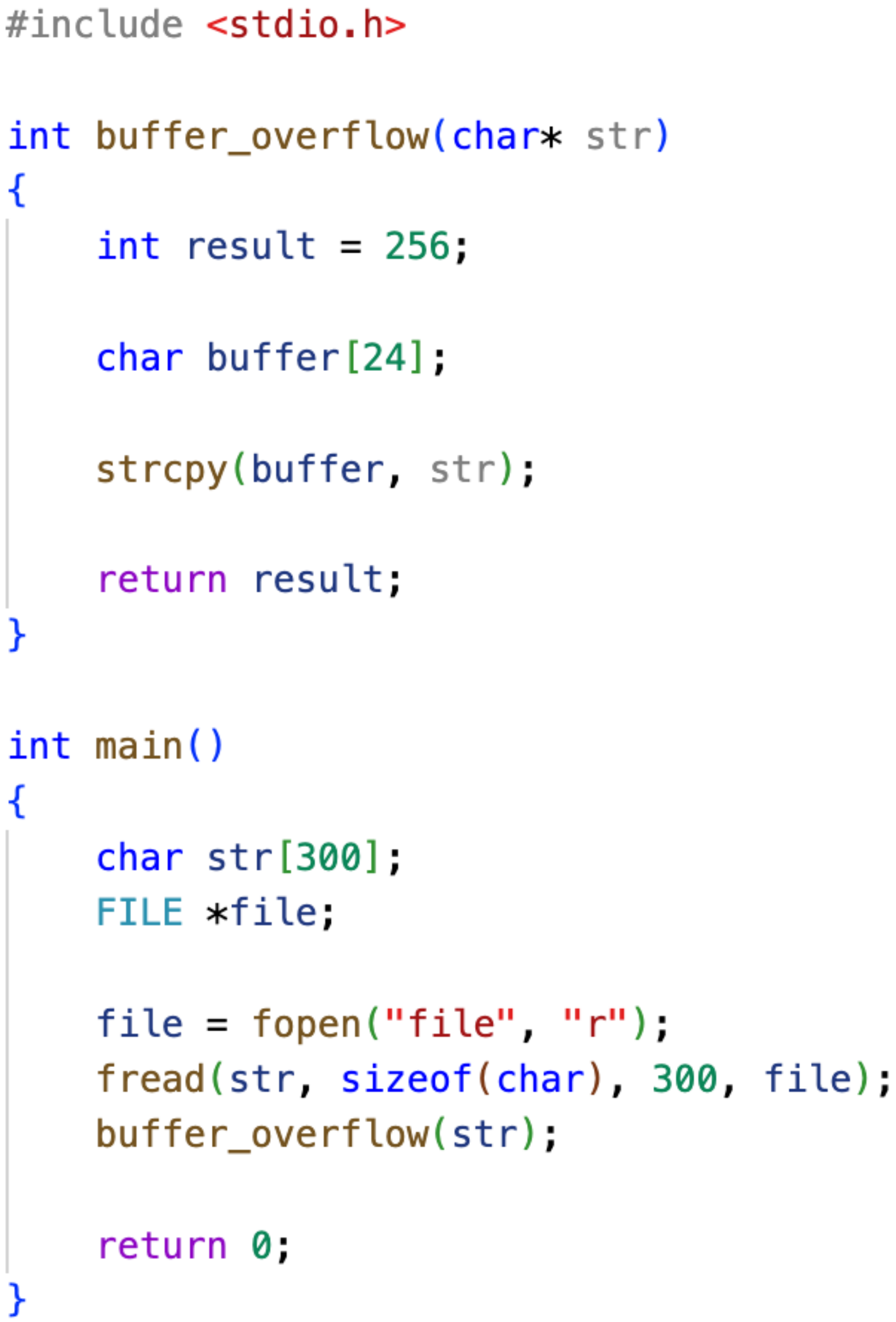
Overflow

C s t r i n g \0

ASCII table



Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
(eot)	4	0004	0x04	$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
(bel)	7	0007	0x07	'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(bs)	8	0010	0x08	(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
(ht)	9	0011	0x09	)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
(np)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f	(del)	127	0177	0x7f



```c
#include <stdio.h>

int buffer_overflow(char* str)
{
    int result = 256;

    char buffer[24];

    strcpy(buffer, str);

    return result;
}

int main()
{
    char str[300];
    FILE *file;

    file = fopen("file", "r");
    fread(str, sizeof(char), 300, file);
    buffer_overflow(str);

    return 0;
}
```

Exercise:

1. Draw the stack
2. Calculate the **offset**
3. Show where you would put your evil code and NOPs
4. Insert a **return address** to run your evil code

The **`buffer`** char array is at address `0xAABB0010`

The frame pointer (**ebp**) is currently set to `0XAABB0050`

Buffer Overflows

1. How they work
2. Countermeasures
3. Shellcode

Countermeasures

1. Developer Safeguards

2. Address Space Layout Randomization (ASLR)

3. Non-Executable Stacks

4. Stack Protector (aka. Stack Canary)

Countermeasures

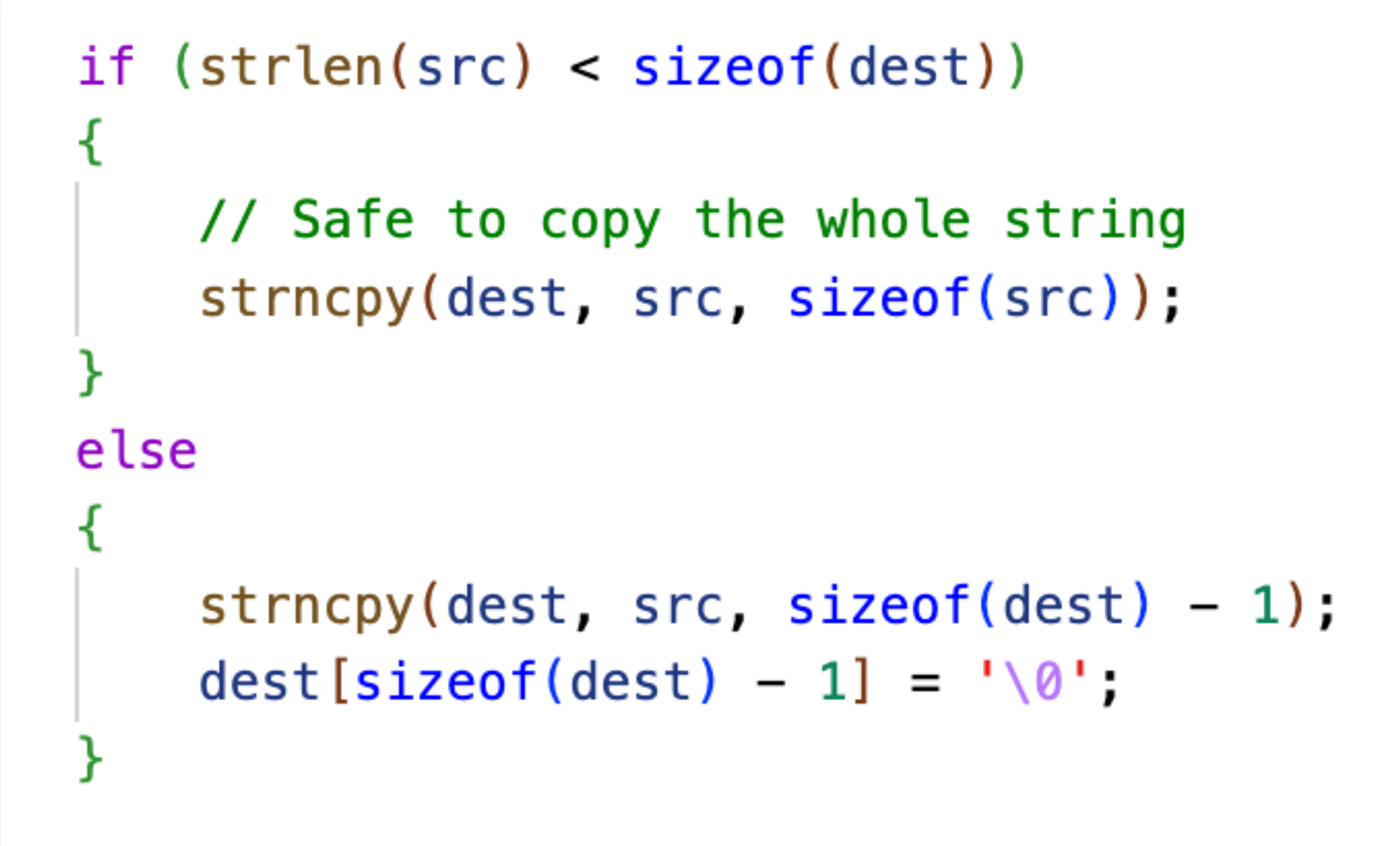
1. Developer Safeguards

2. Address Space Layout Randomization (ASLR)

3. Non-Executable Stacks

4. Stack Protector (aka. Stack Canary)

Always check data length



```c
if (strlen(src) < sizeof(dest))
{
    // Safe to copy the whole string
    strncpy(dest, src, sizeof(src));
}
else
{
    strncpy(dest, src, sizeof(dest) - 1);
    dest[sizeof(dest) - 1] = '\0';
}
```

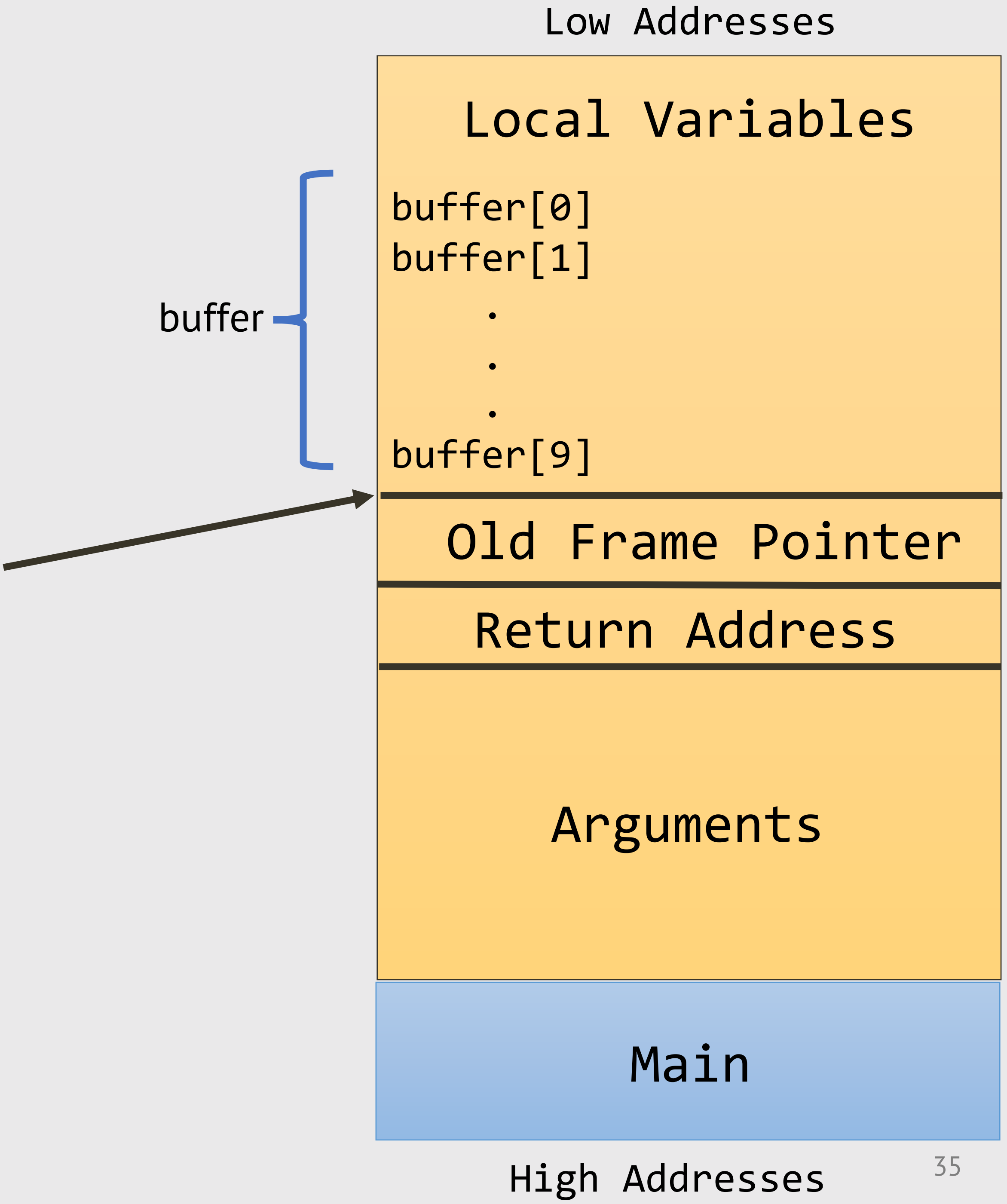
Never let user’s set the length

```
No    char* strcpy(char* dest, const char* src);
Yes   char* strncpy(char* dest, const char* src, size_t n);

No    char* strcat(char* dest, const char* src);
Yes   char* strncat(char* dest, const char* src , size_t n);

No    char* sprintf(char* str, const char* format, …);
Yes   char* snprintf(char* str, size_t size, const char* format, …);

No    char* gets(char* str);
Yes   char* fgets(char* str, int size, FILE* stream);
```

Use a safe libraries

For example, `libsafe`

✓Does not let a buffer grow past the old frame pointer



Use a safer programming language

If you have an option, you could select a language that has protections from buffer overflows

For example, Java

- Automatic bounds checking on all array accesses
- Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException

Countermeasures

1. Developer Safeguards

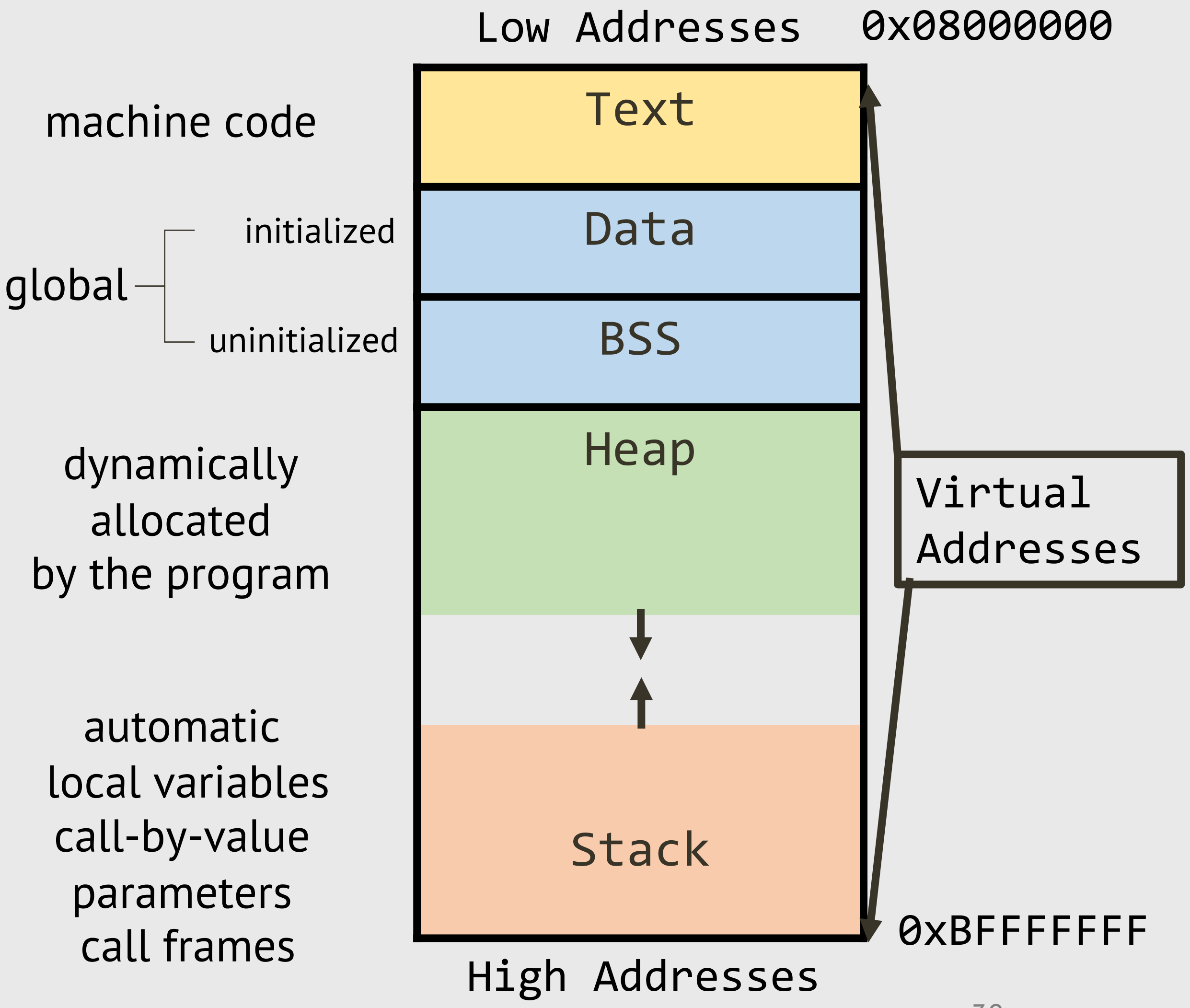
2. Address Space Layout Randomization (ASLR)

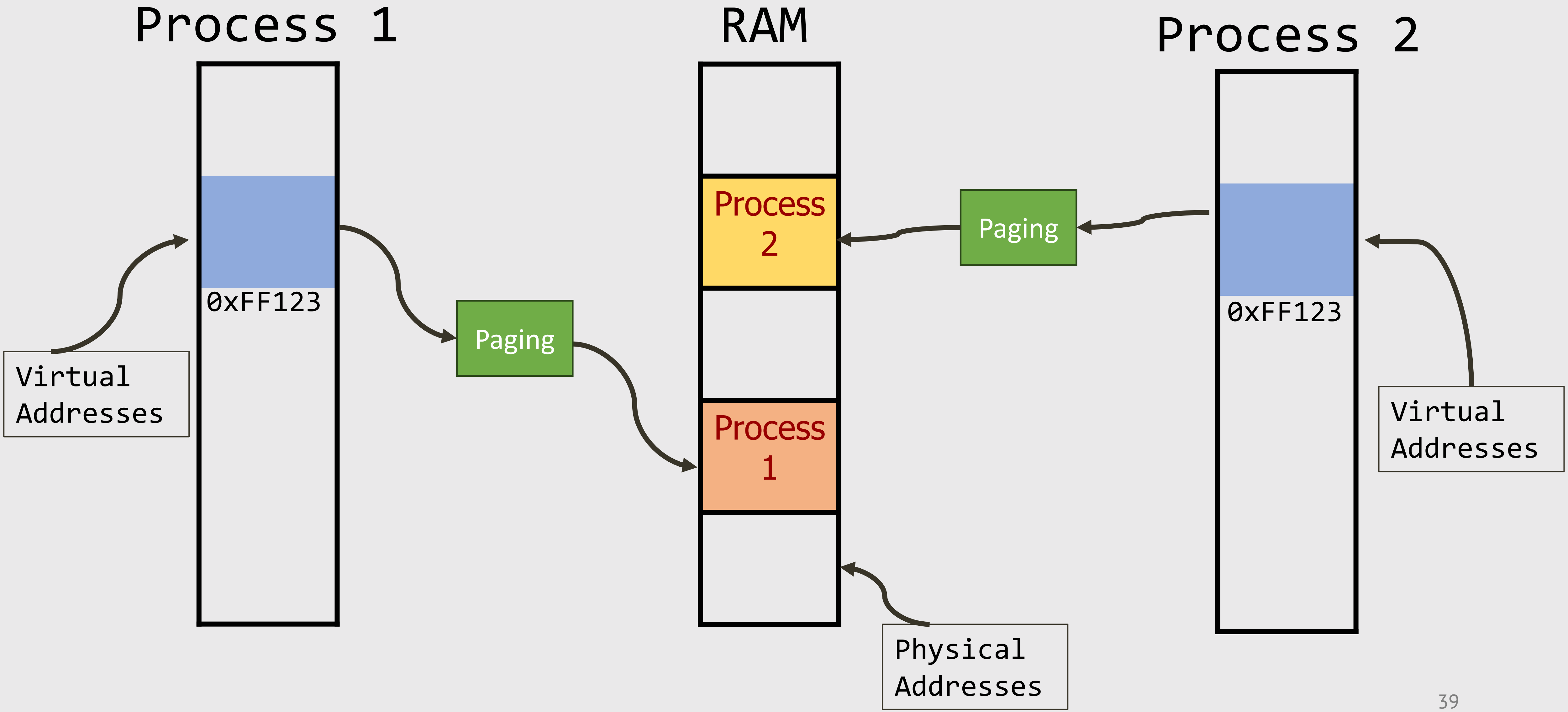
3. Non-Executable Stacks

4. Stack Protector (aka. Stack Canary)

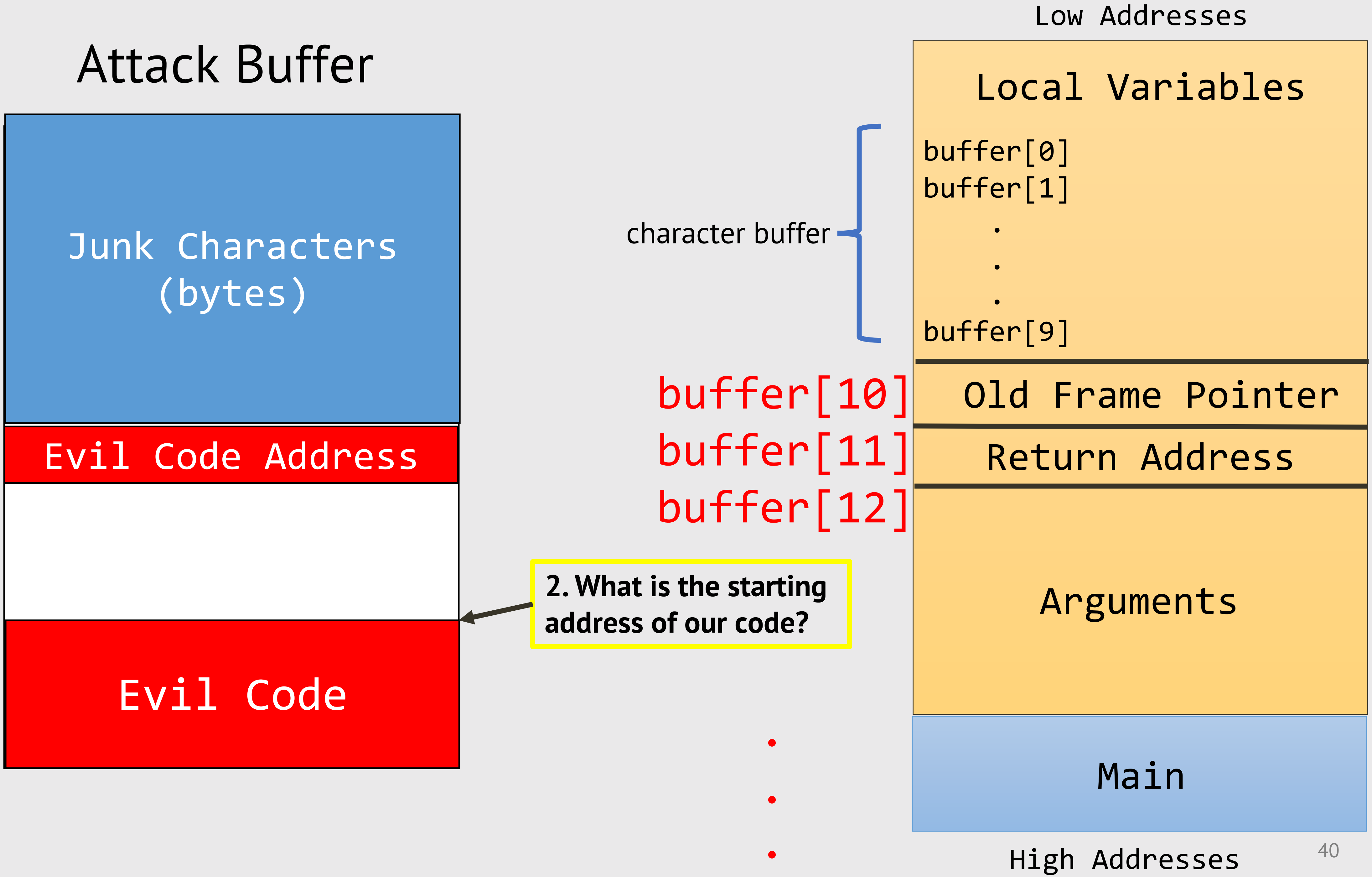
Address Space Layout Randomization (ASLR)

1. Limits a buffer overflow attack once it exists
2. Makes it hard to find return address and NOP sled code




Process 1
RAM
Process 2
0xFF123
Virtual Addresses
Paging
Process 2
Process 1
Paging
0xFF123
Virtual Addresses
Physical Addresses

Attack Buffer



2. What is the starting address of our code?

Countermeasures

1. Developer Safeguards

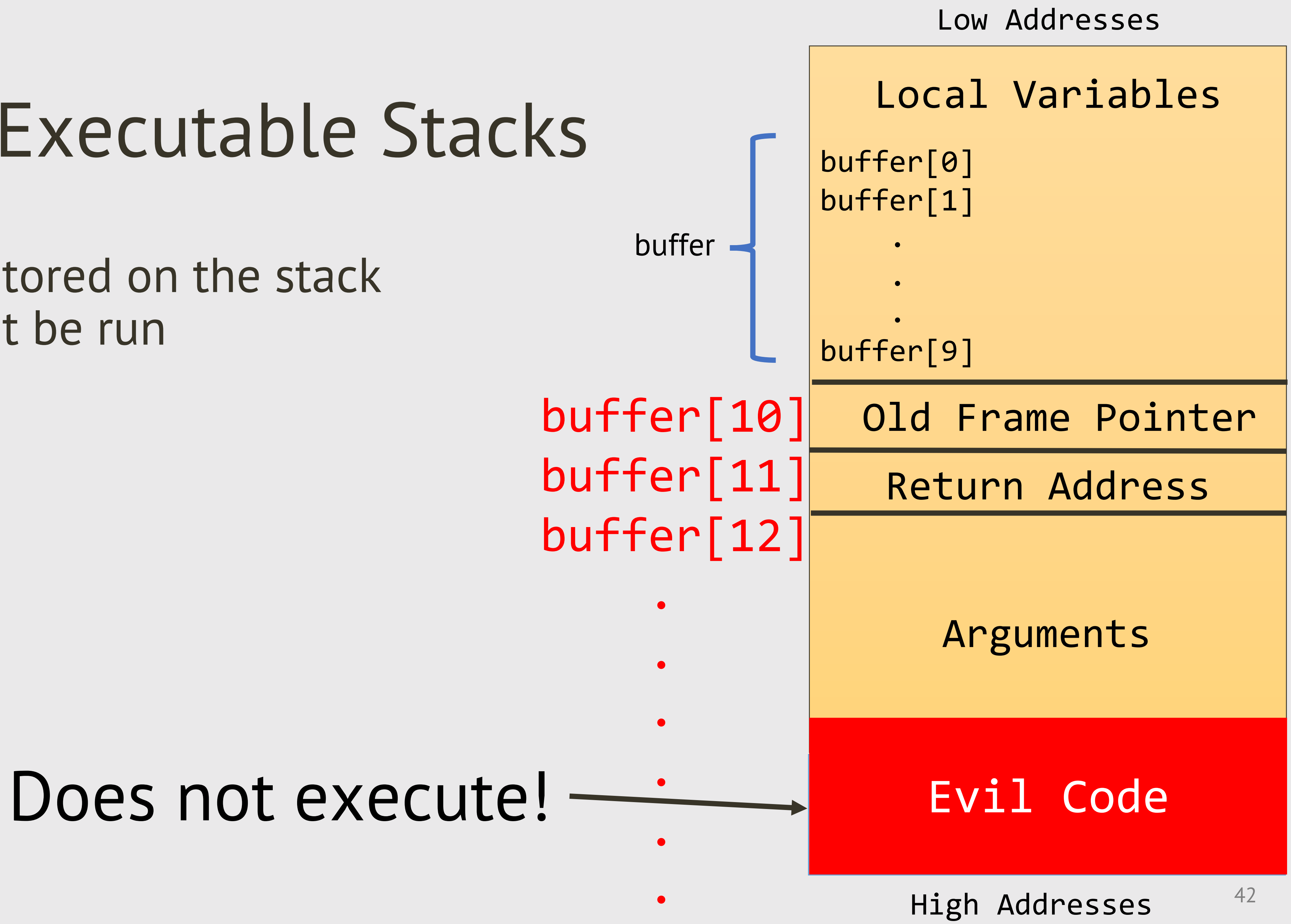
2. Address Space Layout Randomization (ASLR)

3. Non-Executable Stacks

4. Stack Canary

Non-Executable Stacks

- Code stored on the stack can not be run



Low Addresses

Local Variables

buffer

buffer[0]
buffer[1]
.
.
.
buffer[9]

buffer[10]
buffer[11]
buffer[12]

Old Frame Pointer

Return Address

Arguments

Evil Code

Does not execute!

High Addresses

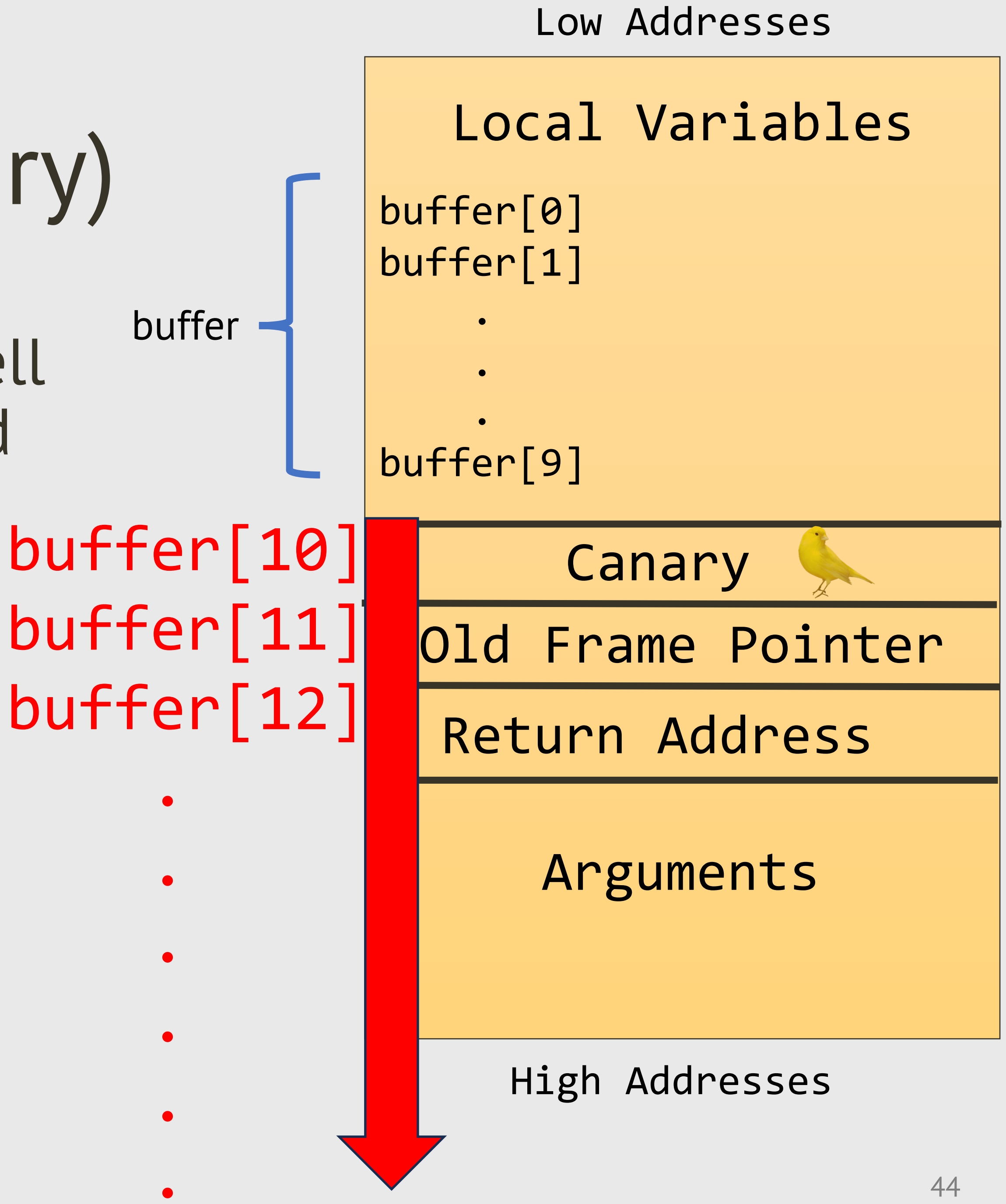
Countermeasures

1. Developer Safeguards
2. Address Space Layout Randomization (ASLR)
3. Non-Executable Stacks
4. Stack Protector (aka. Stack Canary)

Stack Protector (Canary)

- Mark the stack so that we can tell if a buffer overflow has occurred

- Put a random number in the canary spot and if that number is overwritten stop the program

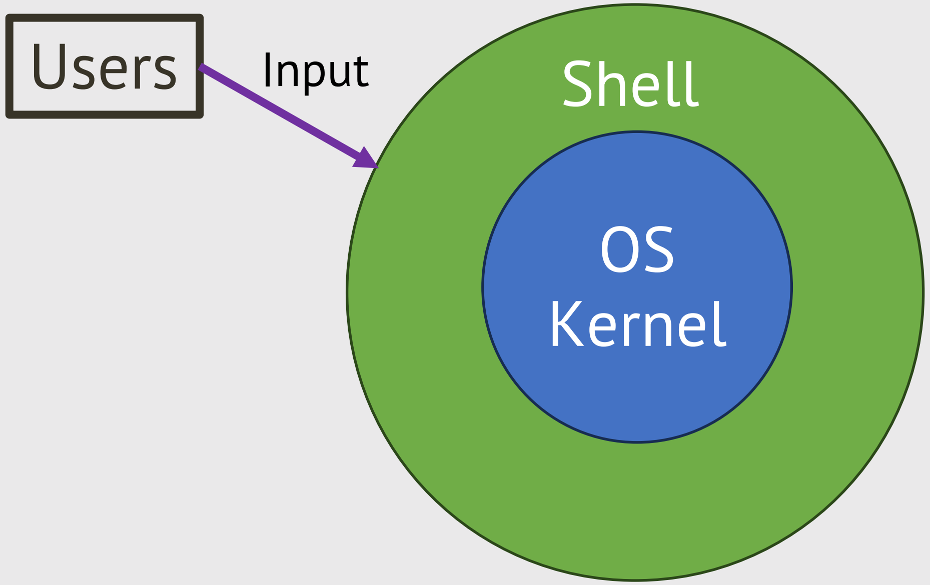


Buffer Overflows

1. How they work
2. Countermeasures
3. Shellcode

What is a shell?

- A **shell** is a computer program that exposes an operating system's services to a human user or other programs
- For example:
  - /bin/sh
  - /bin/csh
  - /bin/bash
  - /bin/zsh

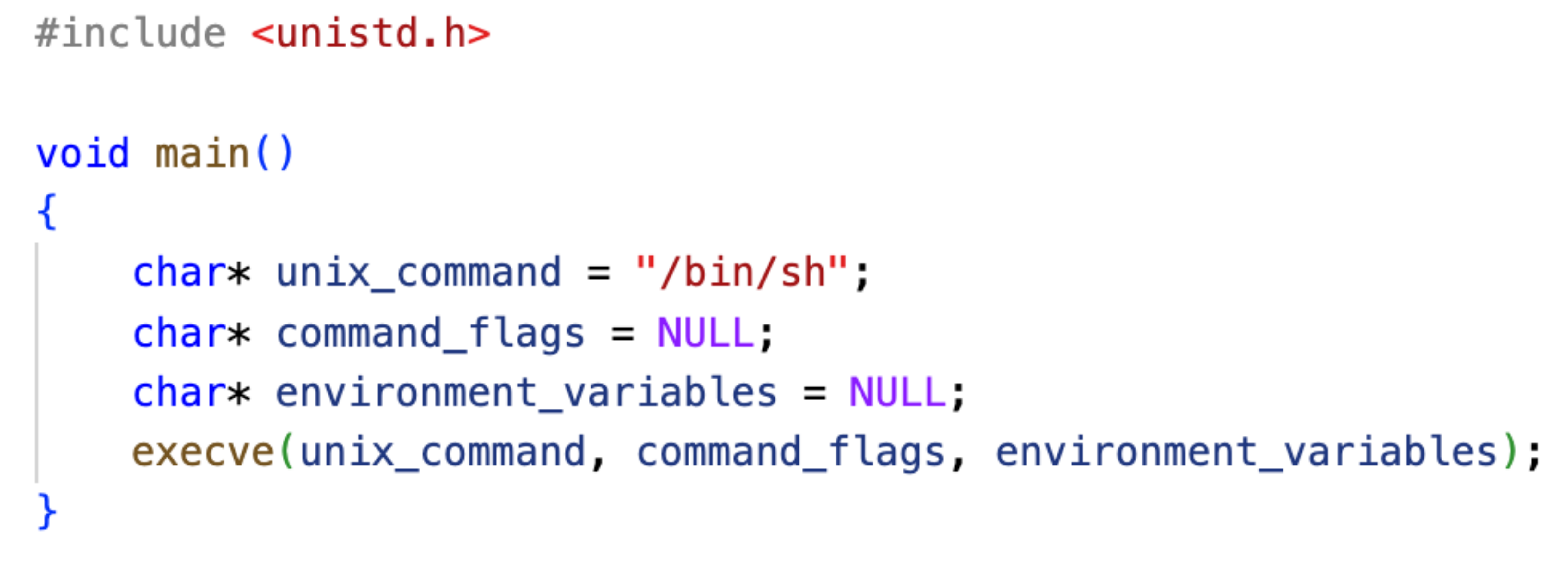


What is shellcode?

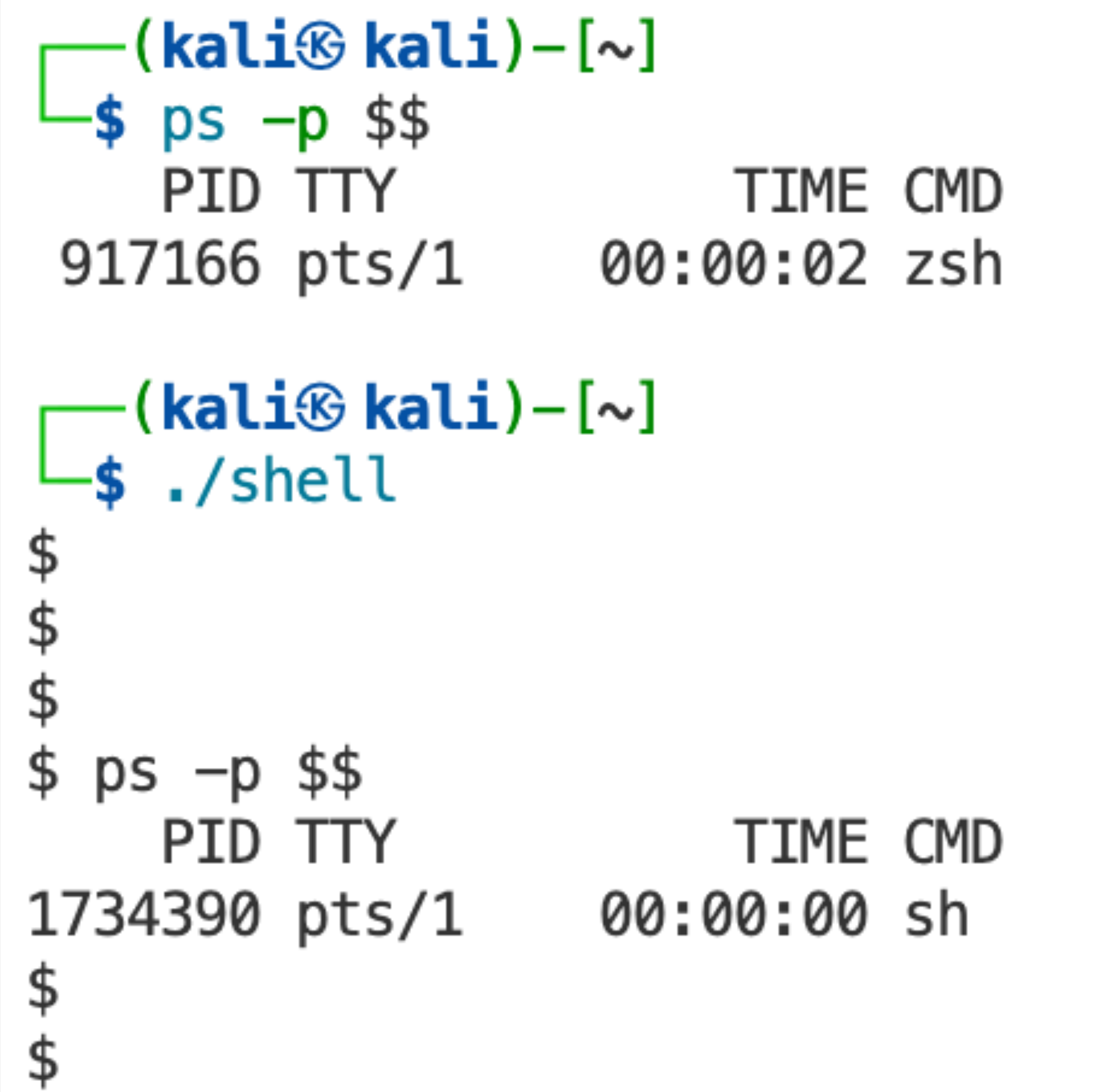
A **shellcode** is a small piece of code used as the payload in the exploitation of a software vulnerability

- It is called *shellcode* because it typically starts a command shell from which the attacker can control the compromised machine

Shell code in C

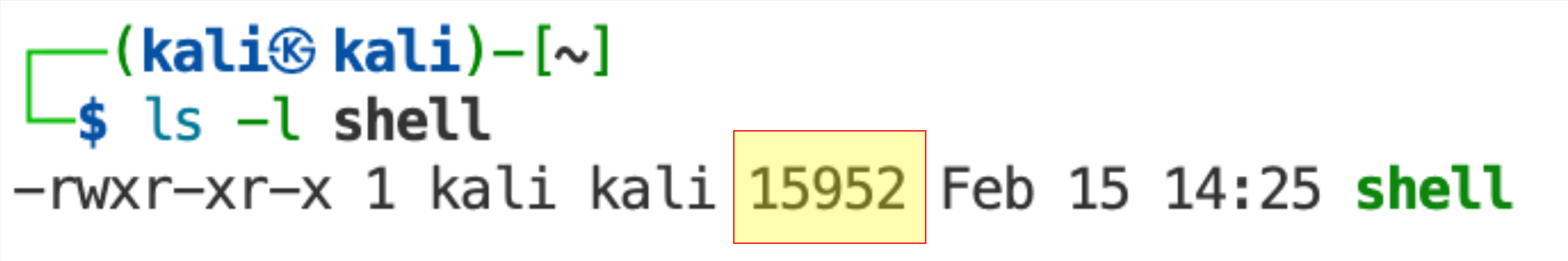


```c
#include <unistd.h>

void main()
{
    char* unix_command = "/bin/sh";
    char* command_flags = NULL;
    char* environment_variables = NULL;
    execve(unix_command, command_flags, environment_variables);
}
```

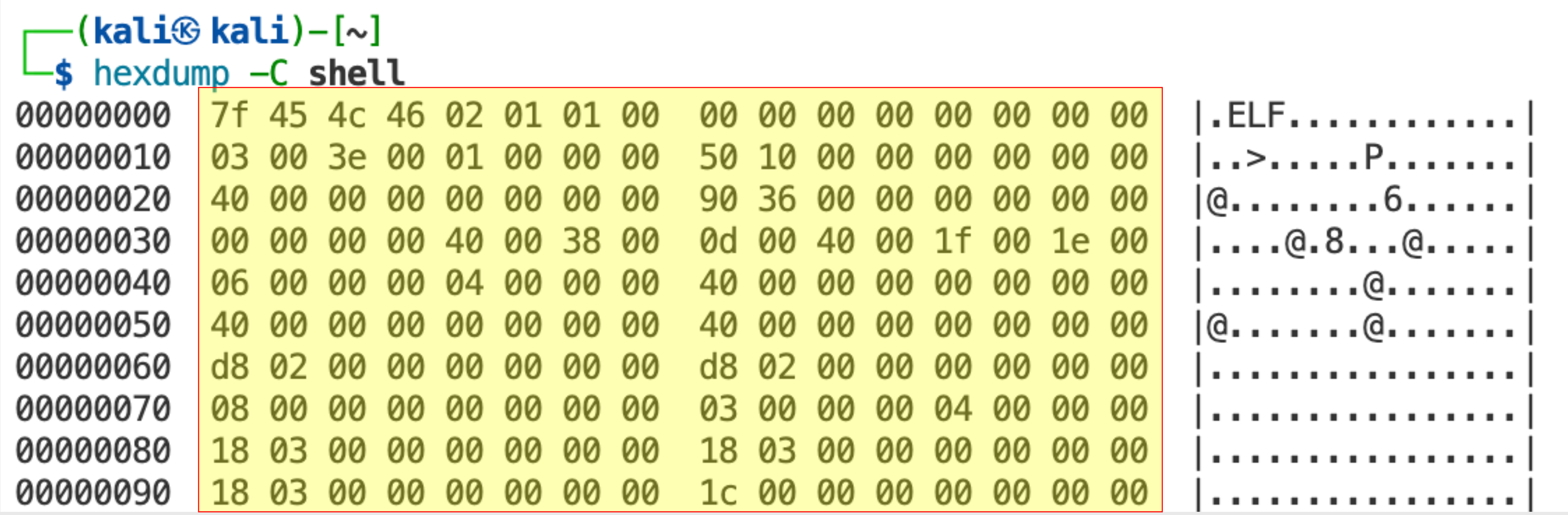


```
┌──(kali㉿kali)-[~]
└─$ ps -p $$
    PID TTY          TIME CMD
 917166 pts/1    00:00:02 zsh

┌──(kali㉿kali)-[~]
└─$ ./shell
$
$
$
$ ps -p $$
    PID TTY          TIME CMD
1734390 pts/1    00:00:00 sh
$
$
```

Problems with trying to use C

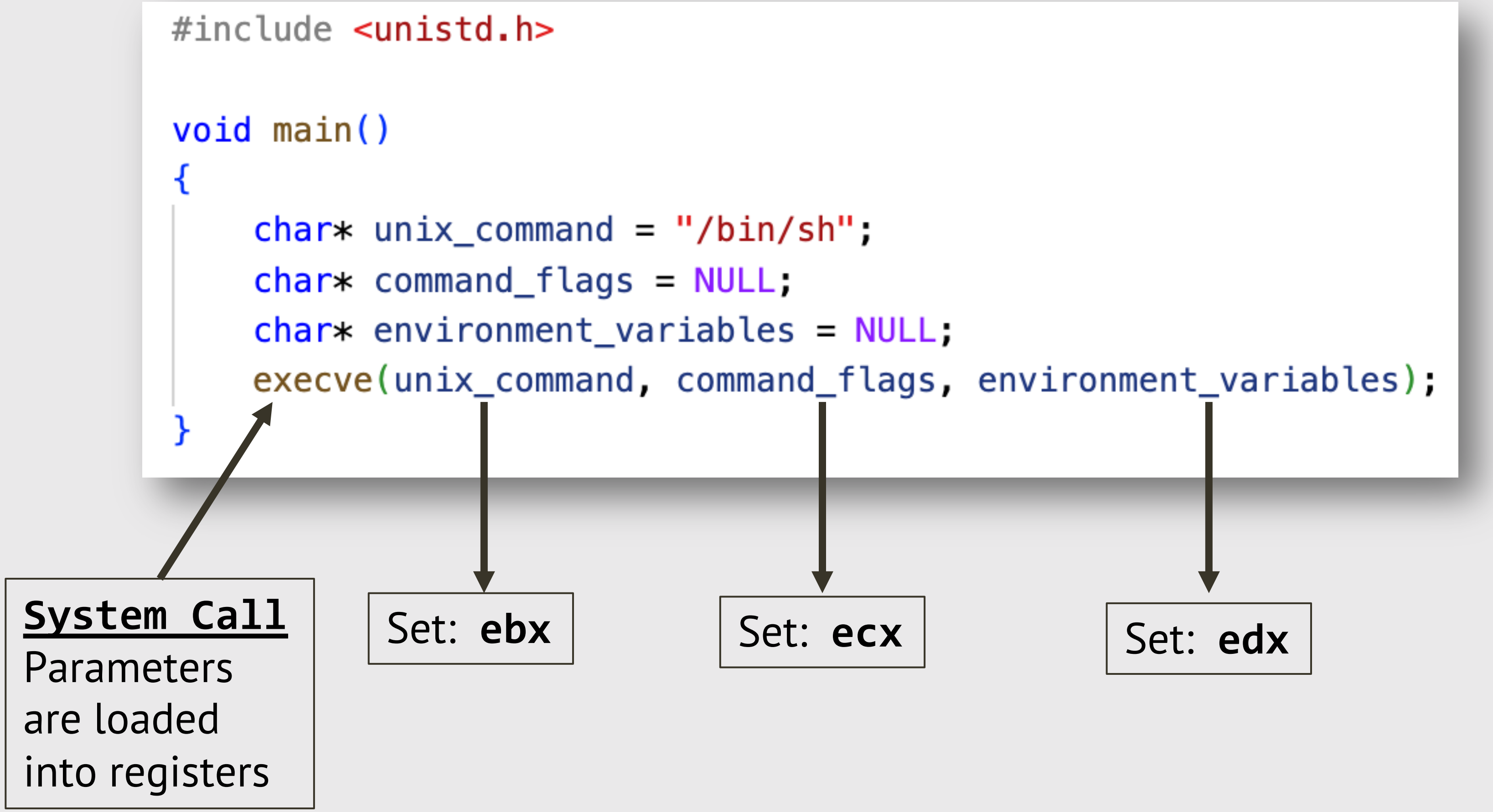


```
┌──(kali㉿kali)-[~]
└─$ ls -l shell
-rwxr-xr-x 1 kali kali 15952 Feb 15 14:25 shell
```



```
┌──(kali㉿kali)-[~]
└─$ hexdump -C shell
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  03 00 3e 00 01 00 00 00  50 10 00 00 00 00 00 00  |..>.....P.......|
00000020  40 00 00 00 00 00 00 00  90 36 00 00 00 00 00 00  |@........6......|
00000030  00 00 00 00 40 00 38 00  0d 00 40 00 1f 00 1e 00  |....@.8...@.....|
00000040  06 00 00 00 04 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000050  40 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |@.......@.......|
00000060  d8 02 00 00 00 00 00 00  d8 02 00 00 00 00 00 00  |................|
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00  |................|
00000080  18 03 00 00 00 00 00 00  18 03 00 00 00 00 00 00  |................|
00000090  18 03 00 00 00 00 00 00  1c 00 00 00 00 00 00 00  |................|
```

Shell code in assembly



```
#include <unistd.h>

void main()
{
    char* unix_command = "/bin/sh";
    char* command_flags = NULL;
    char* environment_variables = NULL;
    execve(unix_command, command_flags, environment_variables);
}
```

System Call
Parameters
are loaded
into registers

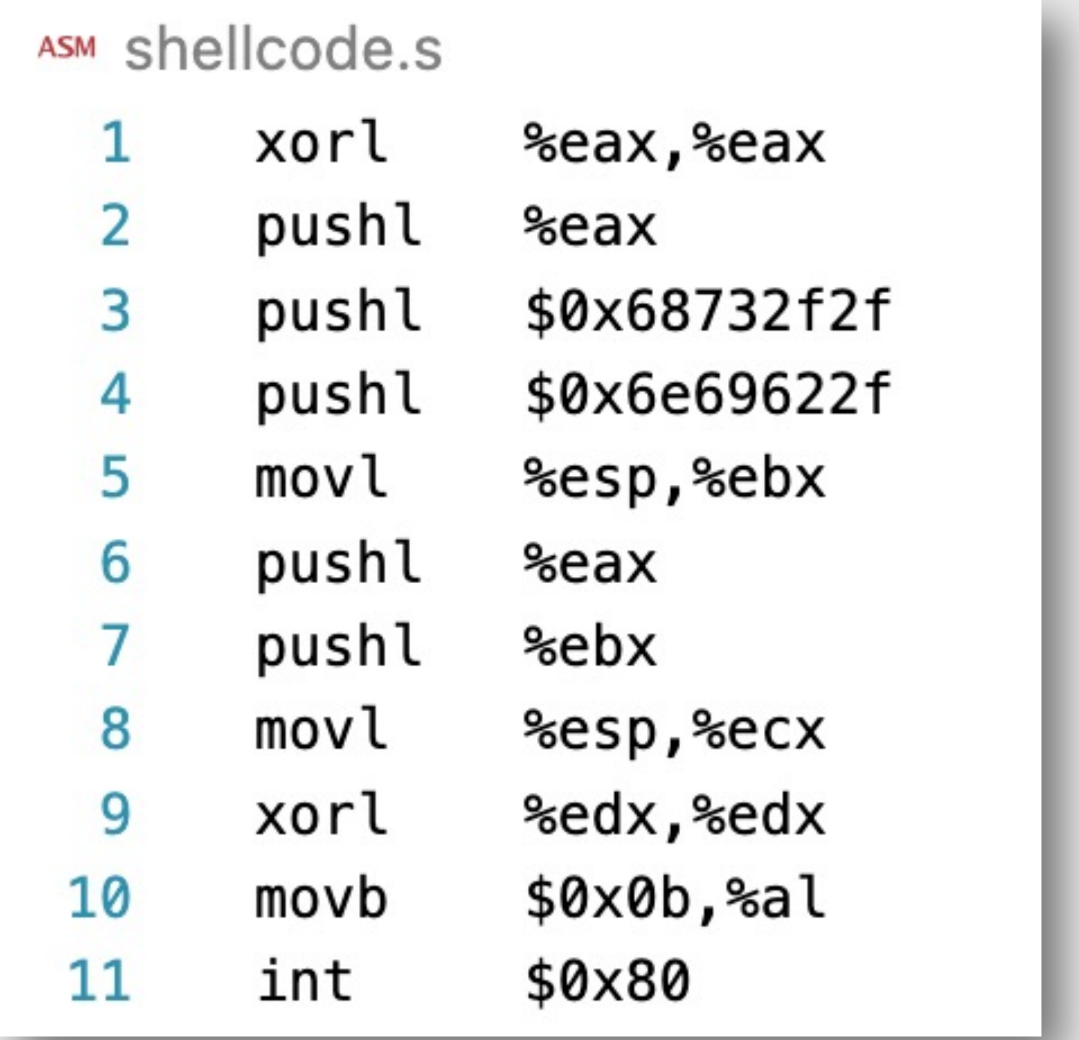
Set: **ebx**

Set: **ecx**

Set: **edx**

In Linux's x86 system call convention, **ebx**, **ecx**, and **edx** are used to pass the first, second, and third arguments to the system call

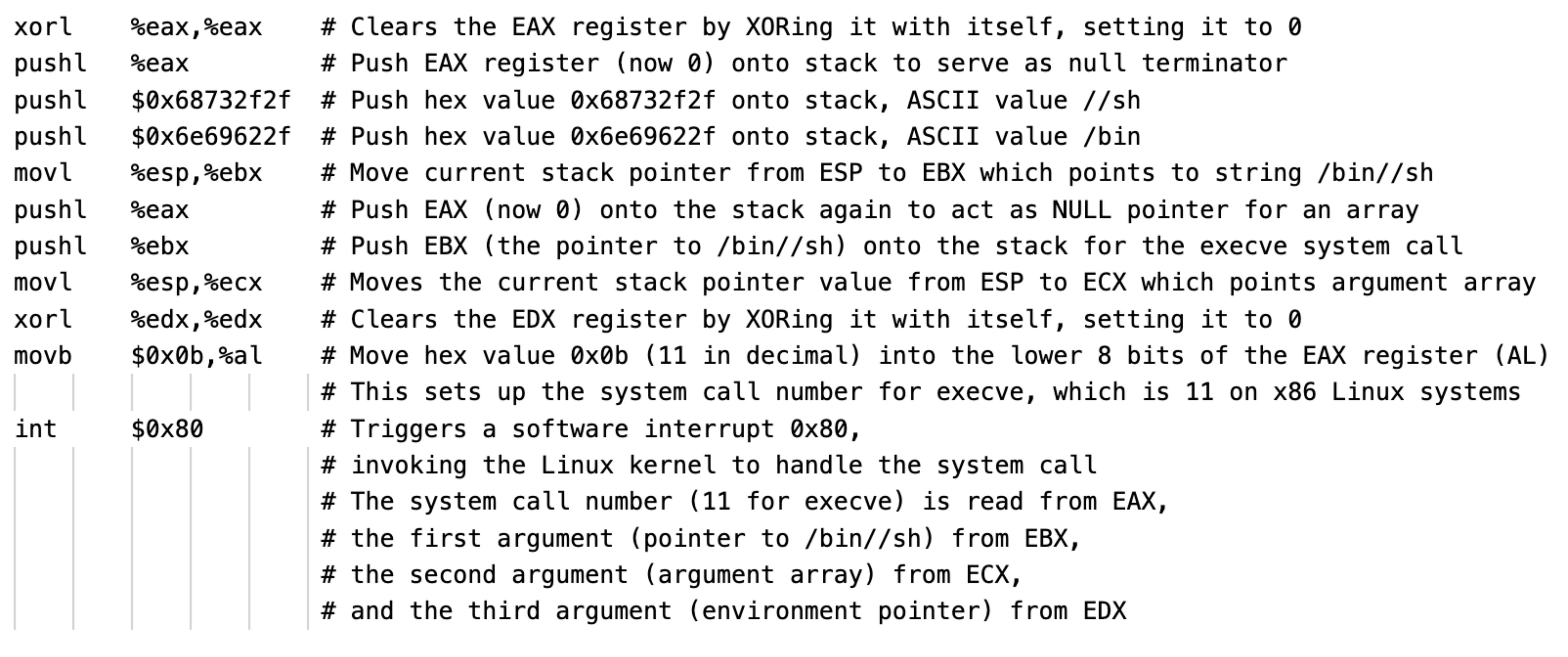
Shell code in assembly



shellcode.s

```
	xorl	%eax,%eax
	pushl	%eax
	pushl	$0x68732f2f
	pushl	$0x6e69622f
	movl	%esp,%ebx
	pushl	%eax
	pushl	%ebx
	movl	%esp,%ecx
	xorl	%edx,%edx
	movb	$0x0b,%al
	int	$0x80
```

Shell code in assembly



```
xorl    %eax,%eax    # Clears the EAX register by XORing it with itself, setting it to 0
pushl   %eax         # Push EAX register (now 0) onto stack to serve as null terminator
pushl   $0x68732f2f  # Push hex value 0x68732f2f onto stack, ASCII value //sh
pushl   $0x6e69622f  # Push hex value 0x6e69622f onto stack, ASCII value /bin
movl    %esp,%ebx    # Move current stack pointer from ESP to EBX which points to string /bin//sh
pushl   %eax         # Push EAX (now 0) onto the stack again to act as NULL pointer for an array
pushl   %ebx         # Push EBX (the pointer to /bin//sh) onto the stack for the execve system call
movl    %esp,%ecx    # Moves the current stack pointer value from ESP to ECX which points argument array
xorl    %edx,%edx    # Clears the EDX register by XORing it with itself, setting it to 0
movb    $0x0b,%al    # Move hex value 0x0b (11 in decimal) into the lower 8 bits of the EAX register (AL)
                     # This sets up the system call number for execve, which is 11 on x86 Linux systems
int     $0x80        # Triggers a software interrupt 0x80,
                     # invoking the Linux kernel to handle the system call
                     # The system call number (11 for execve) is read from EAX,
                     # the first argument (pointer to /bin//sh) from EBX,
                     # the second argument (argument array) from ECX,
                     # and the third argument (environment pointer) from EDX
```