CISC/CMPE 327 Software Quality Assurance

Queen's University, 2019-fall

Lecture #21 Buffer & Heap Overflow

Last week

- Black hat, white hat, and gray hat
- Data vs. Information
- Aspects of data (and example attacks):
  - Integrity
  - Confidentiality
  - Availability
- Security Controls:
  - Technical
  - Physical
  - Administrative
- Types of threats:
  - Adversarial
  - Accidental
  - Structural
  - Environmental

Purpose

- To take control of the target program's execution flow by tricking it into running a piece of malicious or unintended code, **without modifying the program**
  - Changing branches or jump to an arbitrary memory address
- Execution of arbitrary code.
  - Run whatever you would like to do on the target machine.
- Majorly used in system-level exploits.
  - Skipping authentication
  - Jailbreaking your iPhone

Vulnerabilities

- Common Vulnerabilities and Exposures (CVE)
  - Application/Case specific
- Common Vulnerability Scoring System (CVSS)
  - Convery vulnerability severity
- Common Weakness Enumeration (CWE)
  - Dictionary of software vulnerability types

Vulnerability – Heartbleed

A serious vulnerability in the popular OpenSSL cryptographic software library.

- Web servers such as Apache and nginx
  - market share: 66%

CVE – Heartbleed

- CVE-2014-0160
  - The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1_both.c and t1_lib.c, aka the Heartbleed bug.

CVSS – Heartbleed

CVSS Severity and Metrics:

- Base Score: 5.0 MEDIUM
- Impact Subscore: 2.9
- Exploitability Subscore: 10.0

- Access Vector (AV): Network
- Access Complexity (AC): Low
- Authentication (AU): None
- Confidentiality (C): Partial
- Integrity (I): None
- Availability (A): None
- Additional Information: Allows unauthorized disclosure of information

CWE – Heartbleed

- Buffer Errors (CWE-119)

  The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.

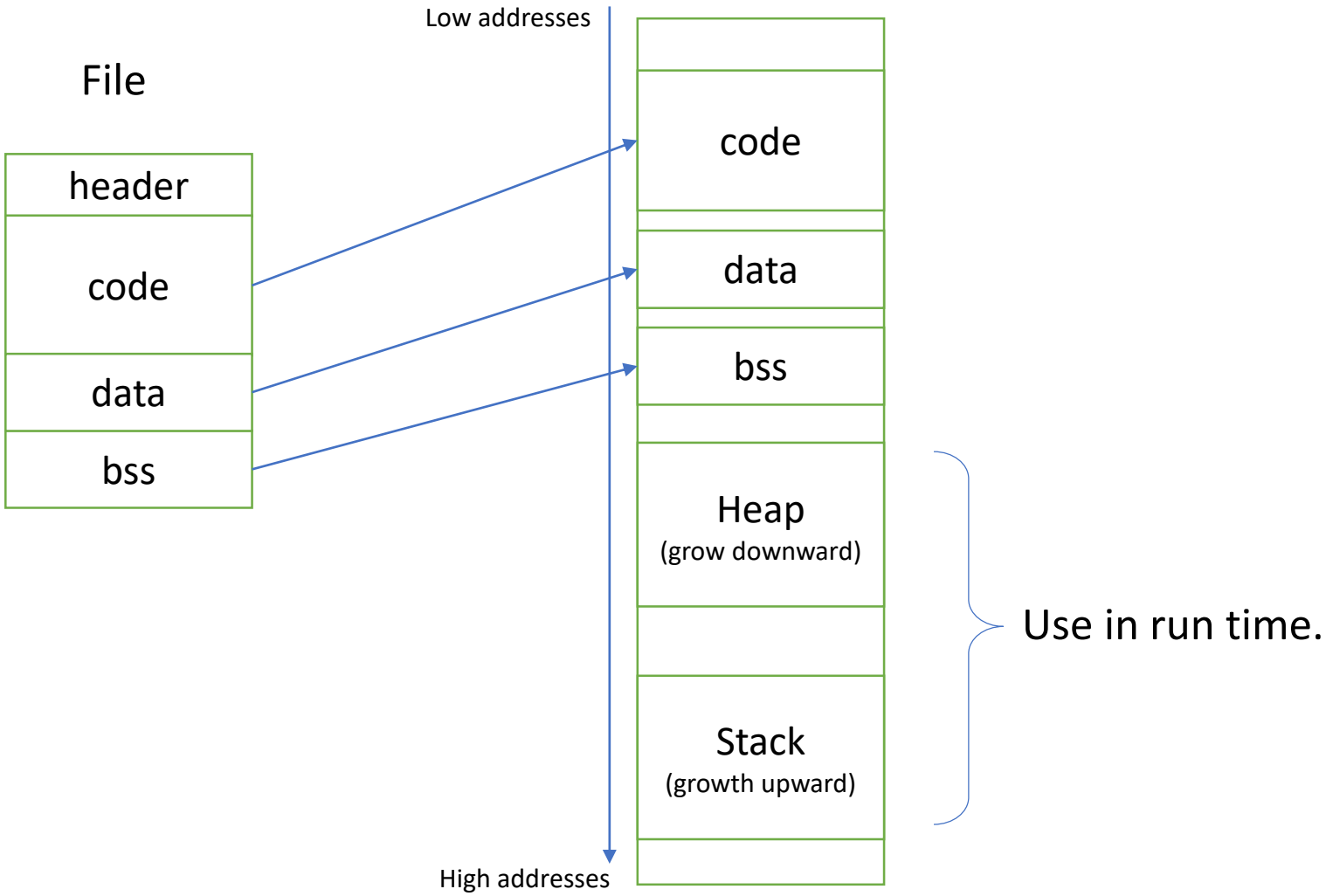
Memory Corruption

Why Memory Corruption

- As an attacker, you do not have the write-access to the targeted binary file.
  - Trying to Jailbreak your iPhone.
  - Trying to gain administrator access as a regular user.
  - Trying to gain administrator access as a malware.
    - The email attachment that you clicked runs as a regular user.
  - Breaking out of your browser.
    - You can't change the browser, but you can change the html and JavaScript code that it executes as input.

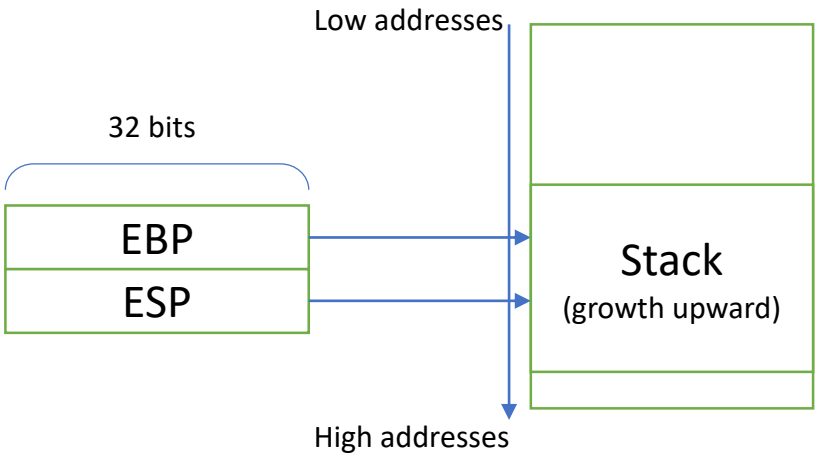
- CVE-2017-11858
  - A buffer overflow vulnerability in the scripting engine used by Microsoft Edge and Internet Explorer *could allow an unauthenticated, remote attacker to execute arbitrary code on a targeted system.*

Memory Address and Layout



Stack

- Variable Size, First-In-Last-Out
- Push -> put something on the top of the stack.
- Pop -> Retrieve and ‘remove’ something from the top of the stack.
- ESP - Stack pointer, “top” of the stack frame. (lower memory address)
- EBP - Base pointer, “bottom” of the stack frame. (higher memory address)



Stack

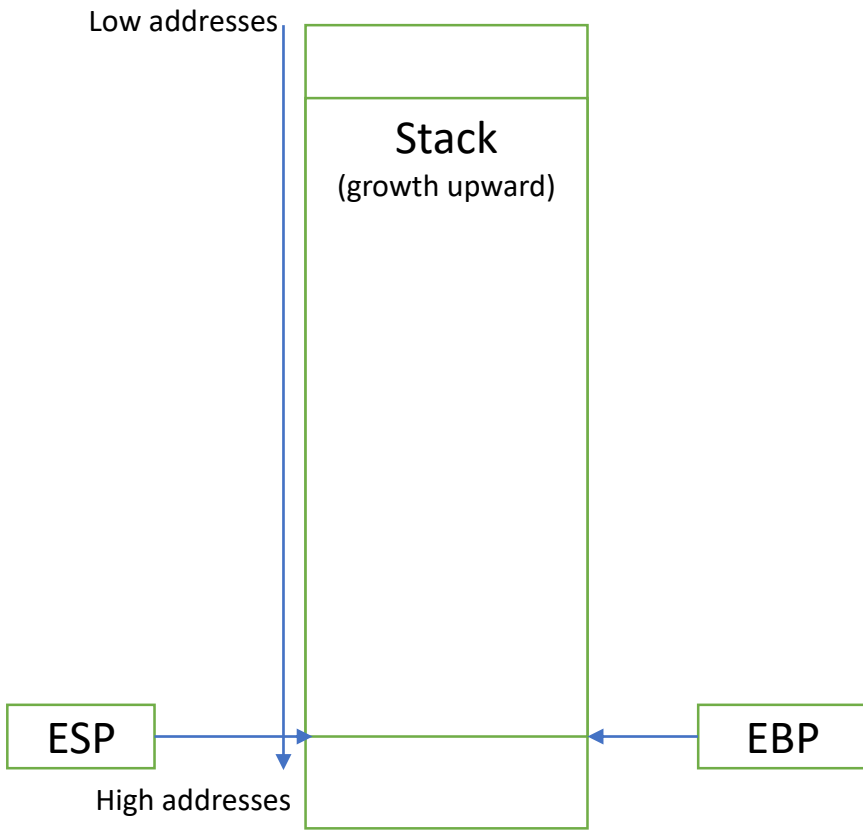
- Original

EAX = 0x00000001

EBX = 0x00000002

ECX = 0xFFFFFFFFF

EAX	0x00000001
EBX	0x00000002
ECX	0xFFFFFFFFF



Push EAX

Push EBX

Pop ECX

Pop EBX

Stack

- Original

EAX = 0x00000001

EBX = 0x00000002

ECX = 0xFFFFFFFFF

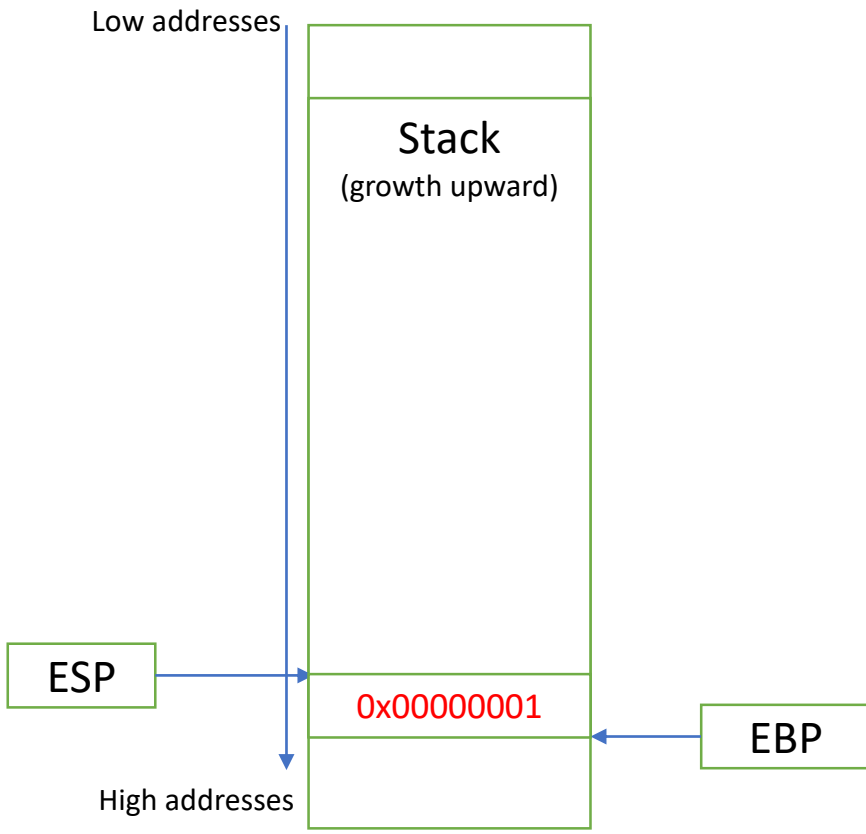
EAX	0x00000001
EBX	0x00000002
ECX	0xFFFFFFFFF

Push EAX

Push EBX

Pop ECX

Pop EBX



Stack

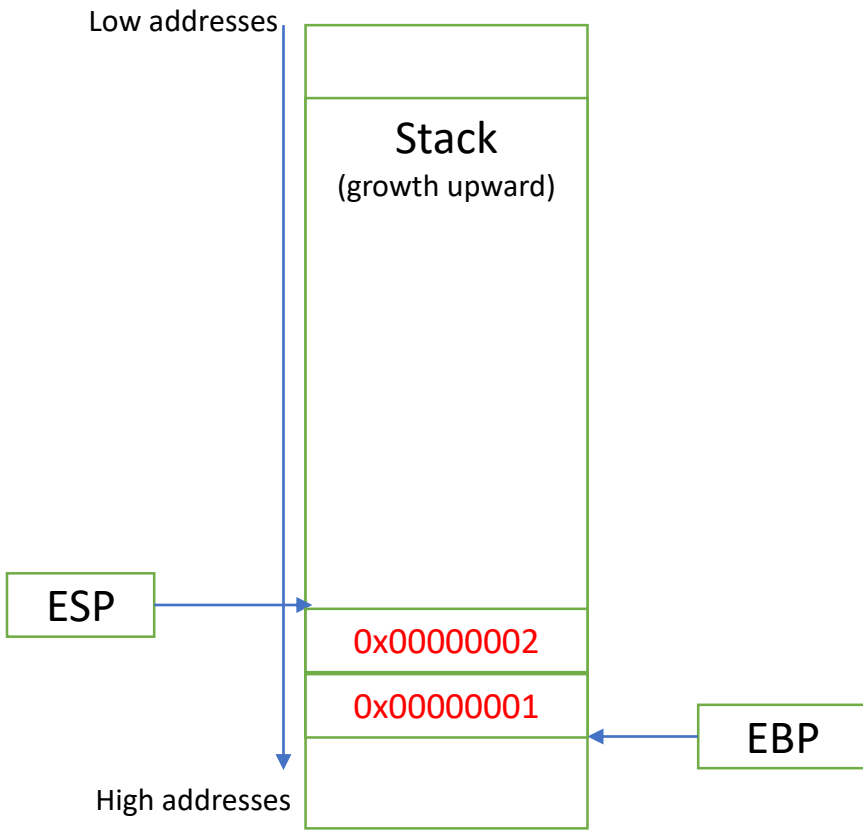
- Original

EAX = 0x00000001

EBX = 0x00000002

ECX = 0xFFFFFFFFF

EAX	0x00000001
EBX	0x00000002
ECX	0xFFFFFFFFF



Push EAX

Push EBX

Pop ECX

Pop EBX

Stack

- Original

EAX = 0x00000001

EBX = 0x00000002

ECX = 0xFFFFFFFF

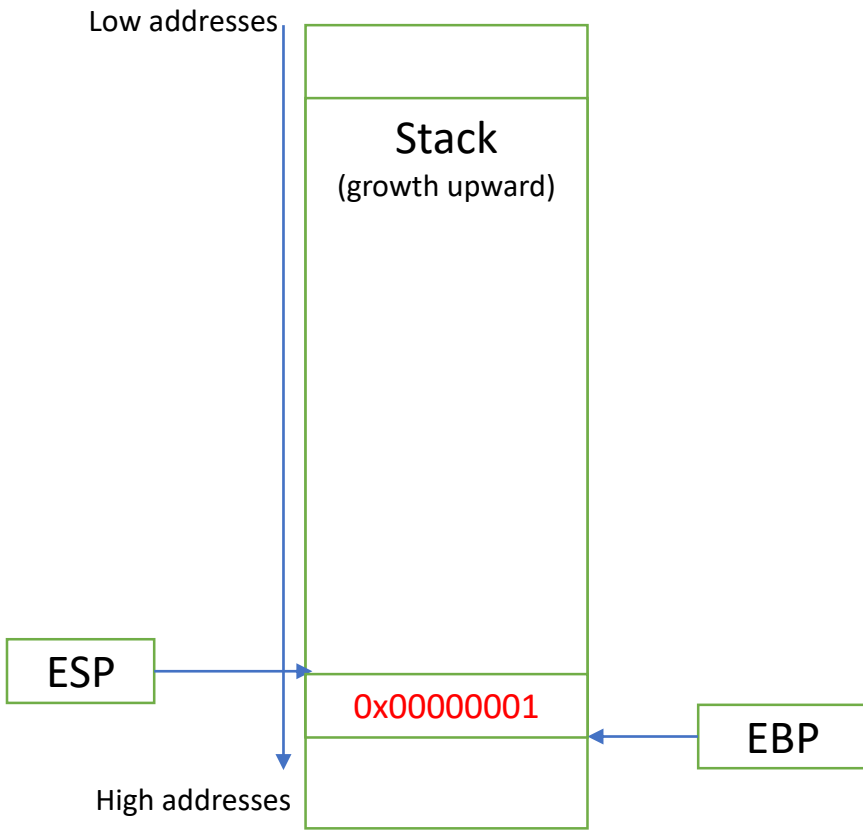
EAX	0x00000001
EBX	0x00000002
ECX	0x00000002

Push EAX

Push EBX

Pop ECX

Pop EBX



Stack

- Original

EAX = 0x00000001

EBX = 0x00000002

ECX = 0xFFFFFFFF

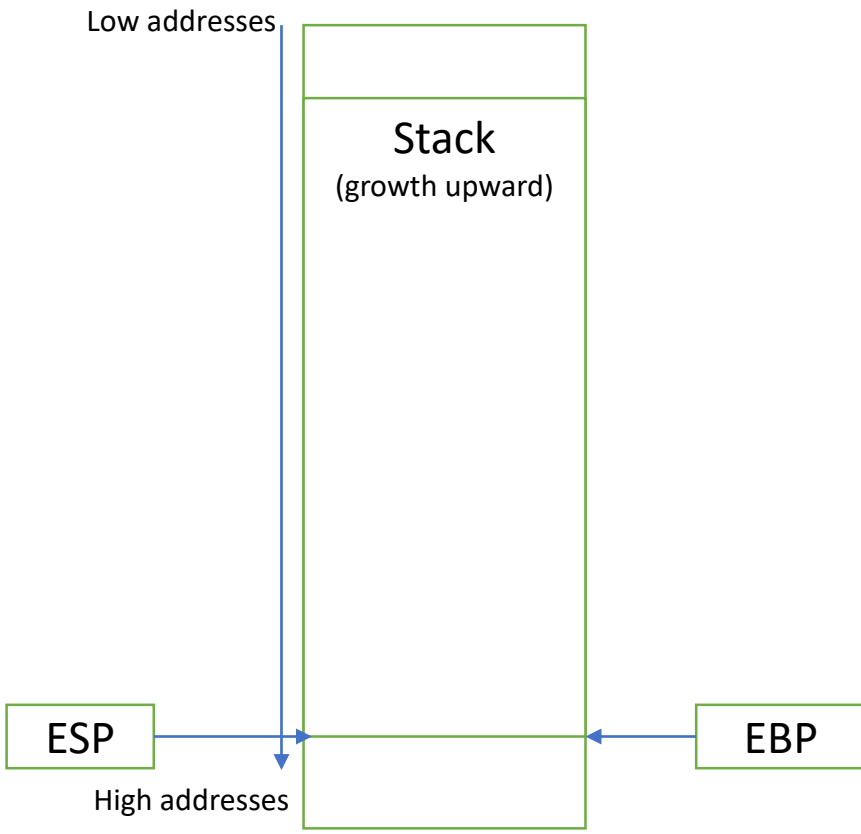
EAX	0x00000001
EBX	0x00000001
ECX	0x00000002

Push EAX

Push EBX

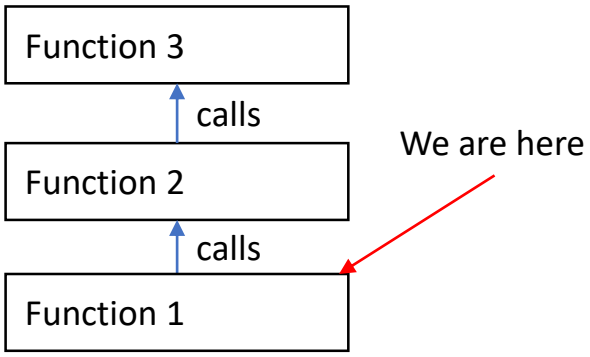
Pop ECX

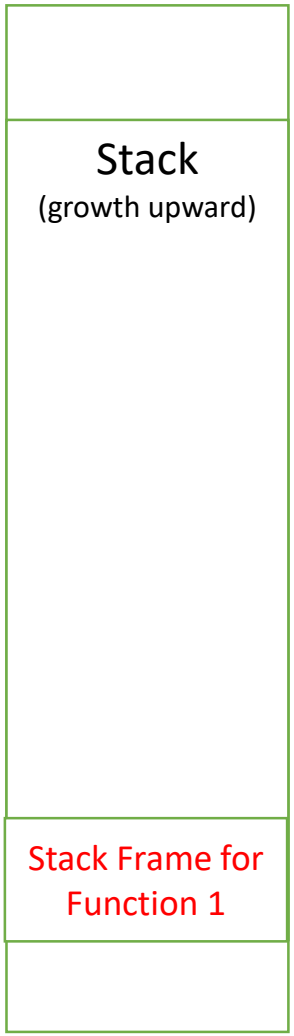
Pop EBX



Stack Use

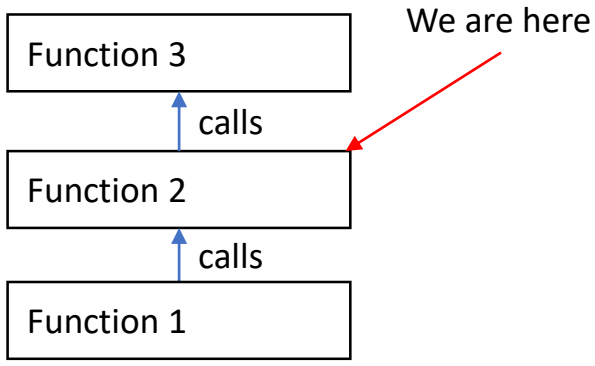
- The stack is used as a temporary scratch pad to store local function variables and context during function calls.
- Stack Frame:
  - Part of the stack segment that is dedicated to be used by a specific function to store:
  - Function arguments
  - Return address
  - Local variables

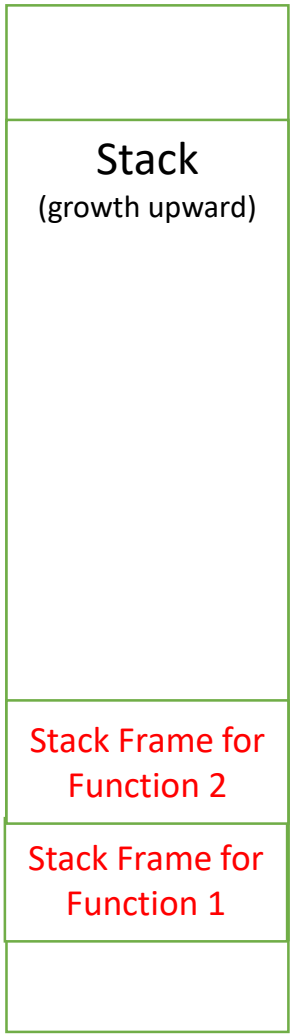




Stack Use

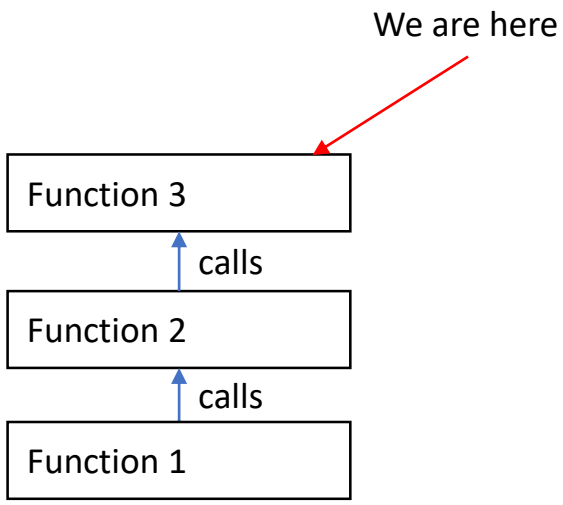
- The stack is used as a temporary scratch pad to store local function variables and context during function calls.
- Stack Frame:
  - Part of the stack segment that is dedicated to be used by a specific function to store:
  - Function arguments
  - Return address
  - Local variables

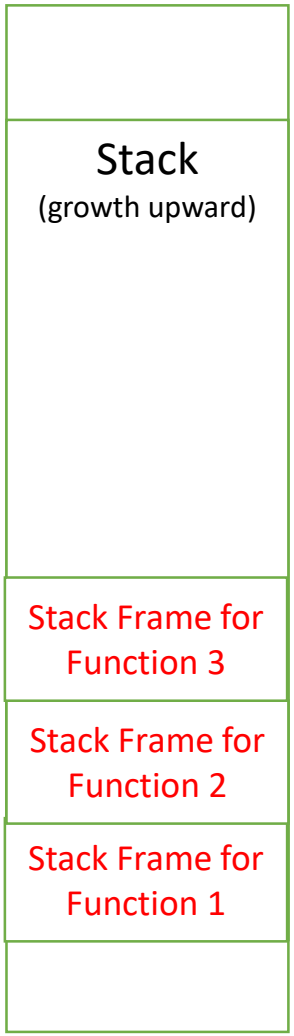




Stack Use

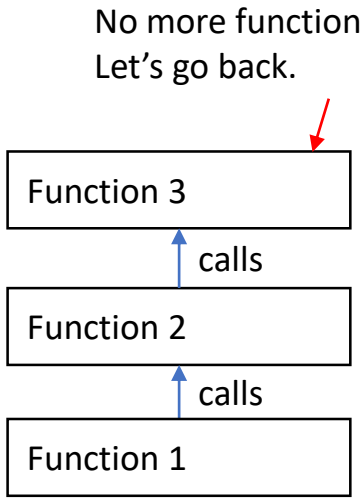
- The stack is used as a temporary scratch pad to store local function variables and context during function calls.
- Stack Frame:
  - Part of the stack segment that is dedicated to be used by a specific function to store:
  - Function arguments
  - Return address
  - Local variables

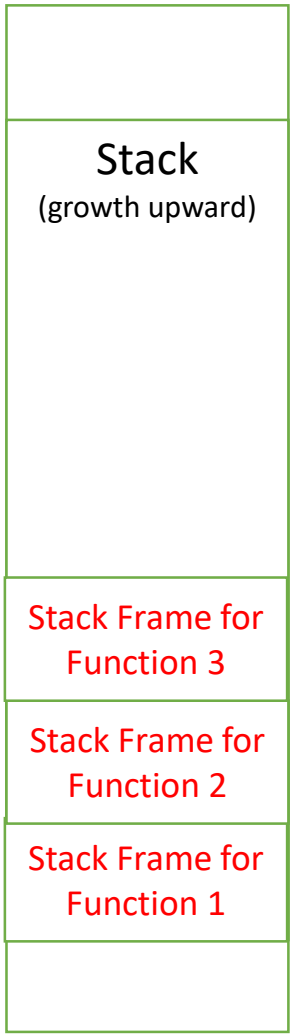




Stack Use

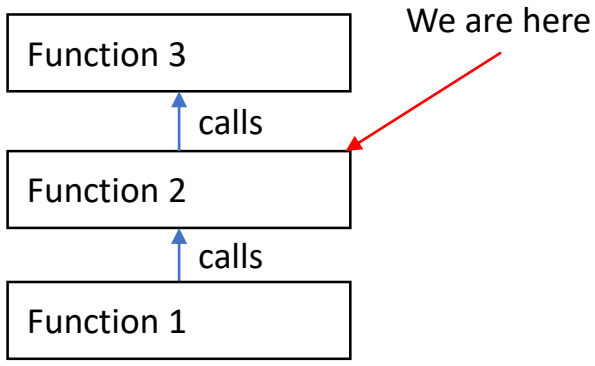
- The stack is used as a temporary scratch pad to store local function variables and context during function calls.
- Stack Frame:
  - Part of the stack segment that is dedicated to be used by a specific function to store:
  - Function arguments
  - Return address
  - Local variables

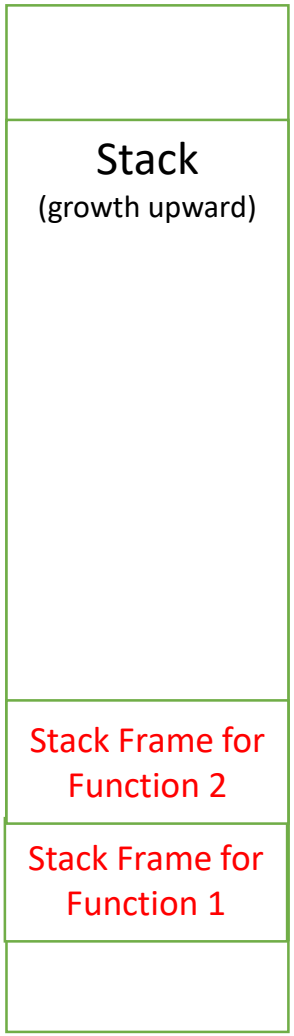




Stack Use

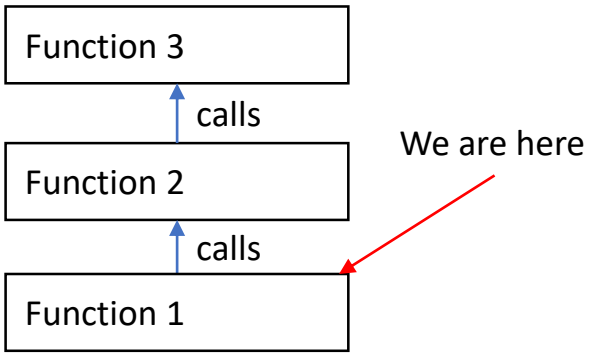
- The stack is used as a temporary scratch pad to store local function variables and context during function calls.
- Stack Frame:
  - Part of the stack segment that is dedicated to be used by a specific function to store:
  - Function arguments
  - Return address
  - Local variables

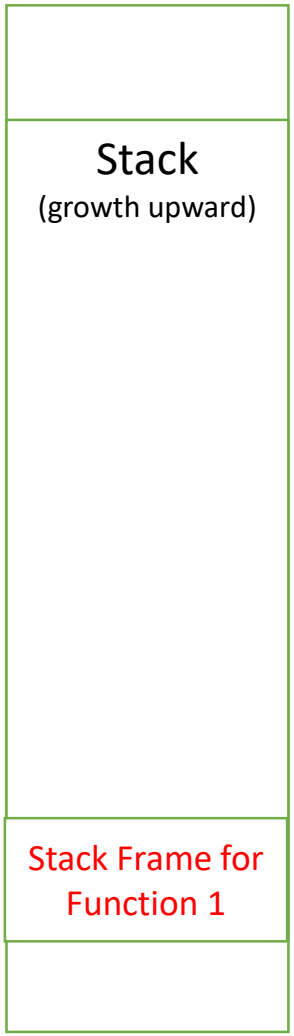




Stack Use

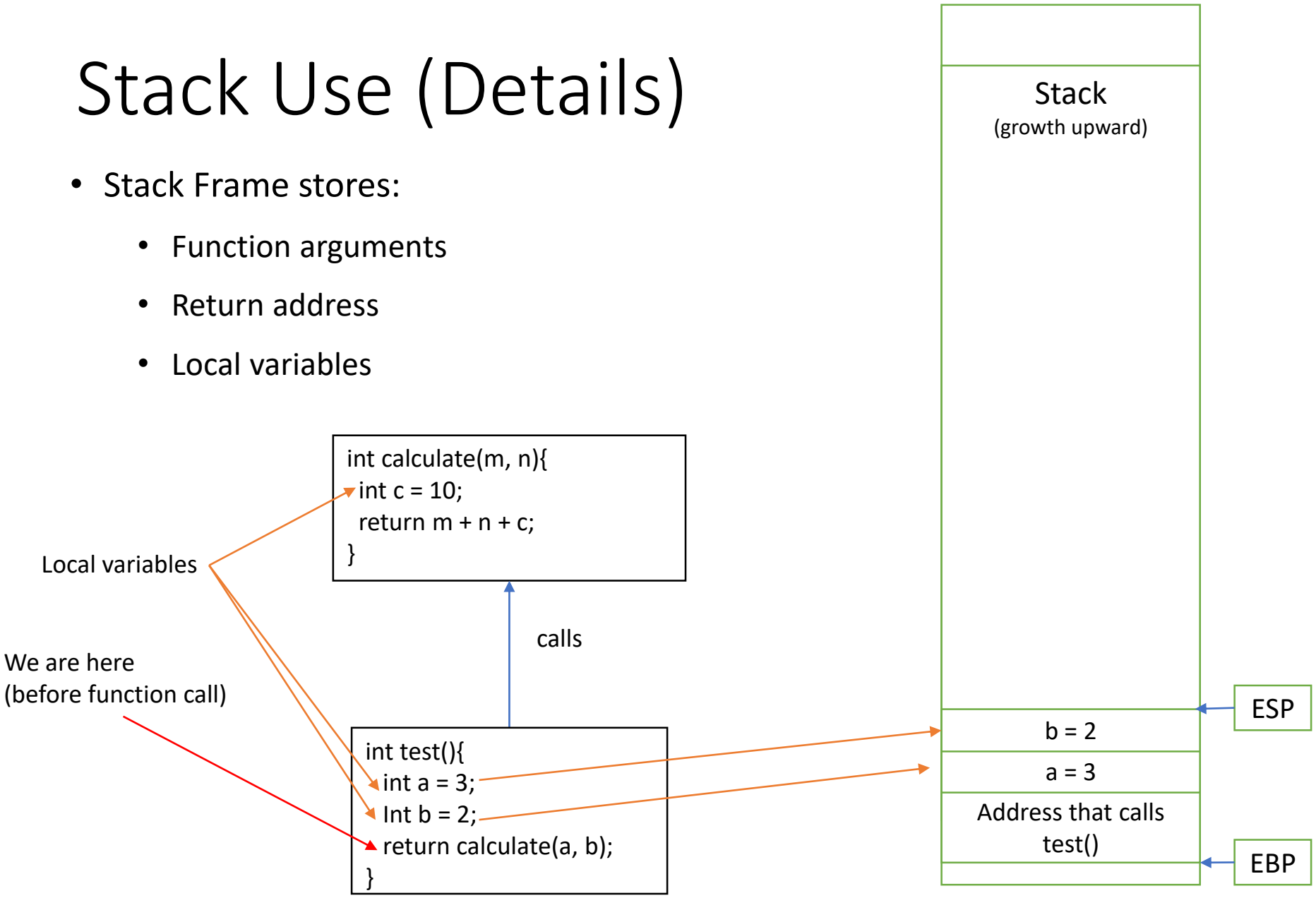
- The stack is used as a temporary scratch pad to store local function variables and context during function calls.
- Stack Frame:
  - Part of the stack segment that is dedicated to be used by a specific function to store:
  - Function arguments
  - Return address
  - Local variables





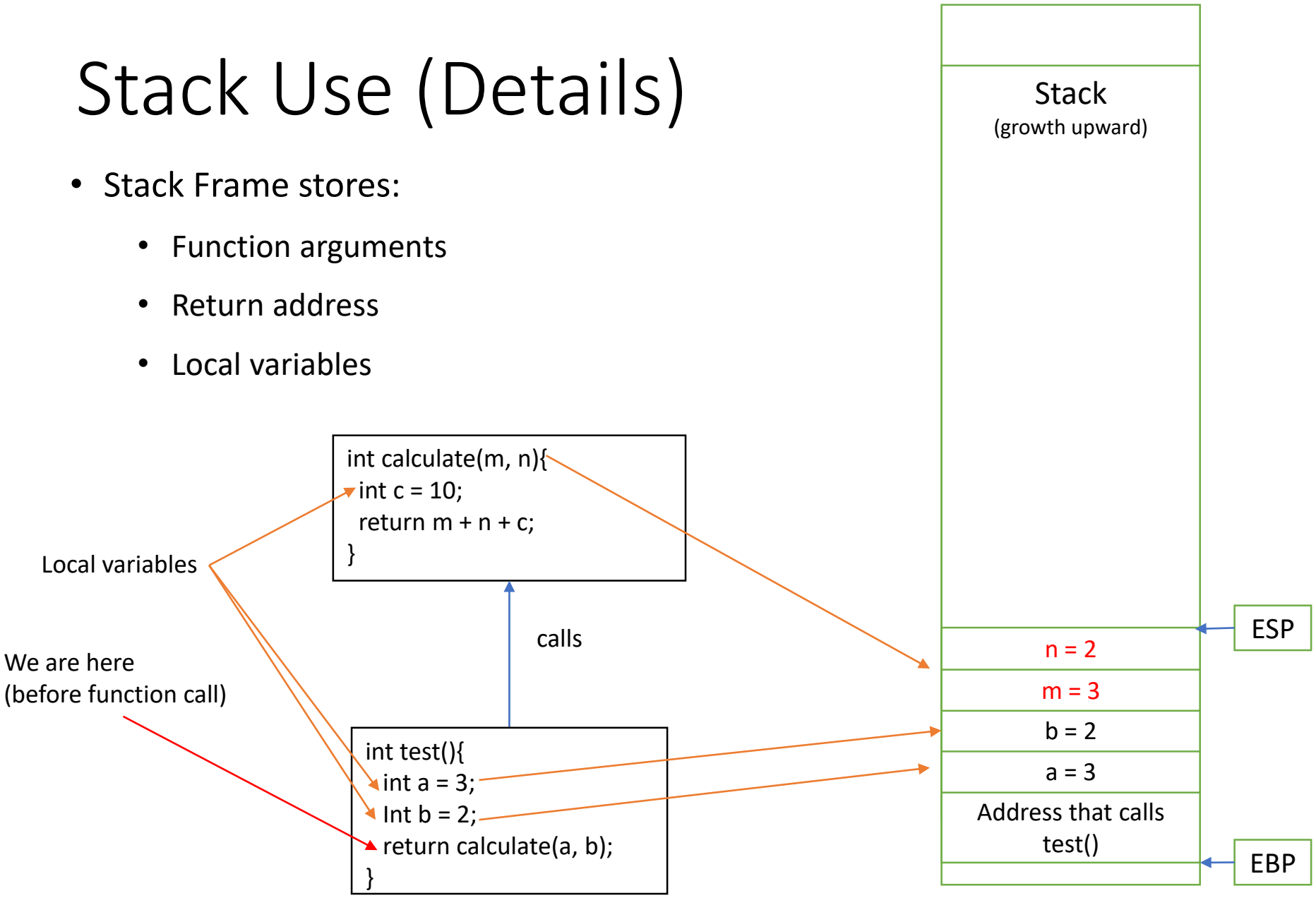
Stack Use (Details)

- Stack Frame stores:
  - Function arguments
  - Return address
  - Local variables



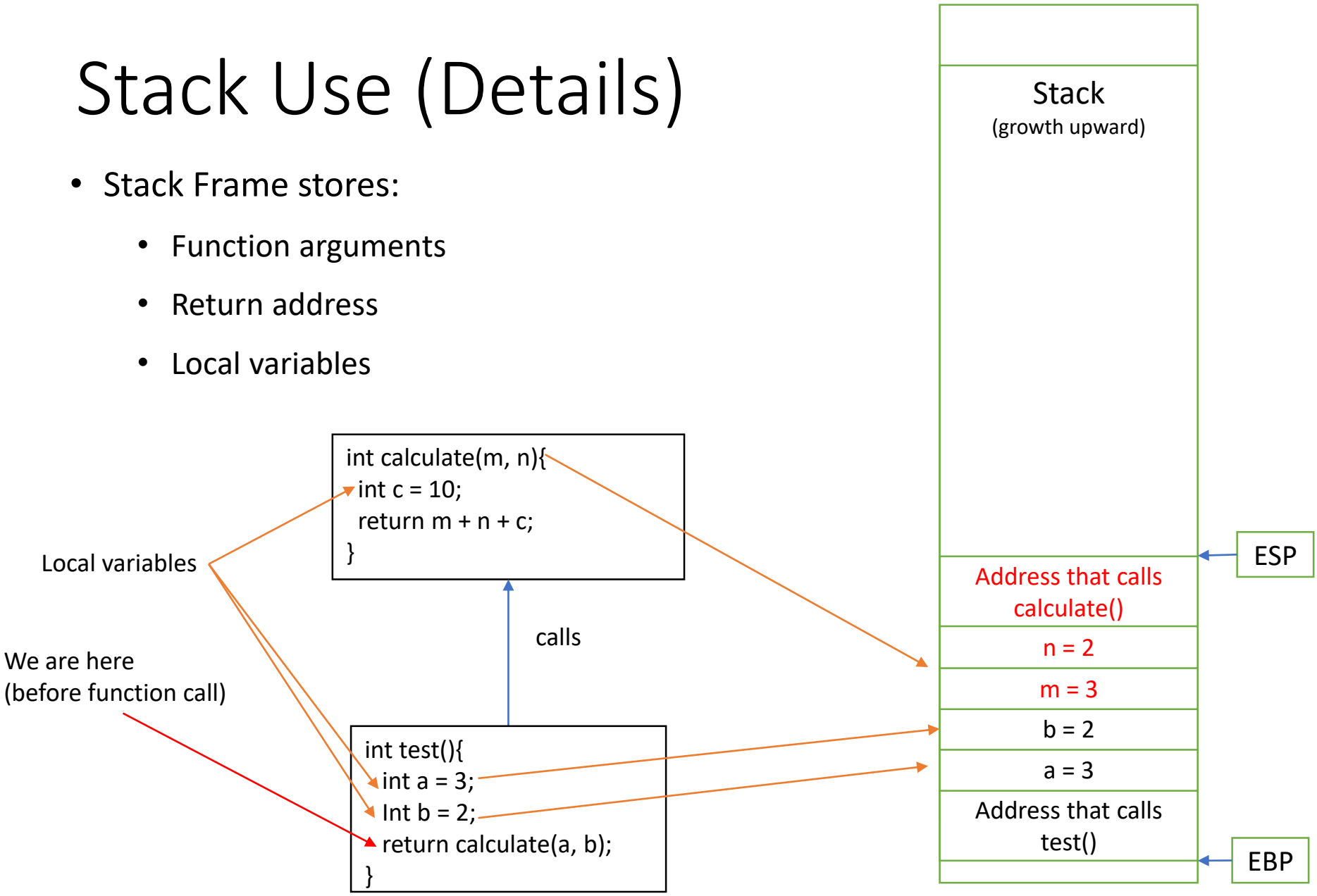
Stack Use (Details)

- Stack Frame stores:
  - Function arguments
  - Return address
  - Local variables



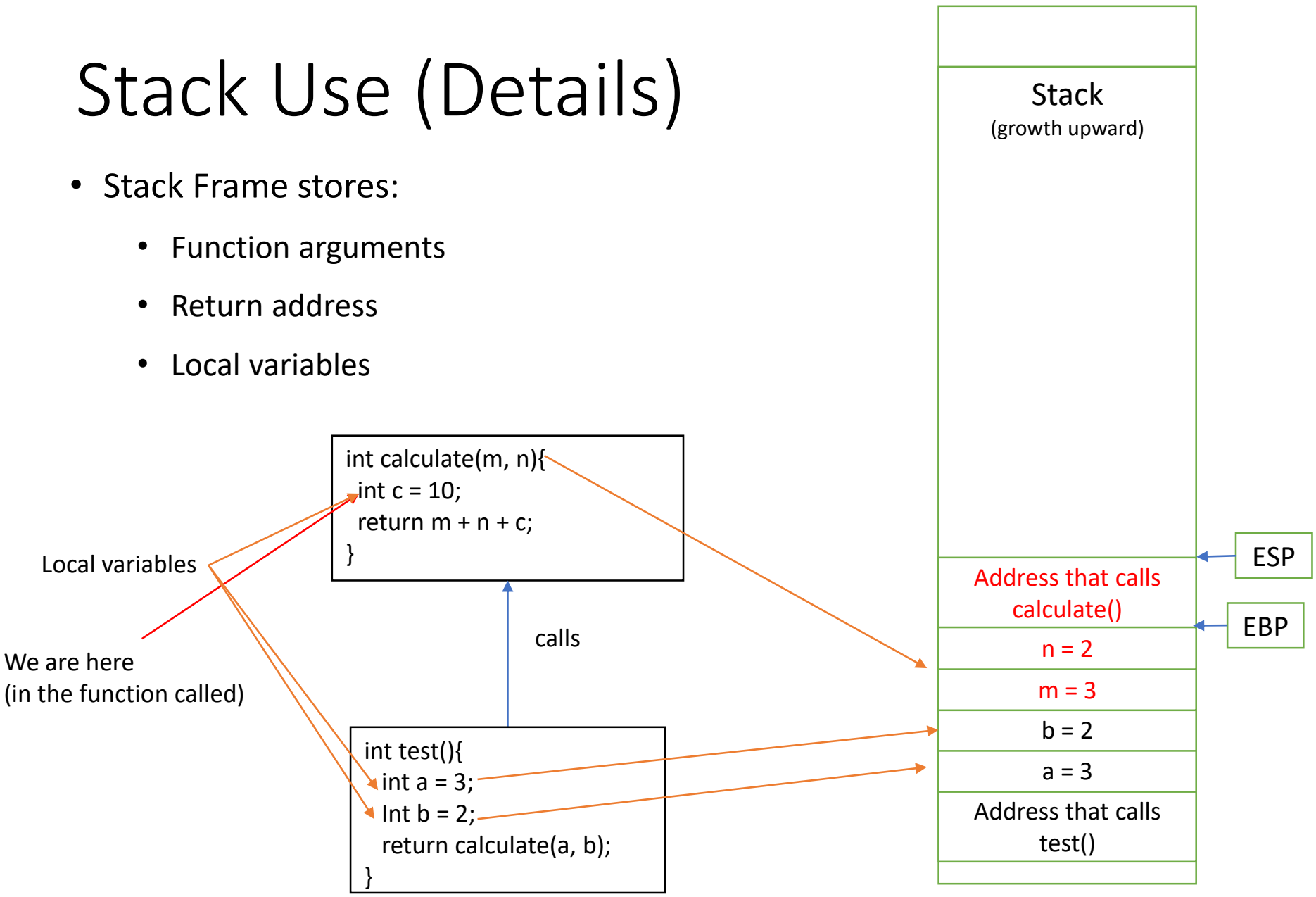
Stack Use (Details)

- Stack Frame stores:
  - Function arguments
  - Return address
  - Local variables



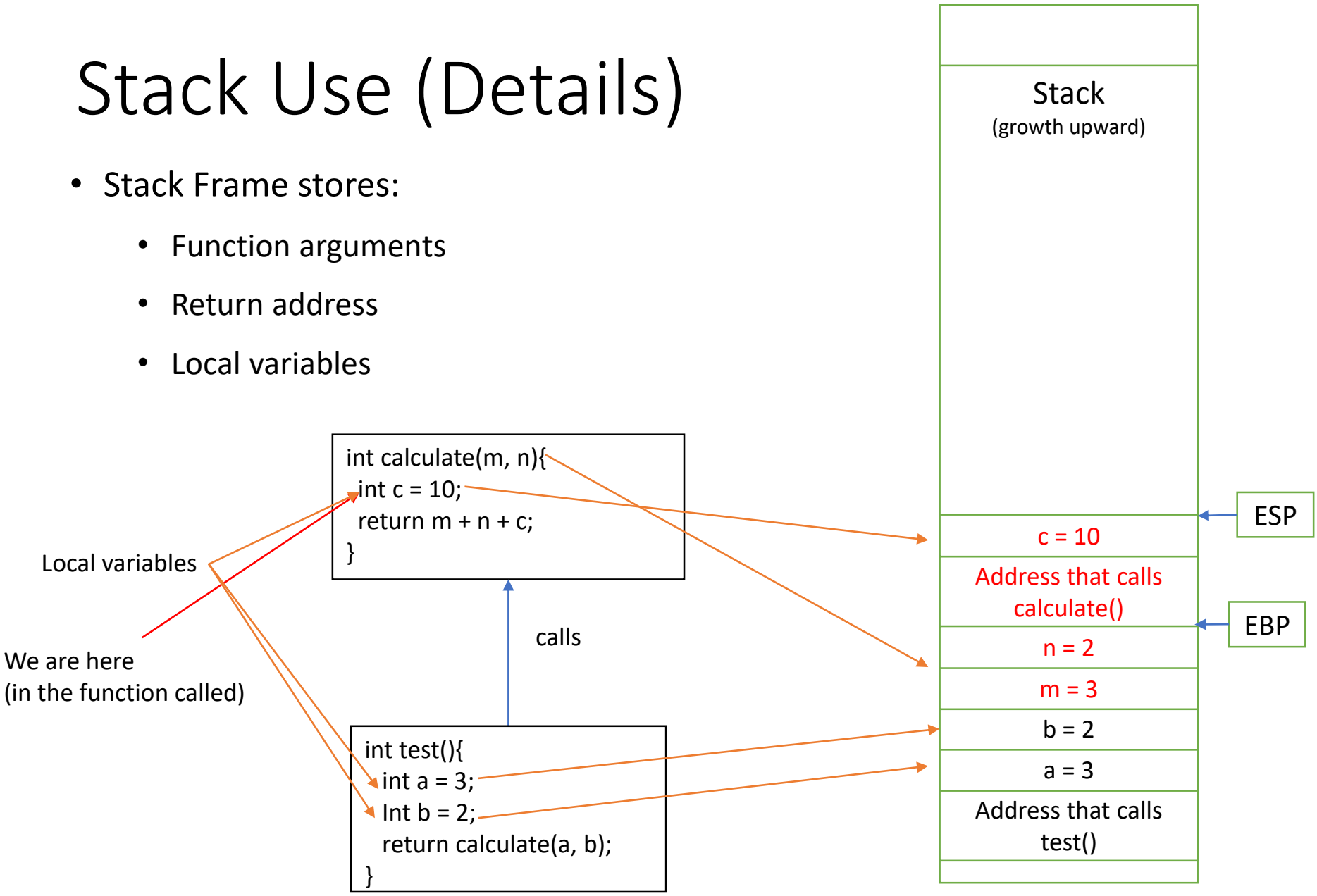
Stack Use (Details)

- Stack Frame stores:
  - Function arguments
  - Return address
  - Local variables



Stack Use (Details)

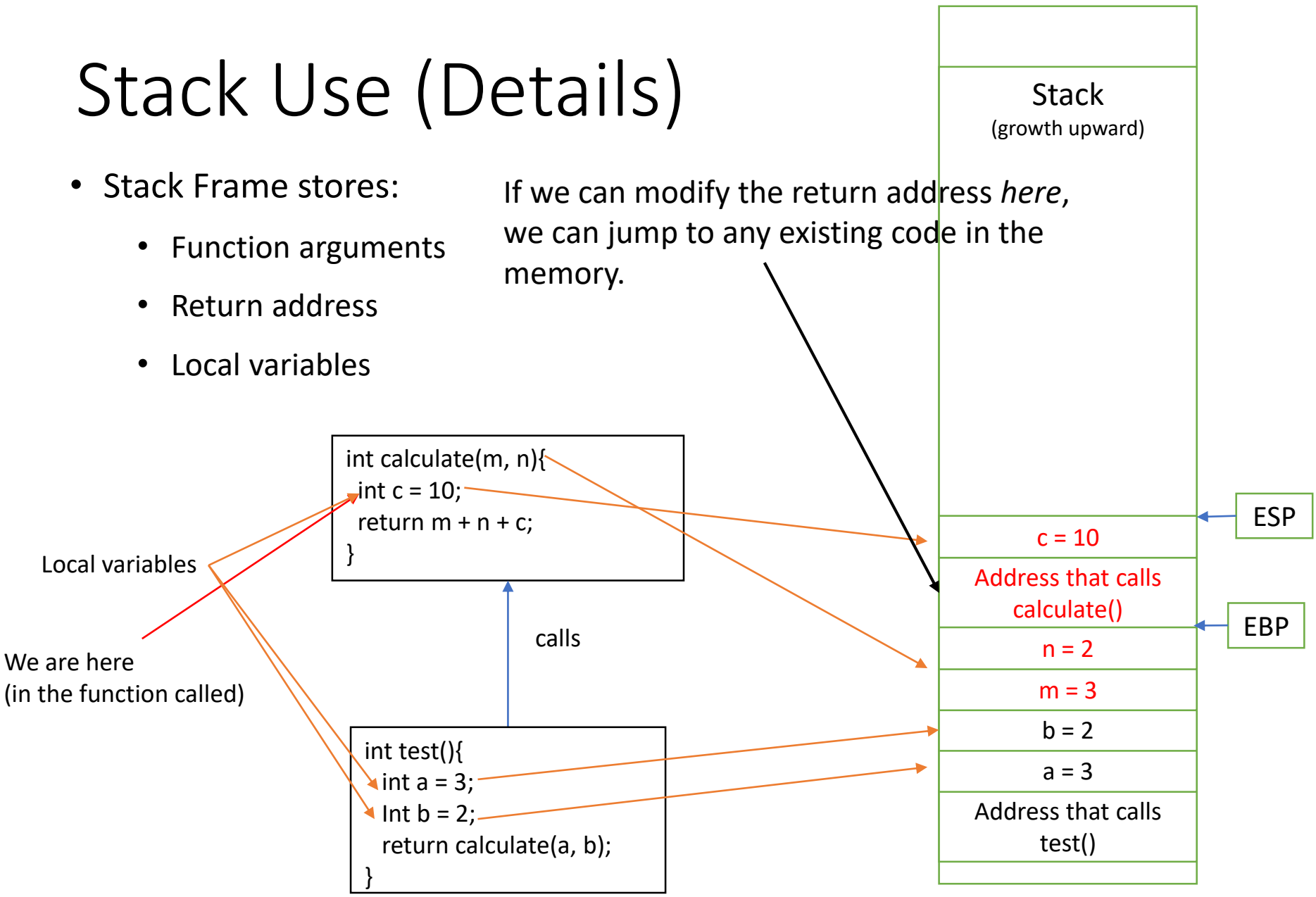
- Stack Frame stores:
  - Function arguments
  - Return address
  - Local variables



Stack Use (Details)

- Stack Frame stores:
  - Function arguments
  - Return address
  - Local variables

If we can modify the return address *here*, we can jump to any existing code in the memory.



Buffer overflow (exploit)

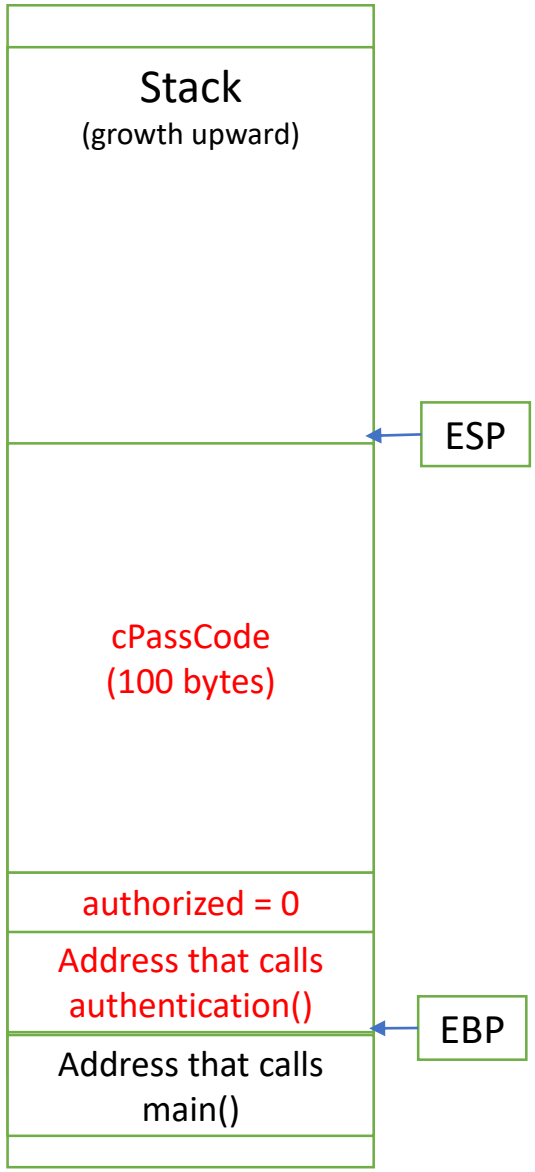
```
#include <stdio.h>
#include <string.h>

int authentication(){
   int authorized = 0;
   char cPassCode[100];
   scanf("%s", cPassCode);
   if (!strcmp(cPassCode, "mcgill-cby02"))
      authorized = 1;

   return authorized;
}

int main()
{
   if (authentication())
      printf("Successfully logged in!");
   else
      printf("Wrong password.");
   return 0;
}
```

We are here in this function



Buffer overflow (exploit)

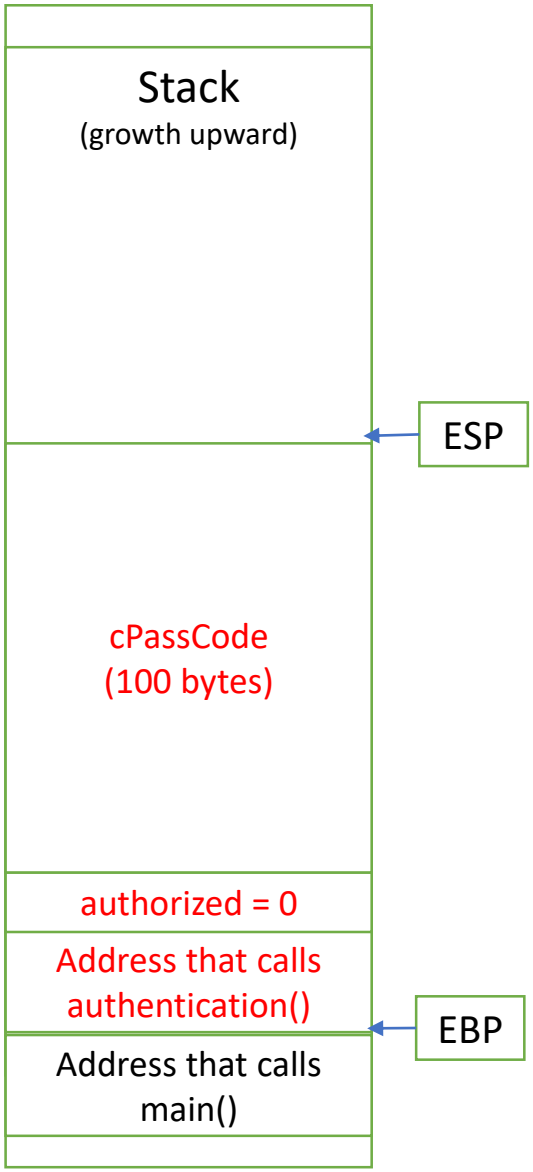
```
#include <stdio.h>
#include <string.h>

int authentication(){
   int authorized = 0;
   char cPassCode[100];
   scanf("%s", cPassCode);
   if (!strcmp(cPassCode, "mcgill-cby02"))
      authorized = 1;

   return authorized;
}

int main()
{
   if (authentication())
      printf("Successfully logged in!");
   else
      printf("Wrong password.");
   return 0;
}
```

0x41 repeated 101 times as input.



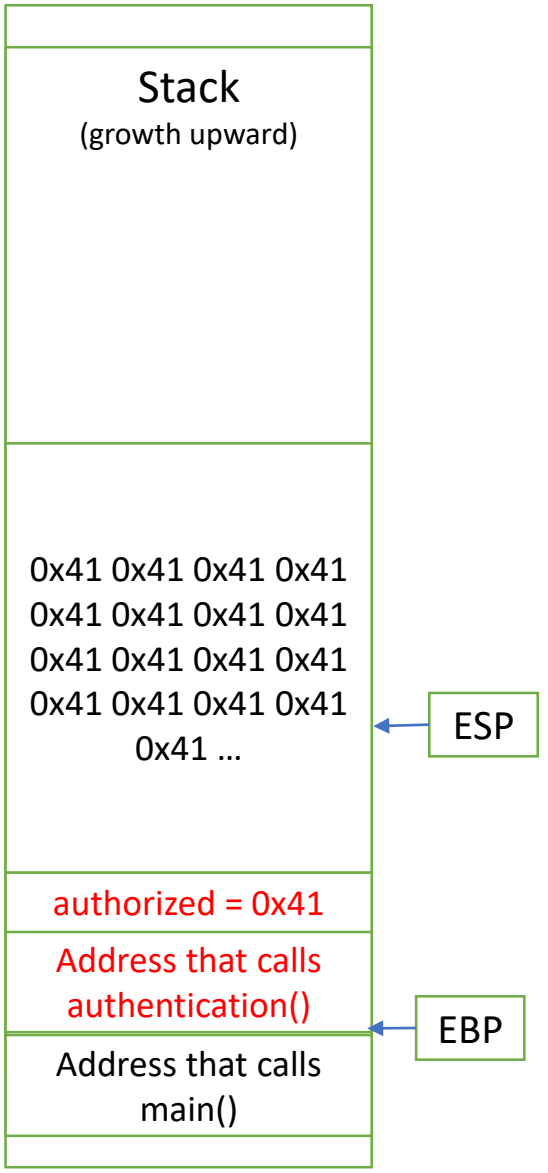
Buffer overflow (exploit)

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

int authentication(){
   int authorized = 0;
   char cPassCode[100];
   scanf("%s", cPassCode);
   if (!strcmp(cPassCode, "mcgill-cby02"))
       authorized = 1;
    return authorized;
}

int main()
{
    if (authentication())
        printf("Successfully logged in!");
    else
        printf("Wrong password.");
    return 0;
}
```

0x41 repeated 101 times as input.



Buffer overflow (exploit)

char cPassCode[100];

scanf("%s", cPassCode);

read user input into the buffer *cPassCode*.

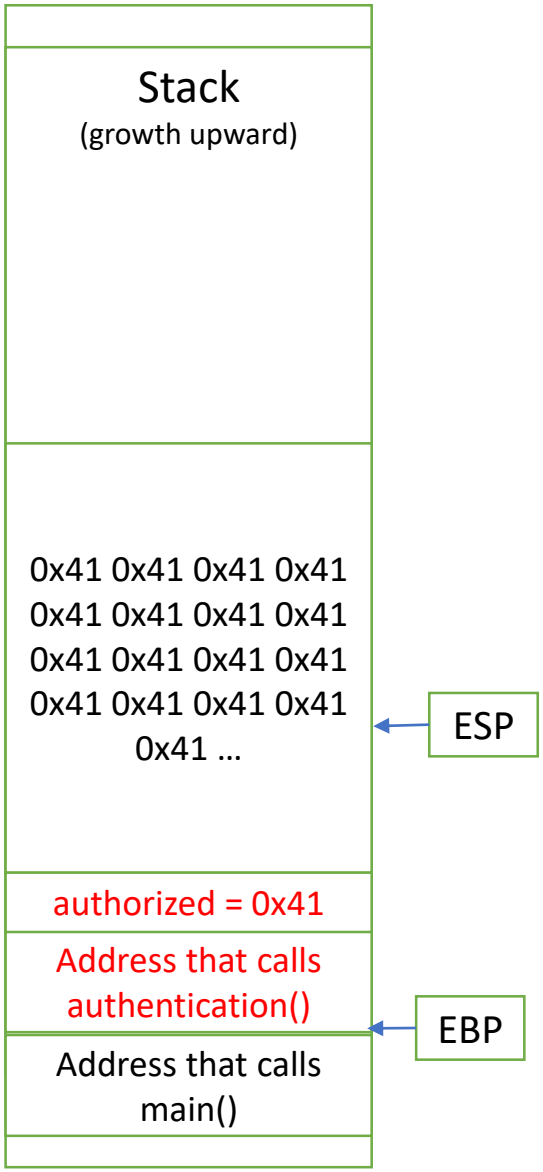
- If user input is *longer* than the allocated buffer, it is over written to the memory.

Input: AAAAAAAAAAAAAAAA … (101 As)

(101 bytes, the buffer is only 100 bytes)

The next byte to the *cPassCode* is *authorized*.

authorized will be written as the encoding of character 'A', which is 0x41.



Fuzzy Test!

1. Randomly running a lot of unexpected inputs (unusually long)
2. Some inputs may cause the program to crash
3. Indicates a potential overflow issue on the user input!
4. Fine tuning the input to find the sweet pot to overwrite the value you want (too short then it is normal output, too long will crash the program)

Buffer overflow (exploit)

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

int authentication(){
   int authorized = 0;
   char cPassCode[100];
   scanf("%s", cPassCode);
   if (!strcmp(cPassCode, "mcgill-cby02"))
       authorized = 1;
    else
        authorized = 0;
    return authorized;
}

int main()
{
    if (authentication())
        printf("Successfully logged in!");
    else
         printf("Wrong password.");
    return 0;
}
```

Swap these two line can prevent modifying the authorized flag.

Is it safe?

Buffer overflow (exploit)

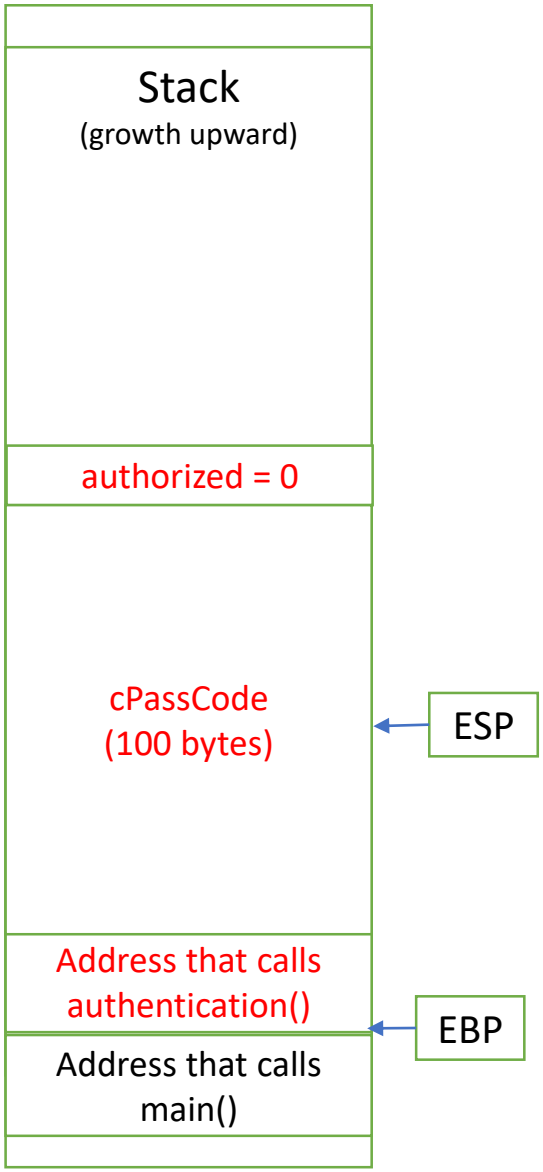
```
#include <stdio.h>
#include <string.h>

int authentication(){
   char cPassCode[100];
   int authorized = 0;
   scanf("%s", cPassCode);
   if (!strcmp(cPassCode, "mcgill-cby02"))
      authorized = 1;
    else
       authorized = 0;
    return authorized;
}

int main()
{
    if (authentication())
       printf("Successfully logged in!");
    else
        printf("Wrong password.");
    return 0;
}
```

Swap these two line can prevent modifying the authorized flag.

Is it safe?



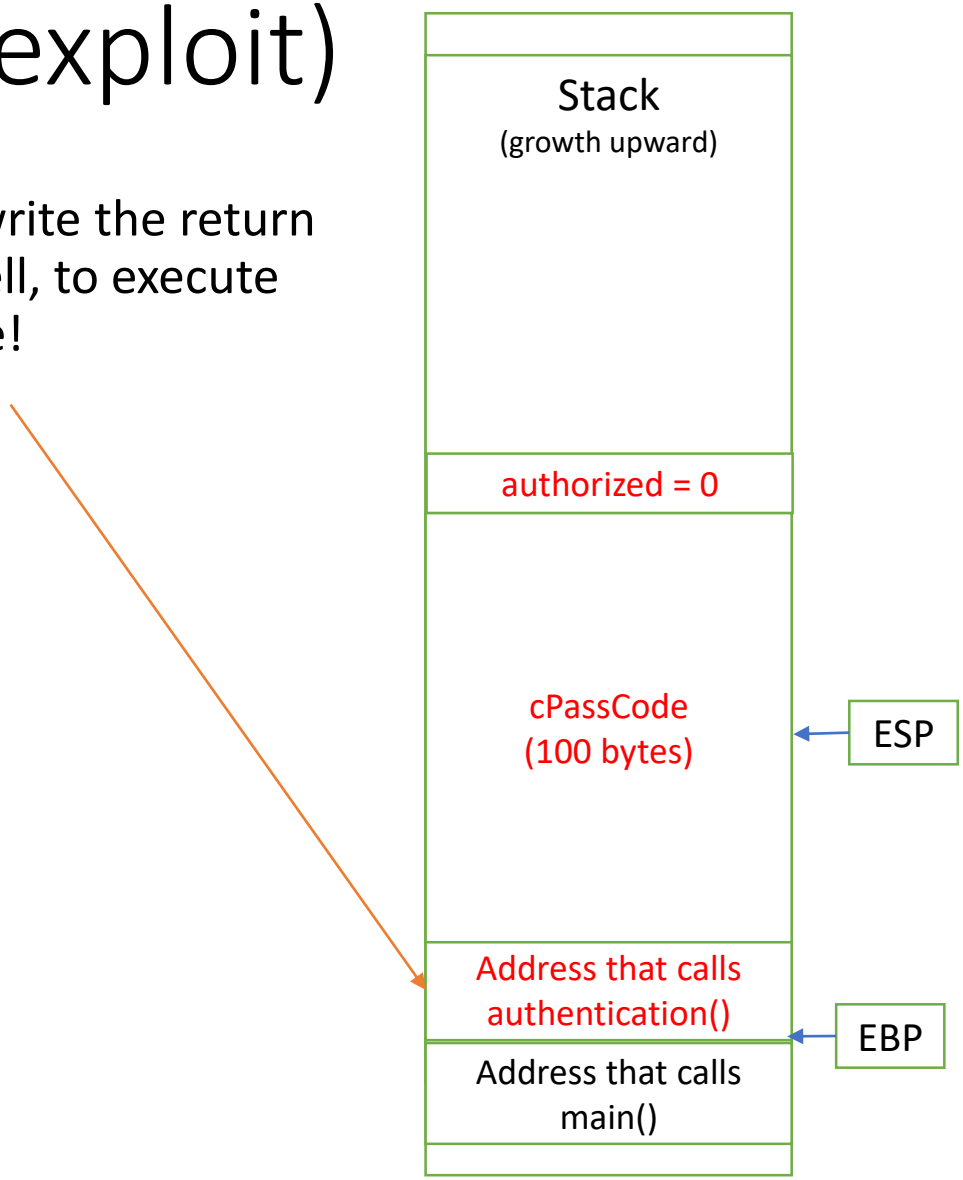
Buffer overflow (exploit)

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

int authentication(){
   char cPassCode[100];
   int authorized = 0;
   scanf("%s", cPassCode);
   if (!strcmp(cPassCode, "mcgill-cby02"))
      authorized = 1;
    else
       authorized = 0;
    return authorized;
}

int main()
{
    if (authentication())
       printf("Successfully logged in!");
    else
        printf("Wrong password.");
    return 0;
}
```

We can overwrite the return address as well, to execute arbitrary code!



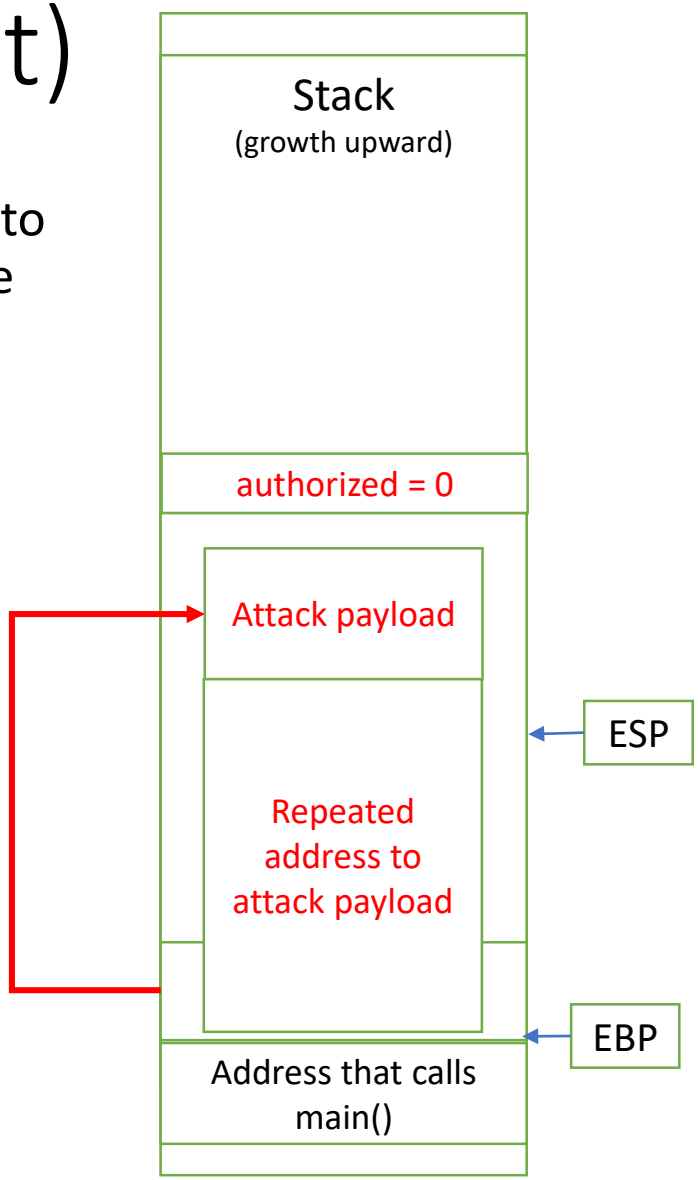
Buffer overflow (exploit)

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

int authentication(){
   char cPassCode[100];
   int authorized = 0;
   scanf("%s", cPassCode);
   if (!strcmp(cPassCode, "mcgill-cby02"))
      authorized = 1;
    else
       authorized = 0;
    return authorized;
}

int main()
{
    if (authentication())
       printf("Successfully logged in!");
    else
        printf("Wrong password.");
    return 0;
}
```

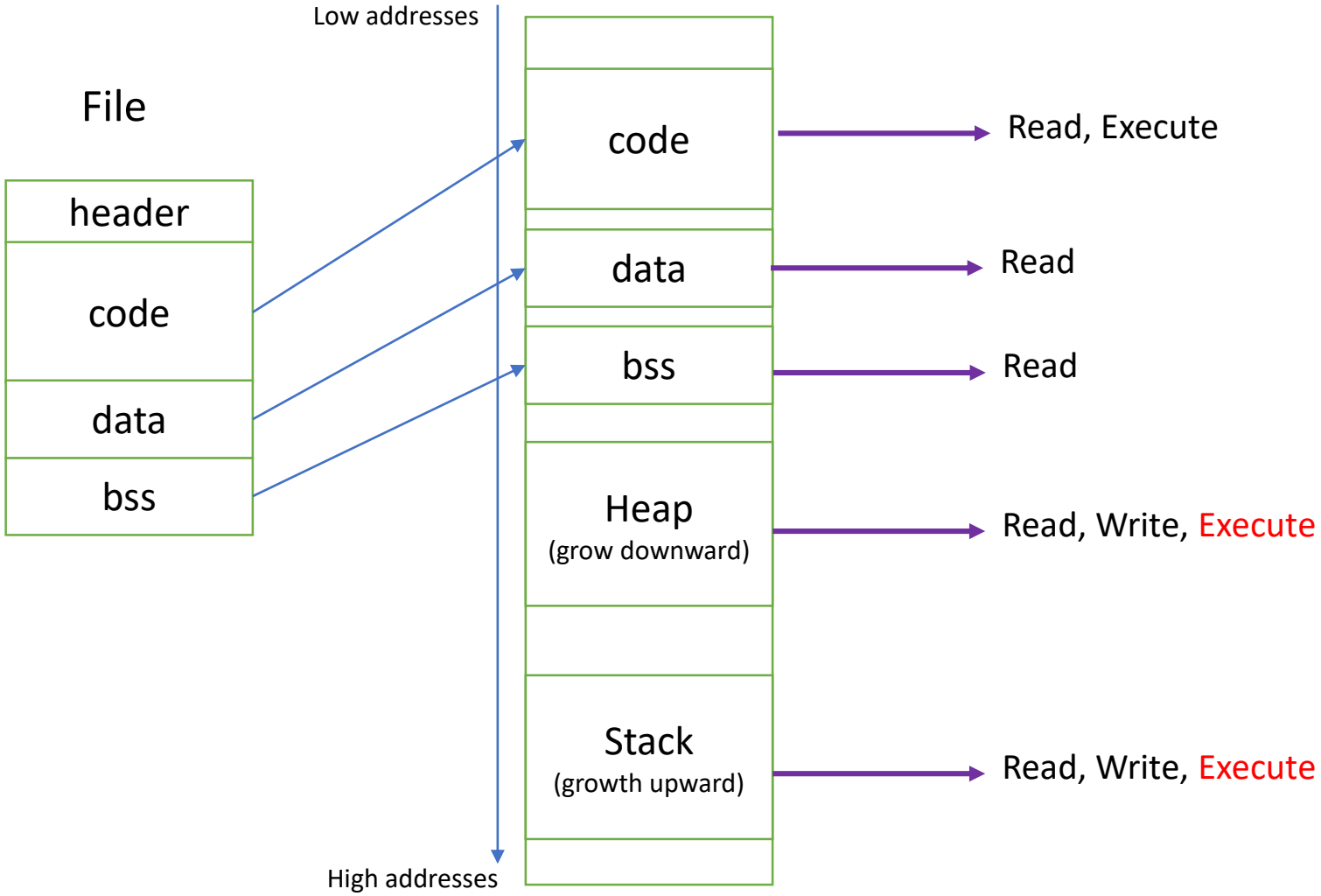
We can overwrite the run to address as well, to execute arbitrary code!



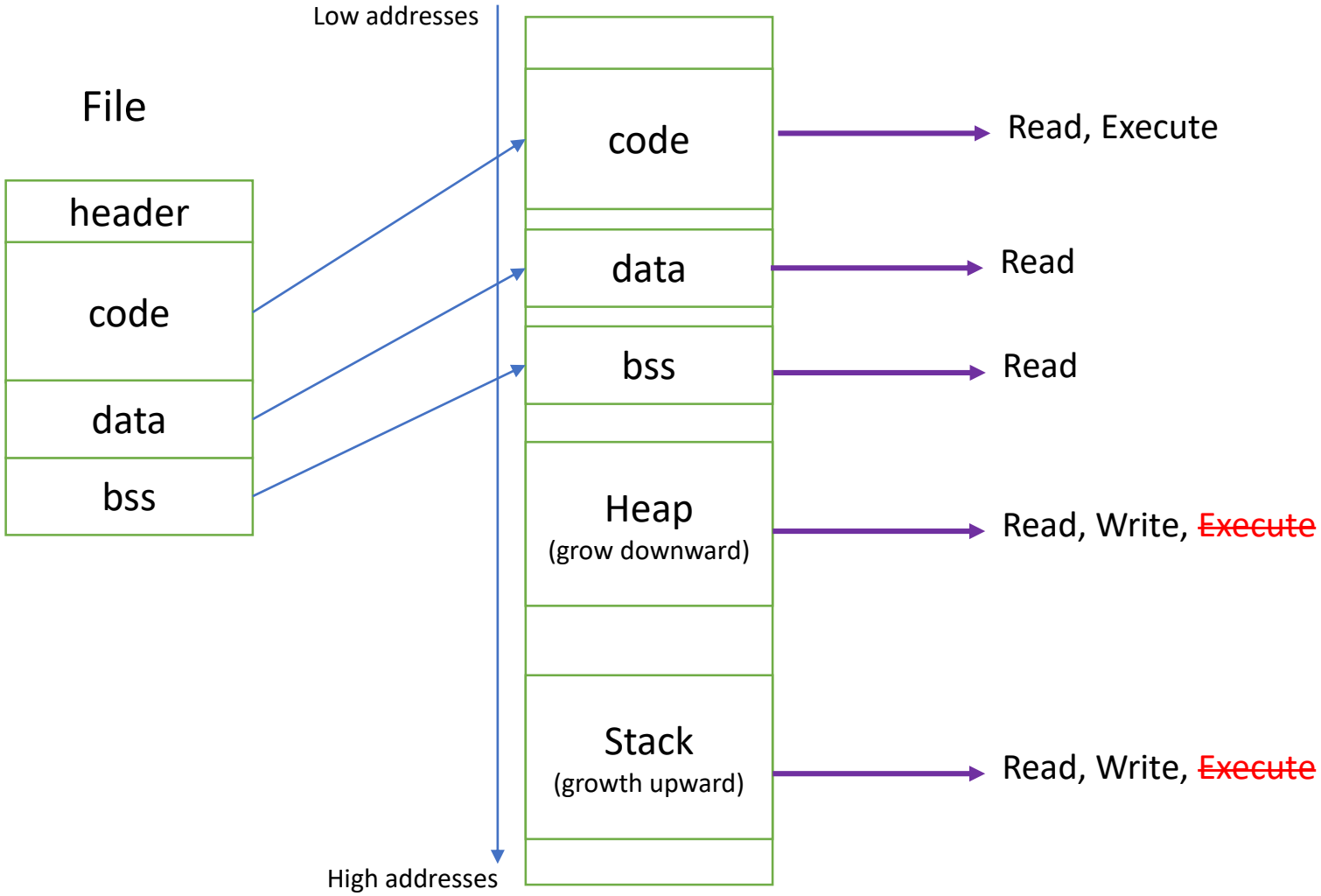
Protection

- Data Execution Prevention (DEP)
- Address Space Layout Randomization (ASLR)
- Stack Canaries

Without DEP


Low addresses
File
header
code
data
bss
code
data
bss
Heap
(grow downward)
Stack
(growth upward)
High addresses
Read, Execute
Read
Read
Read, Write, Execute
Read, Write, Execute

With DEP


Low addresses
File
header
code
data
bss
code
data
bss
Heap
(grow downward)
Stack
(growth upward)
High addresses
Read, Execute
Read
Read
Read, Write, Execute
Read, Write, Execute

DEP

- No memory segment of memory should be ever both writable and executable.
- Enforced by the operation system.
  - Started from Windows XP SP2 (2004)
  - Started from Linux Kernel 2.6.8 (2004)
  - Started from Mac OSX 10.5 (2006)
  - One of the main defence that we need to by pass (from the hackers' perspective)

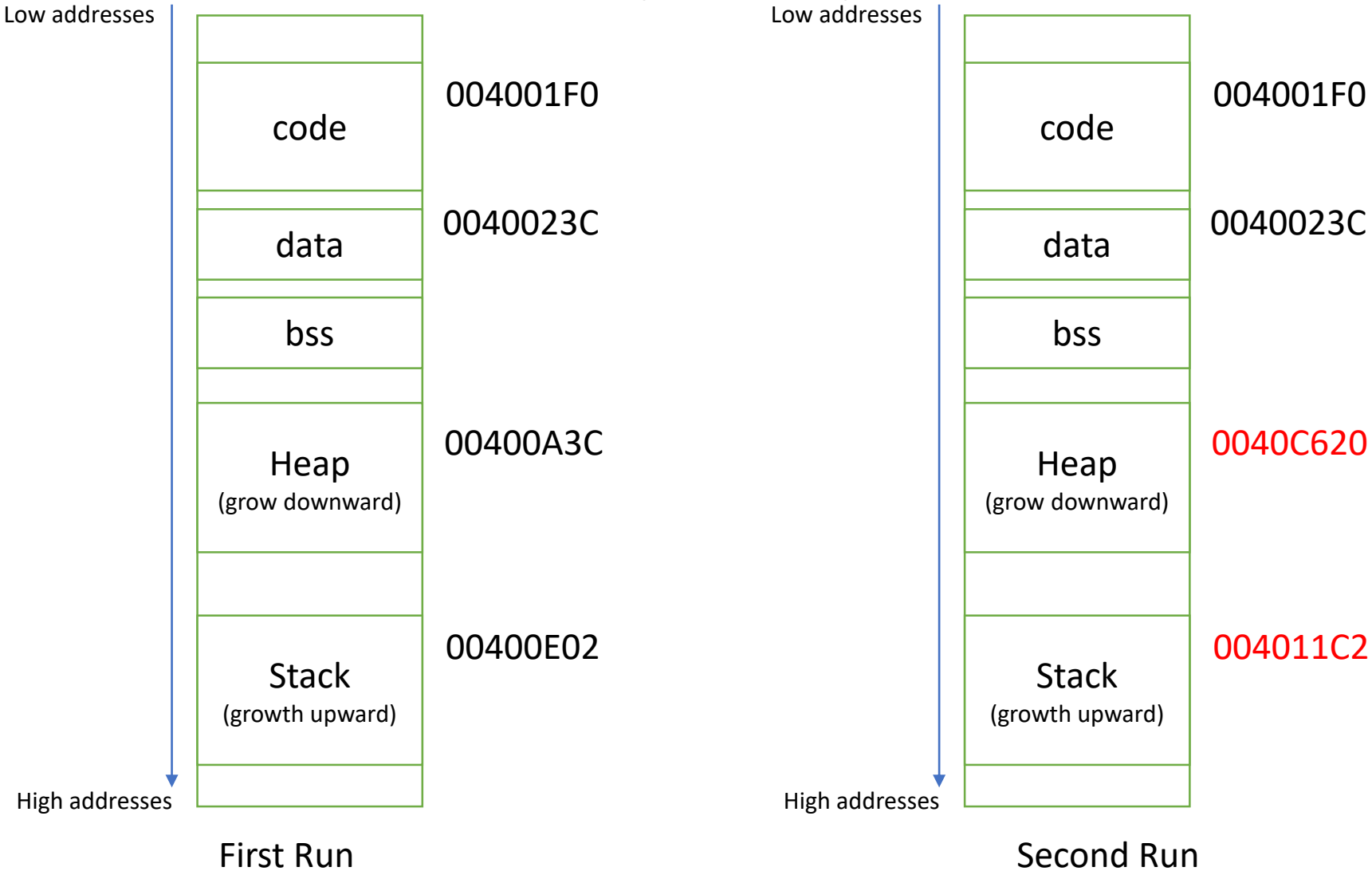
Address Space Layout Randomization (ASLR)

- Important memory segments are random for every execution.
- Randomized stack layout
  - Cannot find the use the address from last run
  - Make multi-run exploit difficult.
    - First injection, then find return address, and finally overwrite return address. (3 runs!)
- Make things harder

Address Space Layout Randomization (ASLR)

- Stack address changed
- Heap address changed
- Library address changed
- Not all segments are randomized!

Address Space Layout Randomization (ASLR)



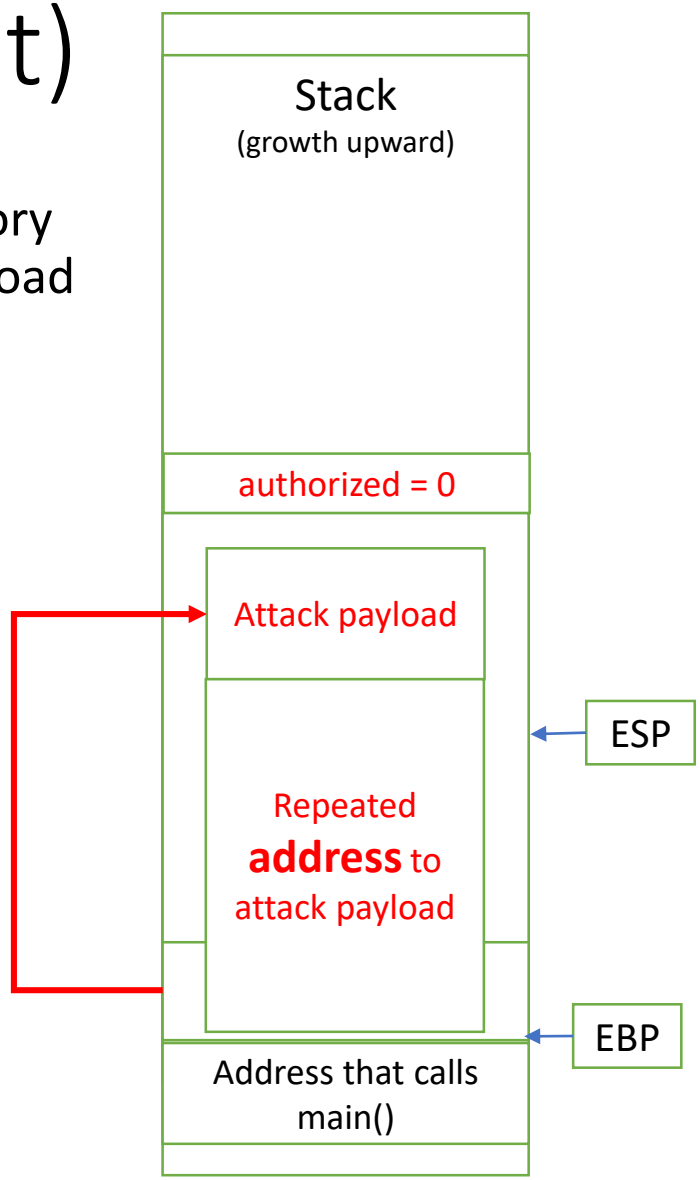
Buffer overflow (exploit)

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

int authentication(){
   char cPassCode[100];
   int authorized = 0;
   scanf("%s", cPassCode);
   if (!strcmp(cPassCode, "mcgill-cby02"))
      authorized = 1;
    else
       authorized = 0;
    return authorized;
}

int main()
{
    if (authentication())
       printf("Successfully logged in!");
    else
        printf("Wrong password.");
    return 0;
}
```

Can't figure out the memory address to the attack payload



Address Space Layout Randomization (ASLR)

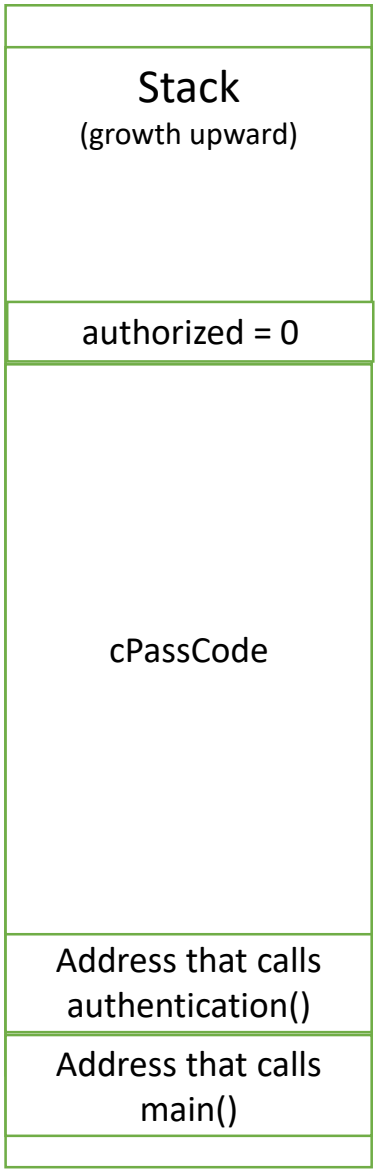
- Xbox 360 - 2005: No ASLR
- PlayStation 3 – 2006: No ASLR
- Nintendo 3DS – 2011: No ASLR
  - Sony drops lawsuit against Geohot (ethical hacker)
  - DON’T hack Sony products!

Canary check (before)

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

int authentication(){
   char cPassCode[100];
   int authorized = 0;
   scanf("%s", cPassCode);
   if (!strcmp(cPassCode, "mcgill-cby02"))
       authorized = 1;
    else
        authorized = 0;
    return authorized;
}

int main()
{
    if (authentication())
        printf("Successfully logged in!");
    else
         printf("Wrong password.");
    return 0;
}
```

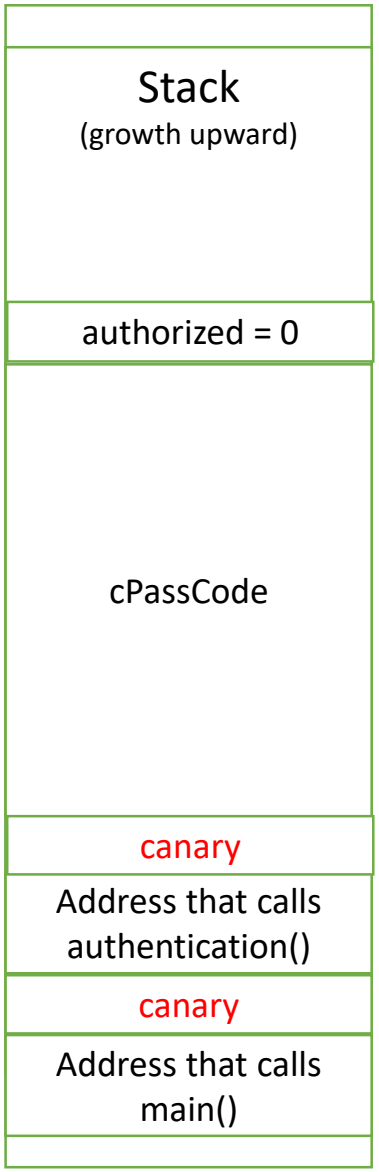


Canary check (after)

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

int authentication(){
   char cPassCode[100];
   int authorized = 0;
   scanf("%s", cPassCode);
   if (!strcmp(cPassCode, "mcgill-cby02"))
       authorized = 1;
    else
        authorized = 0;
    return authorized;
}

int main()
{
    if (authentication())
        printf("Successfully logged in!");
    else
         printf("Wrong password.");
    return 0;
}
```



Canary check (a.k.a. Stack Canaries)

- Protect stack from overflow
- Check integrity before function return
- Canary -> a special integer value
- Push onto the stack before certain trigger (e.g. return address)
- Pop from the stack and check the value before reading the trigger
- Also known as stack cookie.

Take-aways

- CVE CVSS CWE
- How stack and stack frame work
- How/why does buffer overflow happen?
- Prevention
  - DEP
  - ASLR
  - Canary