CS165 – Computer Security

Software Vulnerabilities

October 3, 2024

Outline

- Vulnerabilities!
- Elements of a vulnerability
- Impact of vulnerability exploitation
  - Confidentiality
  - Integrity
  - Availability
- Example vulnerability (and why)
- Threat model

Vulnerability

- A vulnerability is a flaw (e.g., in software) that is accessible to an adversary who can exploit that flaw
- Flaw – Functionality that may violate security
  - E.g., Crash, Use or modify sensitive data
- Accessible – Adversaries may access the flaw
  - Flaw can run using adversary input
- Exploit – Provide inputs to cause security violation
  - Adversary can produce an attack payload

Security Requirements

- Security requirements are described in three categories (CIA)
- Confidentiality (Secrecy)
  - E.g., Prevent leakage of sensitive data to an adversary
- Integrity
  - E.g., Prevent unauthorized modification of sensitive data
- Availability
  - E.g., Prevent blockage of use of critical services

Example Code

- Does this code have a vulnerability?

```c
#include <stdio.h>

int function( char *source )
{
  char buffer[10];

  sscanf( source, "%s", buffer );
  printf( "buffer address: %p\n\n", buffer );
  return 0;
}

int main( int argc, char *argv[] )
{
  function( argv[1] );
}
```

Security Requirements of the Code

- Confidentiality
  - Must not read pointer values
- Integrity
  - Must not modify data other than “buffer”
- Availability
  - Must complete its execution
- Not an exhaustive list

What's a Flaw?

- A vulnerability is a flaw (e.g., in software) that is accessible to an adversary who can exploit that flaw
- Flaw – A functionality that violates security
  - What violates a security requirement (CIA)?

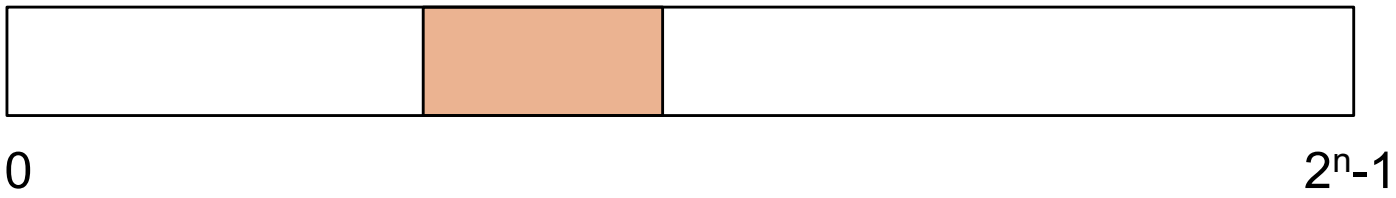
Example Code

- Does this code have a flaw?

```
#include <stdio.h>

int function( char *source )
{
  char buffer[10];

  sscanf( source, "%s", buffer );
  printf( "buffer address: %p\n\n", buffer );
  return 0;
}

int main( int argc, char *argv[] )
{
  function( argv[1] );
}
```

How is ”buffer” represented?

- Variable buffer occupies 10 bytes in the stack region
  - `char buffer[10];`



- buffer is an array of objects of type char
  - C does not represent strings as a data type
- buffer is also a pointer to the memory region of 10 bytes
  - C uses the variable “buffer” to store the memory location of these 10 bytes in the process’s address space
  - `printf(“0x%x\n”, buffer);  // prints addr`
  - `printf(“%s\n”, buffer);  // prints value`

What's a Flaw?

- A vulnerability is a flaw (e.g., in software) that is accessible to an adversary who can exploit that flaw
- Flaw – A functionality that violates security
  - What violates a security requirement (CIA)?
- In the example code, memory outside of "buffer" may be written illicitly, violating integrity

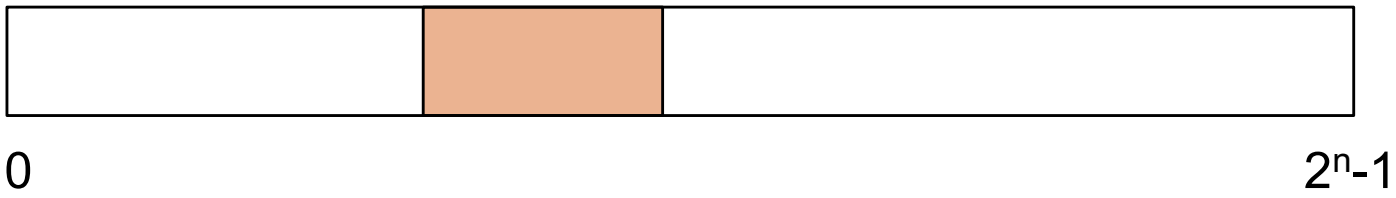
Example Code

- The function `sscanf` writes each byte from ”source” to “buffer” until a 0-byte in “source”

```
#include <stdio.h>

int function( char *source )
{
  char buffer[10];

  sscanf( source, "%s", buffer );
  printf( "buffer address: %p\n\n", buffer );
  return 0;
}

int main( int argc, char *argv[] )
{
  function( argv[1] );
}
```

Why is that a flaw?

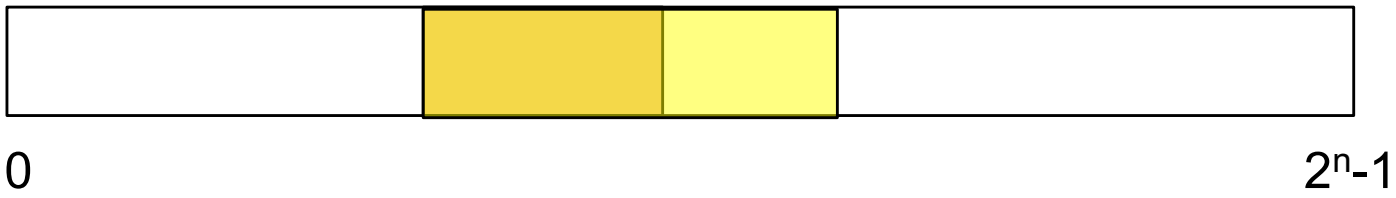
- buffer occupies 10 bytes in the stack region
  - `sscanf(source, "%s, buffer)`



- `sscanf` starts at the memory location "buffer"
  - And writes until a null byte is found in "source"
  - Does source have to have a null byte within its first 10 bytes?

Why is that a flaw?

- Buffer occupies 10 bytes in the stack region
  - `sscanf(source, "%s, buffer)`



- `sscanf` starts at the memory location "buffer"
  - And writes until a null byte is found in "source"
  - Which could be more than 10 bytes
- Which illicitly writes memory outside of the allocated region for "buffer"
  - What is there? We'll see

What's Accessibility?

- A vulnerability is a flaw (e.g., in software) that is accessible to an adversary who can exploit that flaw
- Accessibility – Can an adversary access the flaw?
  - What does "access" mean?

Back to Accessibility

- A vulnerability is a flaw (e.g., in software) that is accessible to an adversary who can exploit that flaw
- Accessibility – Can an adversary access the flaw?
  - I.e., Cause the flawed action to happen
- Can the adversary cause the flawed code to run? Can the adversary supply the inputs to cause the flawed action to happen?

Example Code

- Is “source” accessible to an adversary?
- Can an adversary cause “sscanf” to run?

```
#include <stdio.h>

int function( char *source )
{
  char buffer[10];

  sscanf( source, "%s", buffer );
  printf( "buffer address: %p\n\n", buffer );
  return 0;
}

int main( int argc, char *argv[] )
{
  function( argv[1] );
}
```

Example Code

- An adversary can supply the value “source” from the command line

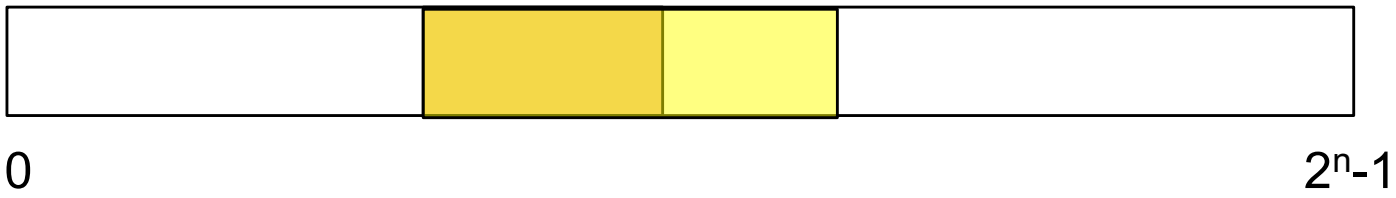
```
#include <stdio.h>

int function( char *source )
{
  char buffer[10];

  sscanf( source, "%s", buffer );
  printf( "buffer address: %p\n\n", buffer );
  return 0;
}

int main( int argc, char *argv[] )
{
  function( argv[1] );
}
```

What's Exploitation?

- A vulnerability is a flaw (e.g., in software) that is accessible to an adversary who can exploit that flaw
- Exploit – Can the adversary use the accessible flaw to cause the program's execution to violate a security requirement
  - What are violations of security requirements?

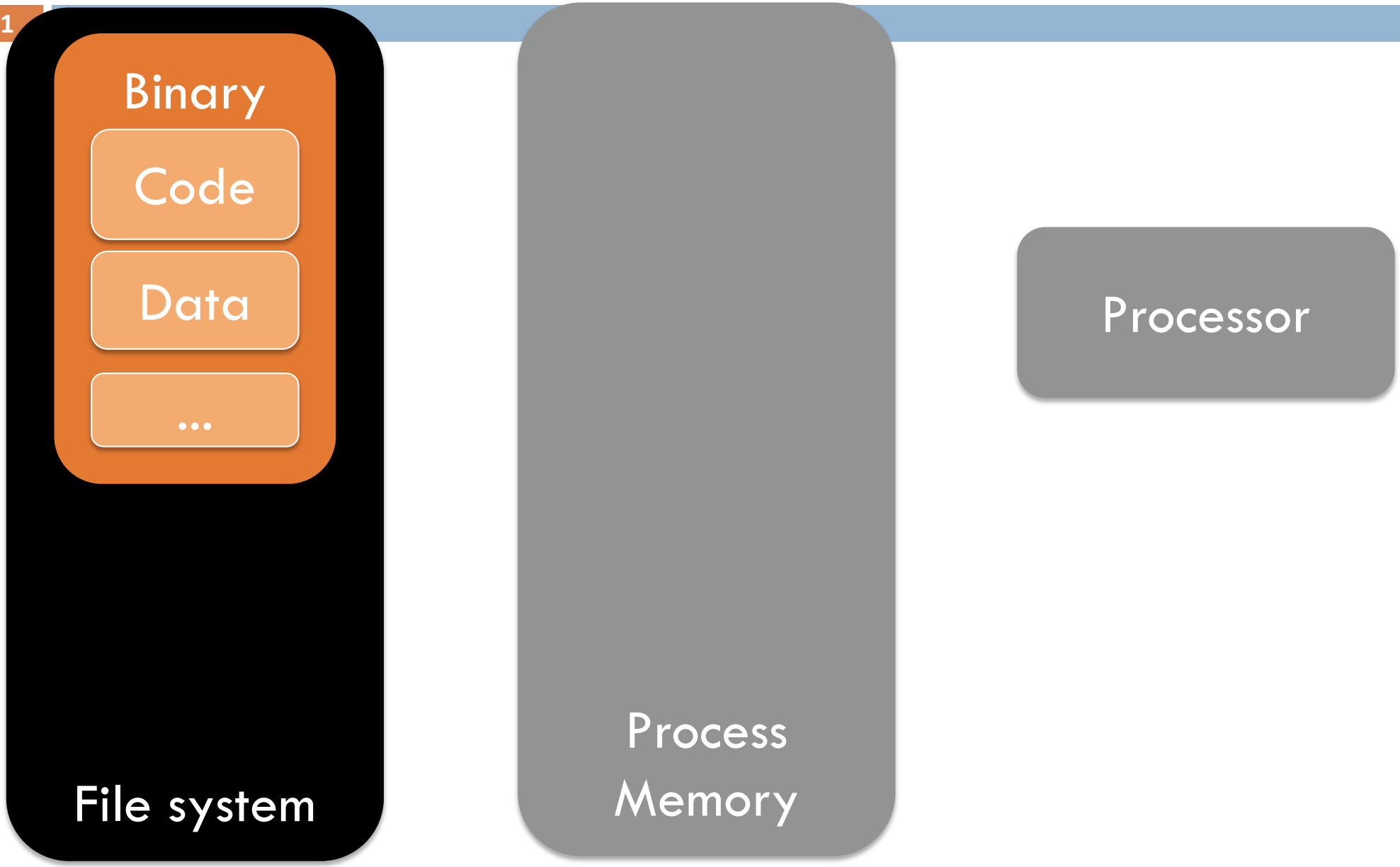
What Can Be Exploited?

- What is in the yellow memory area
  - Stack memory

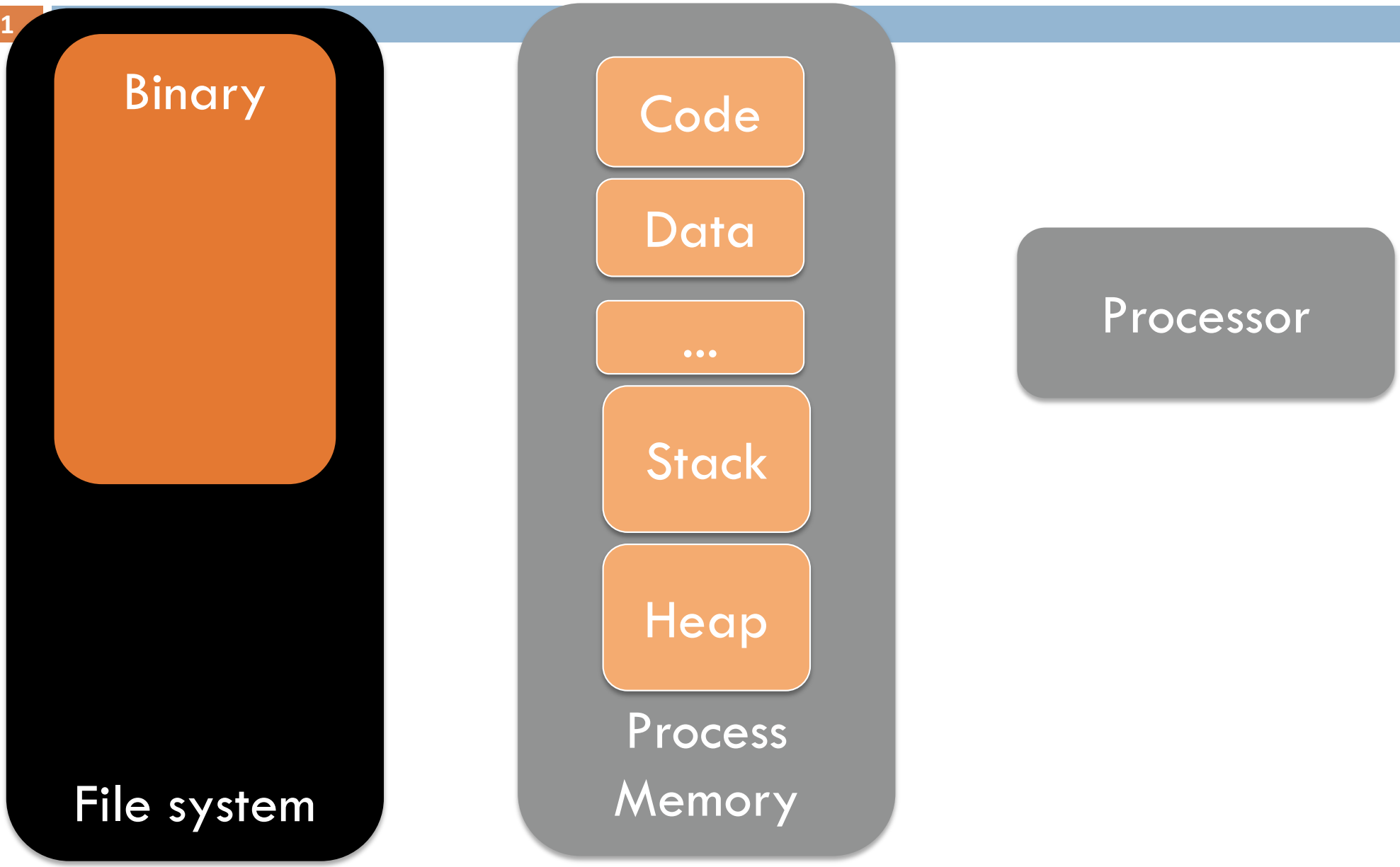


- What does stack memory look like?

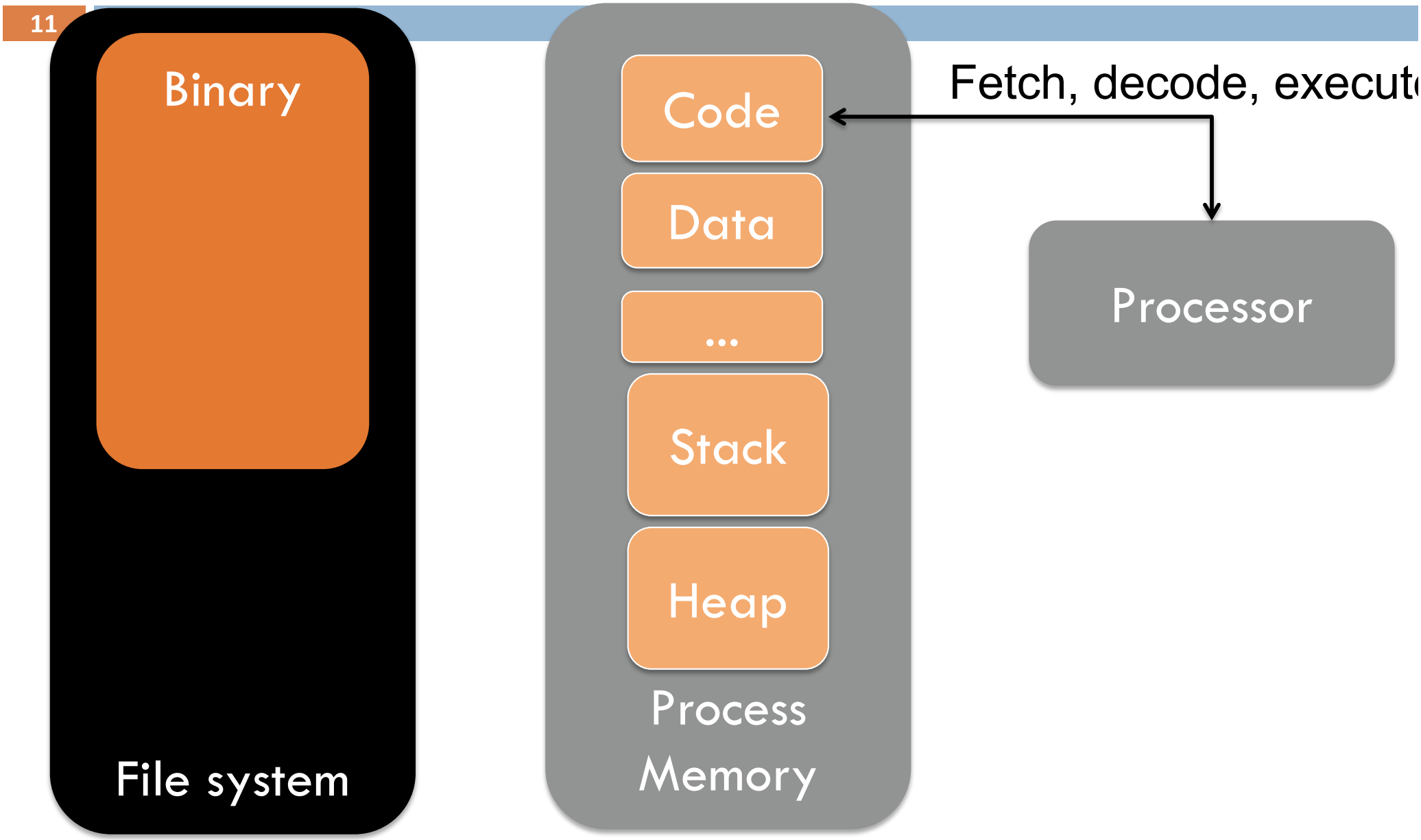
Basic Execution



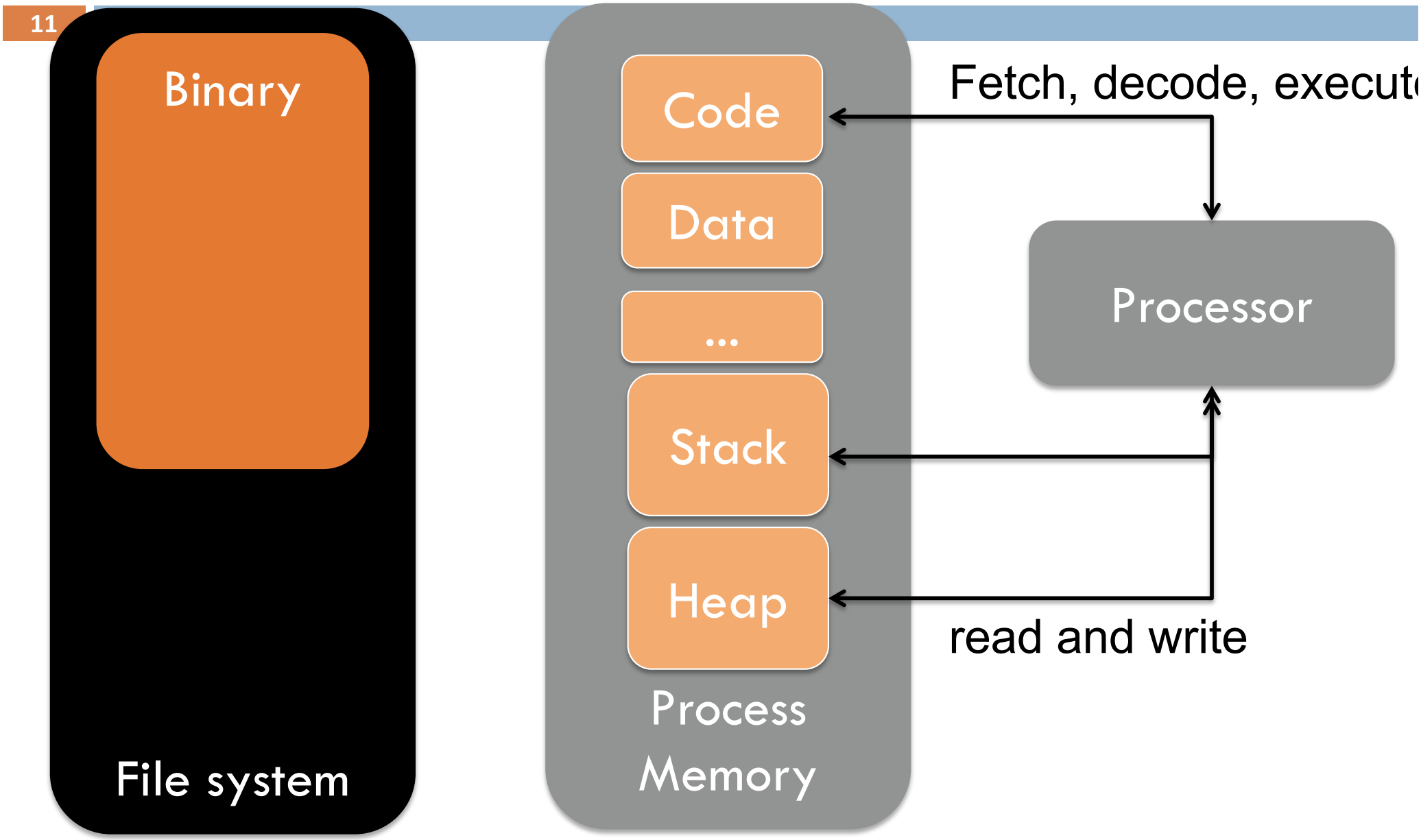
Basic Execution



Basic Execution

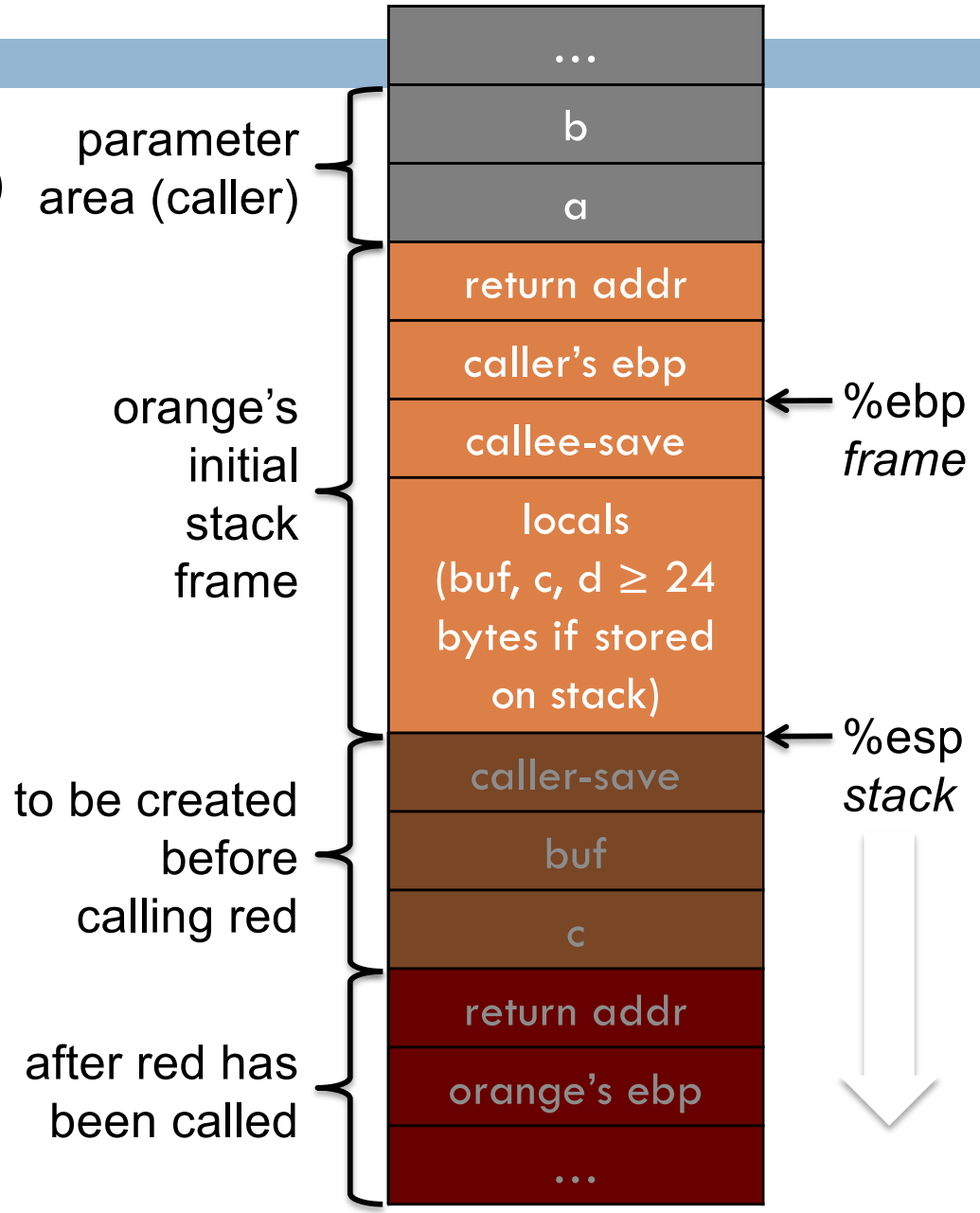


Basic Execution



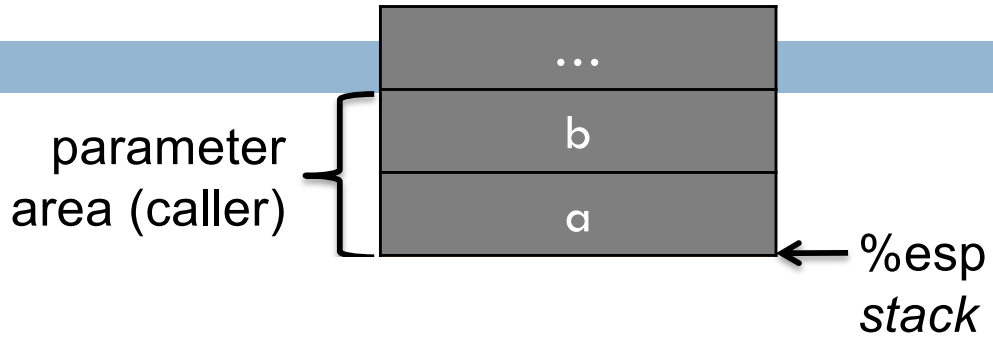
cdecl – the default for Linux & gcc


```
int orange(int a, int b)
{
   char buf[16];
   int c, d;
   if(a > b)
      c = a;
   else
      c = b;
   d = red(c, buf);
   return d;
}
```



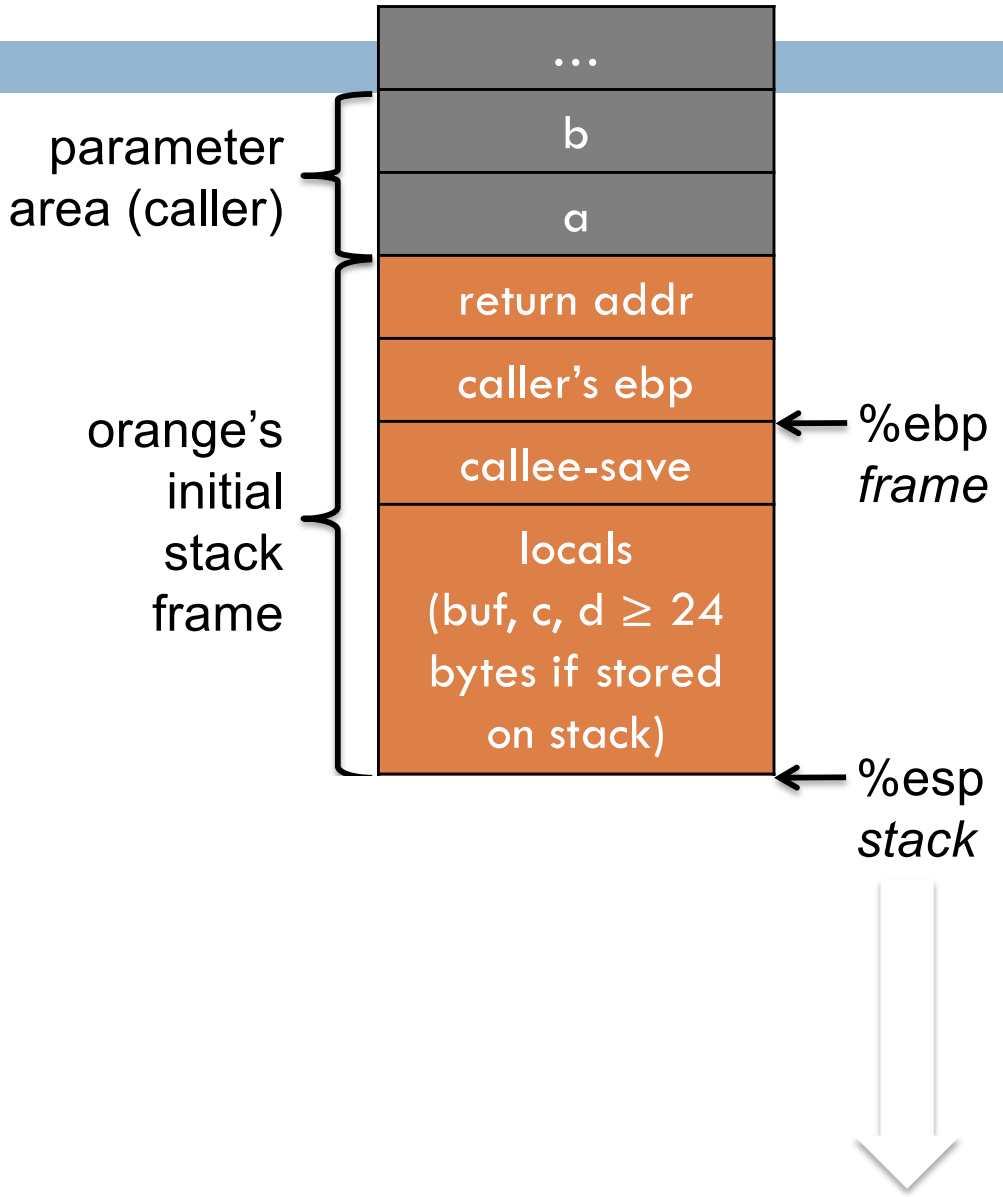
cdecl – the default for Linux & gcc


```
int orange(int a, int b)
{
   char buf[16];
   int c, d;
   if(a > b)
      c = a;
   else
      c = b;
   d = red(c, buf);
   return d;
}
```



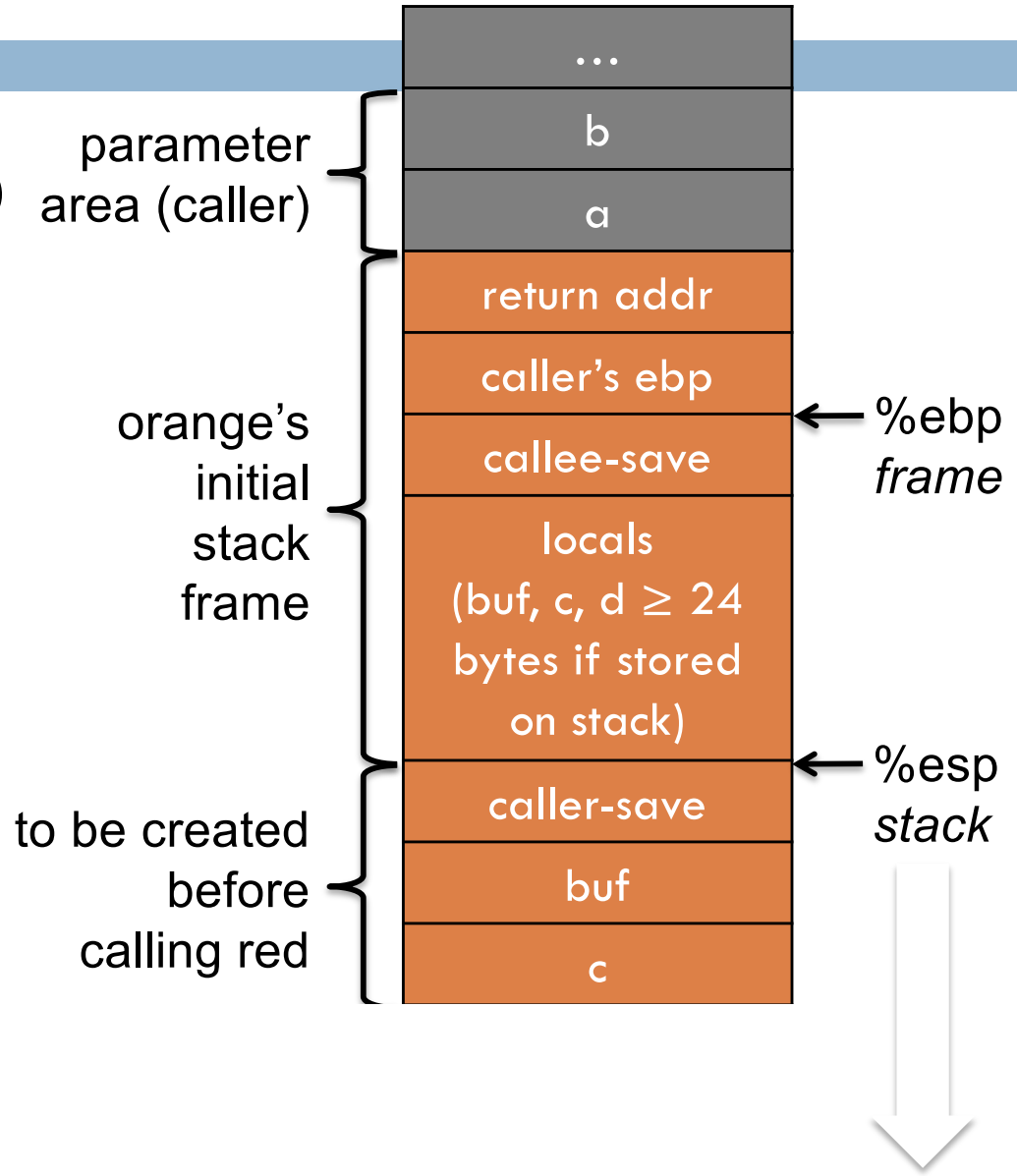
cdecl – the default for Linux & gcc


```
int orange(int a, int b)
{
   char buf[16];
   int c, d;
   if(a > b)
      c = a;
   else
      c = b;
   d = red(c, buf);
   return d;
}
```



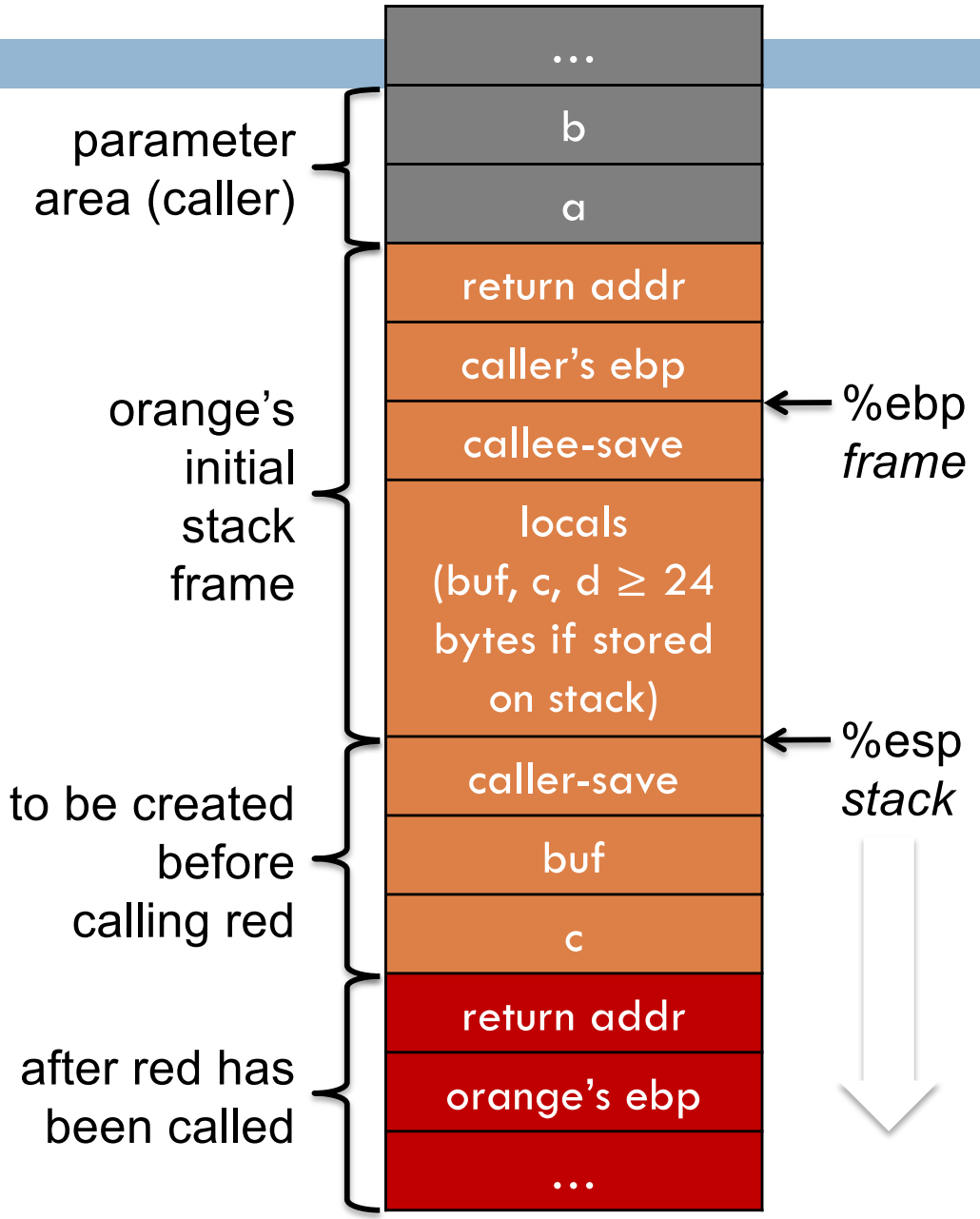
cdecl – the default for Linux & gcc


```
int orange(int a, int b)
{
   char buf[16];
   int c, d;
   if(a > b)
      c = a;
   else
      c = b;
   d = red(c, buf);
   return d;
}
```



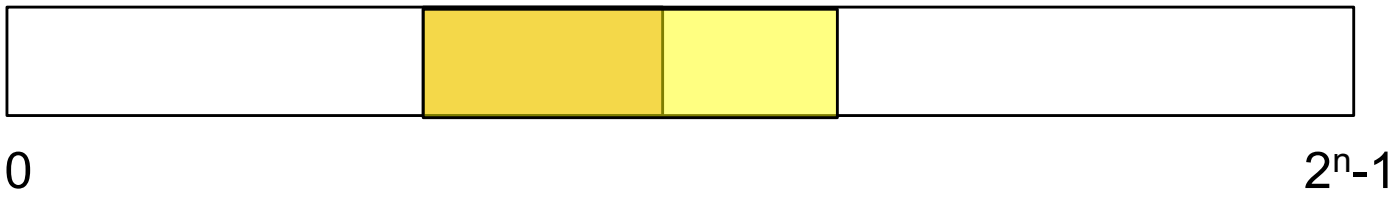
cdecl – the default for Linux & gcc


```
int orange(int a, int b)
{
   char buf[16];
   int c, d;
   if(a > b)
      c = a;
   else
      c = b;
   d = red(c, buf);
   return d;
}
```



What Can Be Exploited?

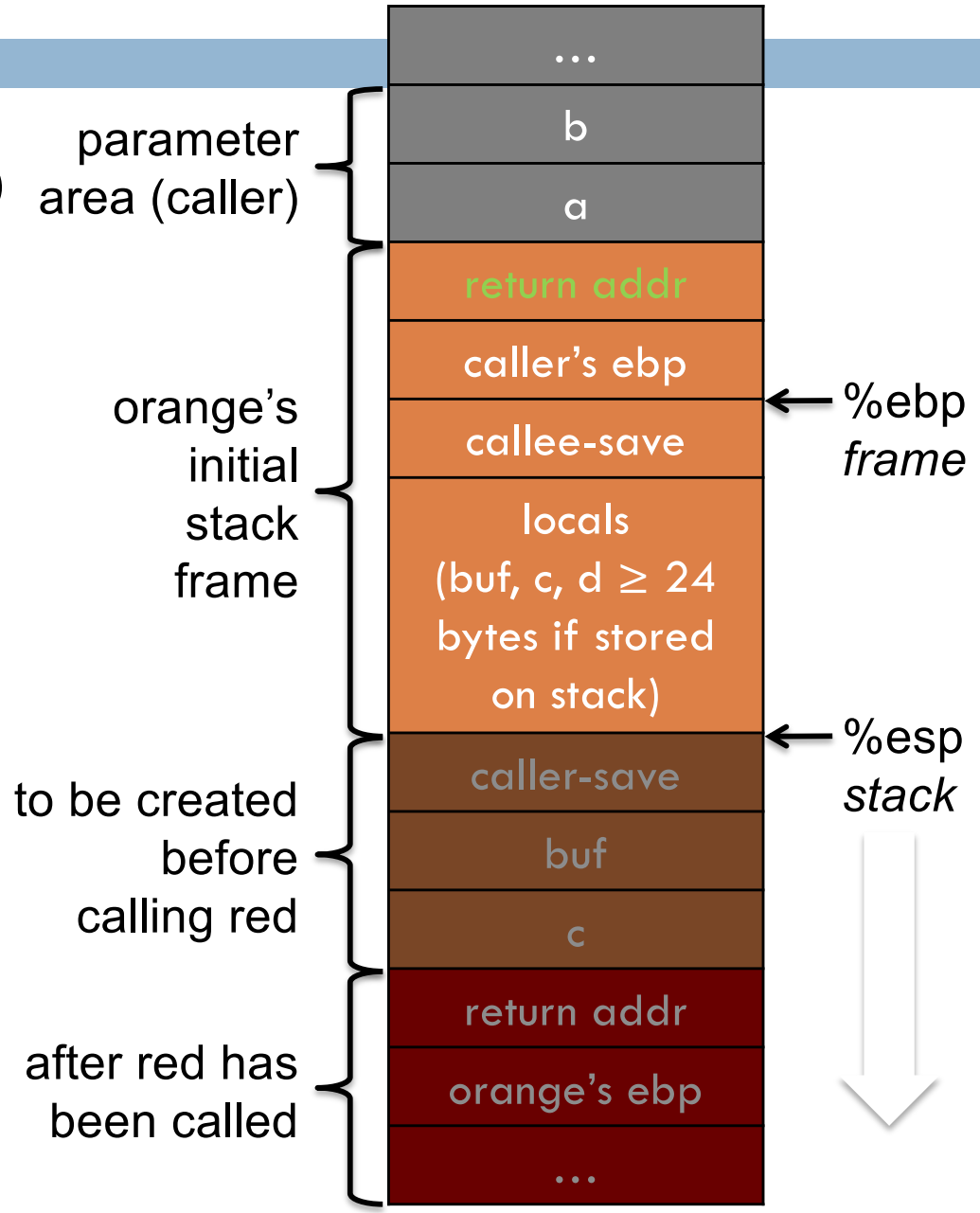
- What is in the yellow memory area
  - Stack memory



- What can be written illicitly by sscanf in this program?
  - Local variables: none, as “buffer[10]” is the only one
  - Frame pointer (ebp)
  - Return address
  - Prior stack frames (gray)

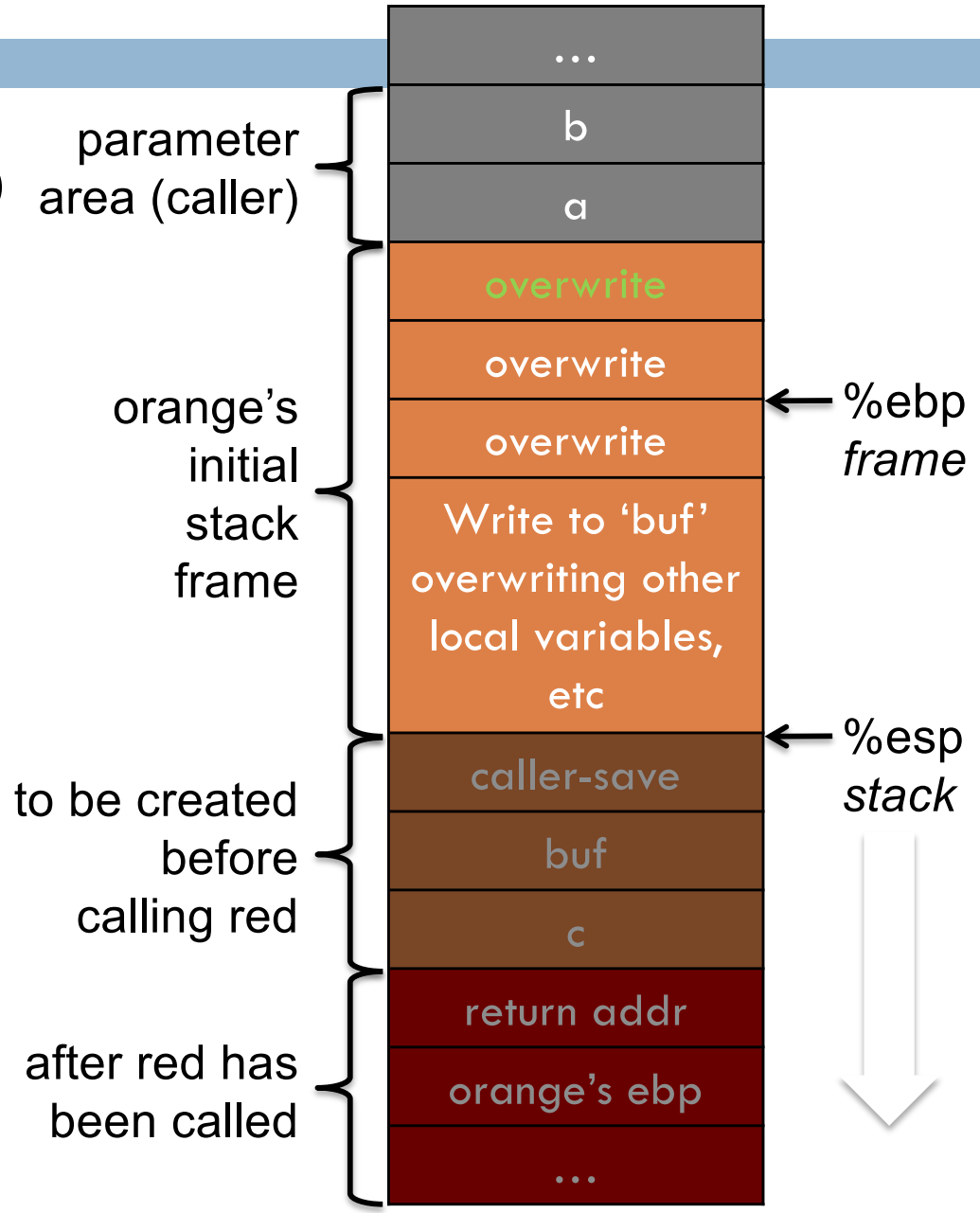
Hijack Control Flow – Ret Addr


```
int orange(int a, int b)
{
   char buf[16];
   int c, d;
   if(a > b)
      c = a;
   else
      c = b;
   d = red(c, buf);
   return d;
}
```



Hijack Control Flow – Ret Addr


```
int orange(int a, int b)
{
   char buf[16];
   int c, d;
   if(a > b)
      c = a;
   else
      c = b;
   d = red(c, buf);
   return d;
}
```



Return Address

- The return address of a function determines the code that is run when the function returns
  - By modifying this value, you can change how the program executes
  - In arbitrary and powerful ways
- The main way of hijacking programs for many years, but now there are new hijacking techniques that are harder for defenders to detect

Threat Model

- Vulnerabilities connect exploitable flaws in programs with adversary accessibility
- What resources an adversary can access from a program's attack surface and the operations an adversary can perform on those resources form the Threat Model
  - You should consider systematically what threats your programs face

Threat Model

- What are the resources and operations on those resources that form the threats here?

```c
#include <stdio.h>

int function( char *source )
{
  char buffer[10];

  sscanf( source, "%s", buffer );
  printf( "buffer address: %p\n\n", buffer );
  return 0;
}

int main( int argc, char *argv[] )
{
  function( argv[1] );
}
```

Threat Model

- Can supply (write) the value “source” from the command line – argv[1] – supply (write) the value of argc

```
#include <stdio.h>

int function( char *source )
{
  char buffer[10];

  sscanf( source, "%s", buffer );
  printf( "buffer address: %p\n\n", buffer );
  return 0;
}

int main( int argc, char *argv[] )
{
  function( argv[1] );
}
```

Kinds of Flaws

- The kind of flaw shown in this example is called a memory error
  - Most common type of flaw in current vulnerabilities
  - Approximately 70% of vulnerabilities reported by Google and Microsoft independently
- But, there are many other types of flaws
  - Even in connecting the program with its environment

Conclusions

- Vulnerabilities that compromise confidentiality or integrity are common
- Vulnerabilities allow adversary to access flaws that they can exploit to violate security requirements
- We demonstrated a memory error vulnerability to hijack the return address on the stack
  - Buffer overflow (more later)
- Many types of flaws are out there that may be exploitable

Questions