CS165 – Computer Security

Software Vulnerabilities

September 29, 2025

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 or bug
- 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
- Violating these requirements is a security violation
- But, security requirements are application-specific

Example Code

- Does this code have a vulnerability?

```
#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
  - Only outputs are printed “buffer” value and location
- 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 may violate security
  - What violates a security requirement (CIA)?

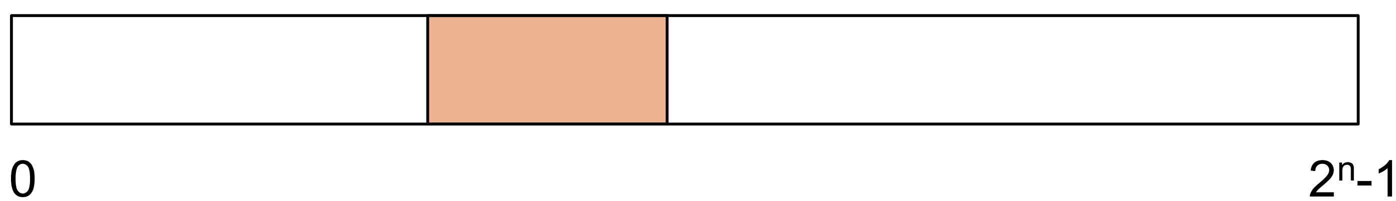
Example Code

- 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[10] – 1 byte each
  - 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
  - How does that happen?

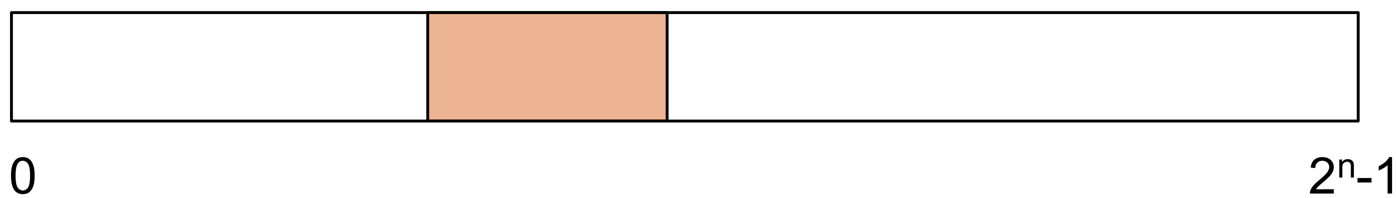
Example Code

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

```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] );
}
```

Why is that a flaw?

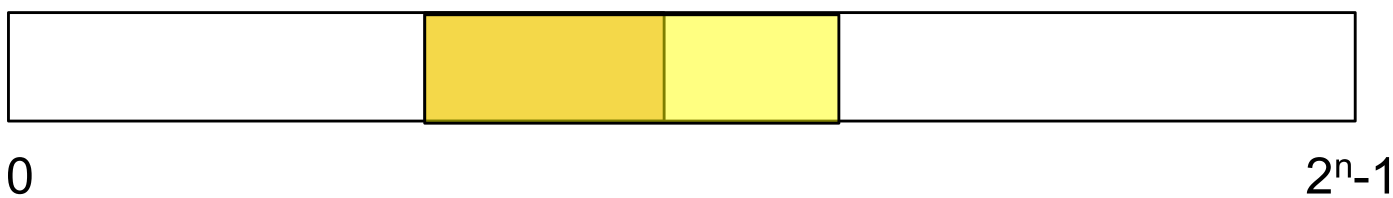
- buffer occupies 10 bytes in the stack region
  - `sscanf(source, “%s, buffer)`



- `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 before the null bytes
- Which illicitly writes memory outside of the allocated region for "buffer"
  - What is there? We'll see

Memory Errors

- 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
- Can occur in languages that are not memory-safe
  - We'll discuss what "memory-safe" means
  - Several languages are not memory safe (C/C++) or allow unsafe uses (Rust)
  - These languages are used for high performance

Other Kinds of Flaws

- But, there are many other causes of flaws
  - Which typically involve other ways of mixing adversary-controlled input with code
  - Supply inputs that change the way the program executes (e.g., change the return address)
    - Without a memory error
- Includes (we will discuss later)
  - SQL injection
  - Cross-site scripting and request forgery
  - LLM prompt injections

Language Popularity

- Top 8 languages per the TIOBE index https://www.tiobe.com/tiobe-index/
  - 1. Python
  - 2. C++
  - 3. C
  - 4. Java
  - 5. C#
  - 6. JavaScript
  - 7. Visual Basic
  - 8. Go
- Other languages expend overhead to enforce memory safety (e.g., garbage collection)

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] );
}
```

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 must consider systematically what threats your programs face

Threat Model

- Can supply (write) the values from the command line – argv[] and 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] );
}
```

Example Code

- An adversary can supply the value "source" indirectly from the command line

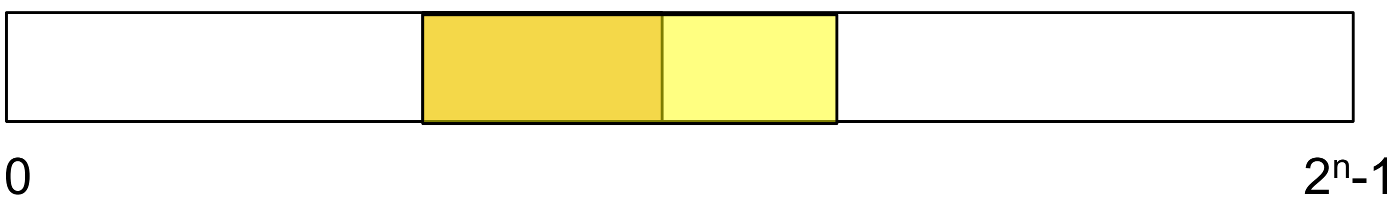
```
#include <stdio.h>

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?

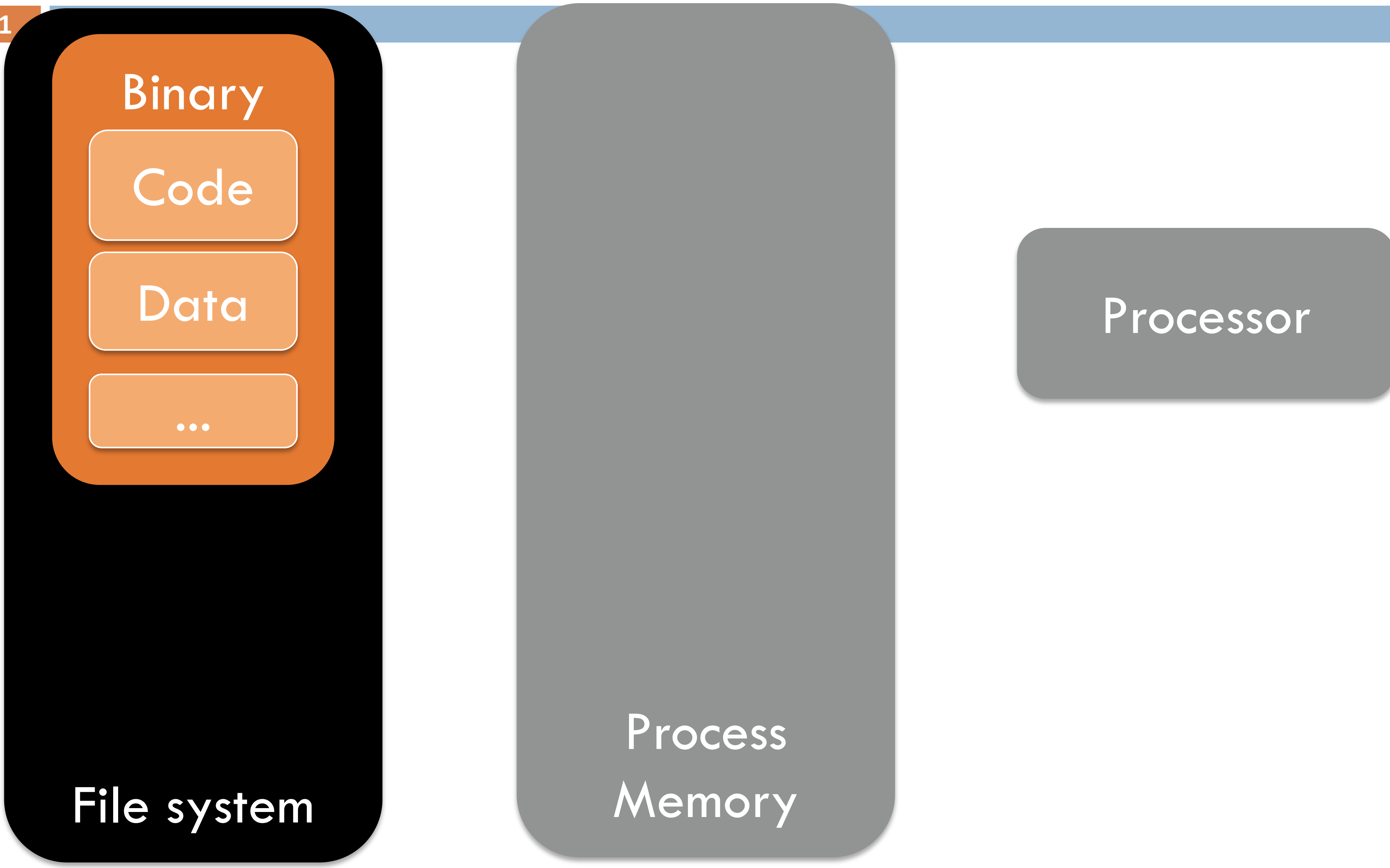
What Can Be Exploited?

- What is in the yellow memory area
  - Stack memory

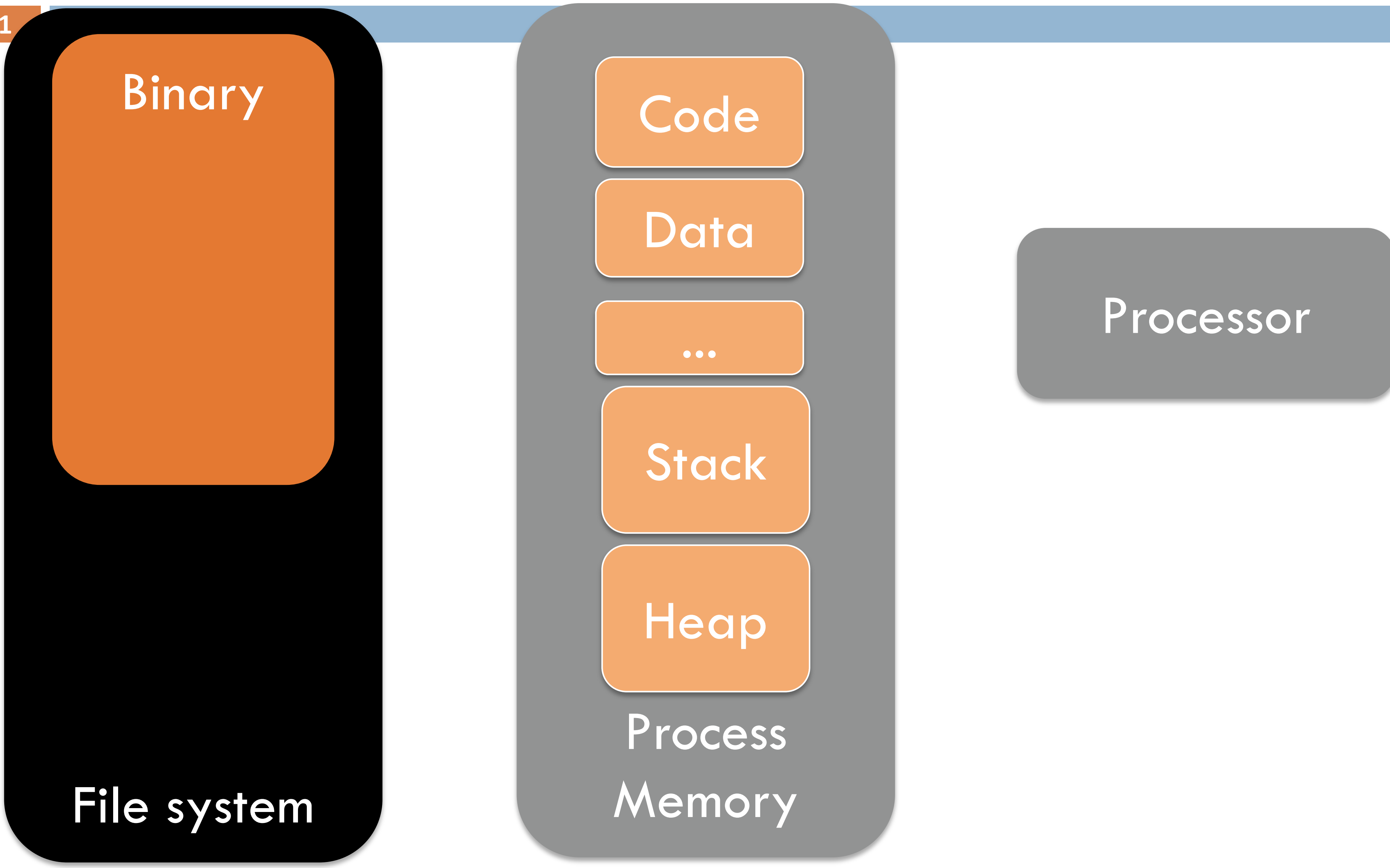


- What does stack memory look like?

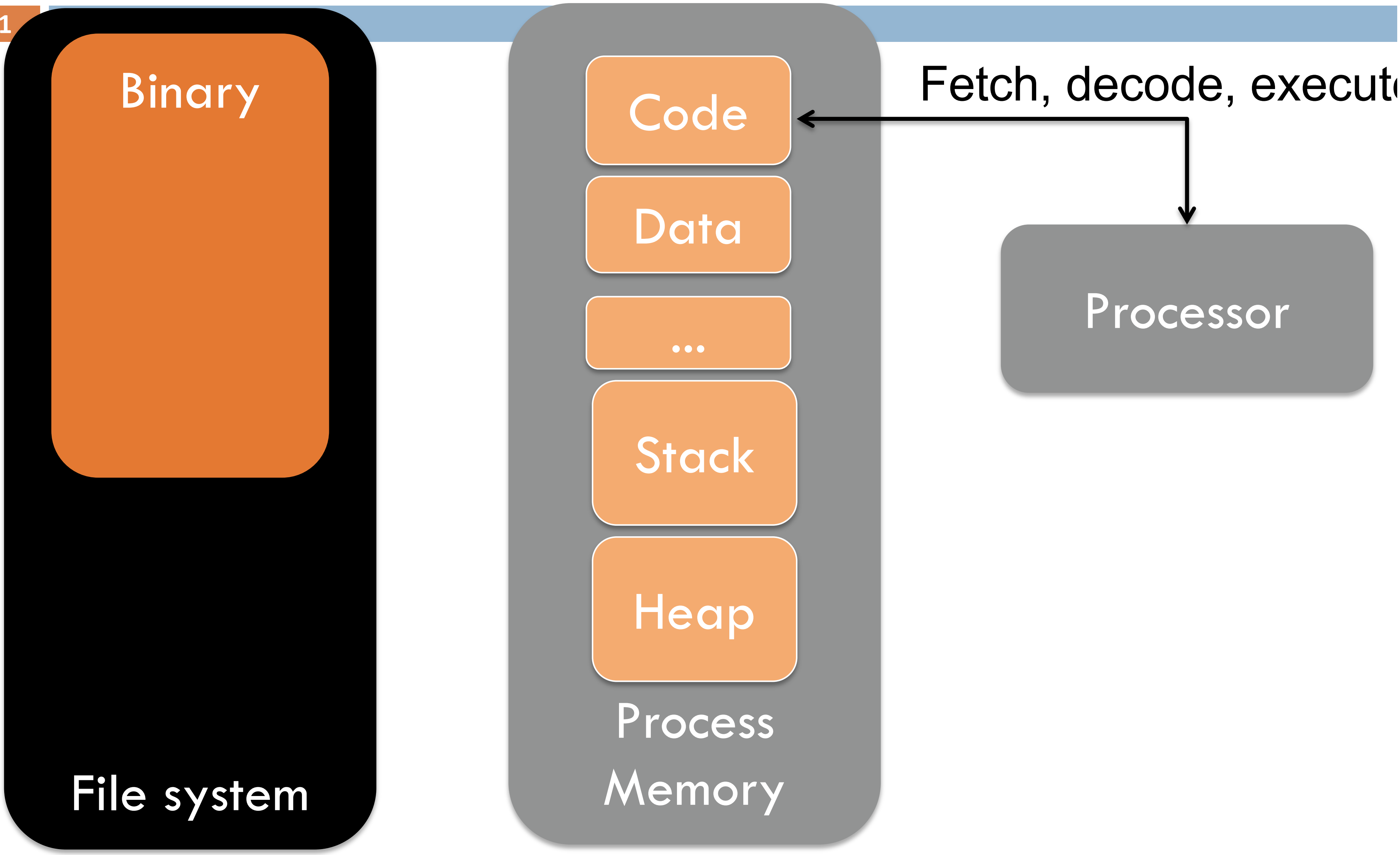
Basic Execution



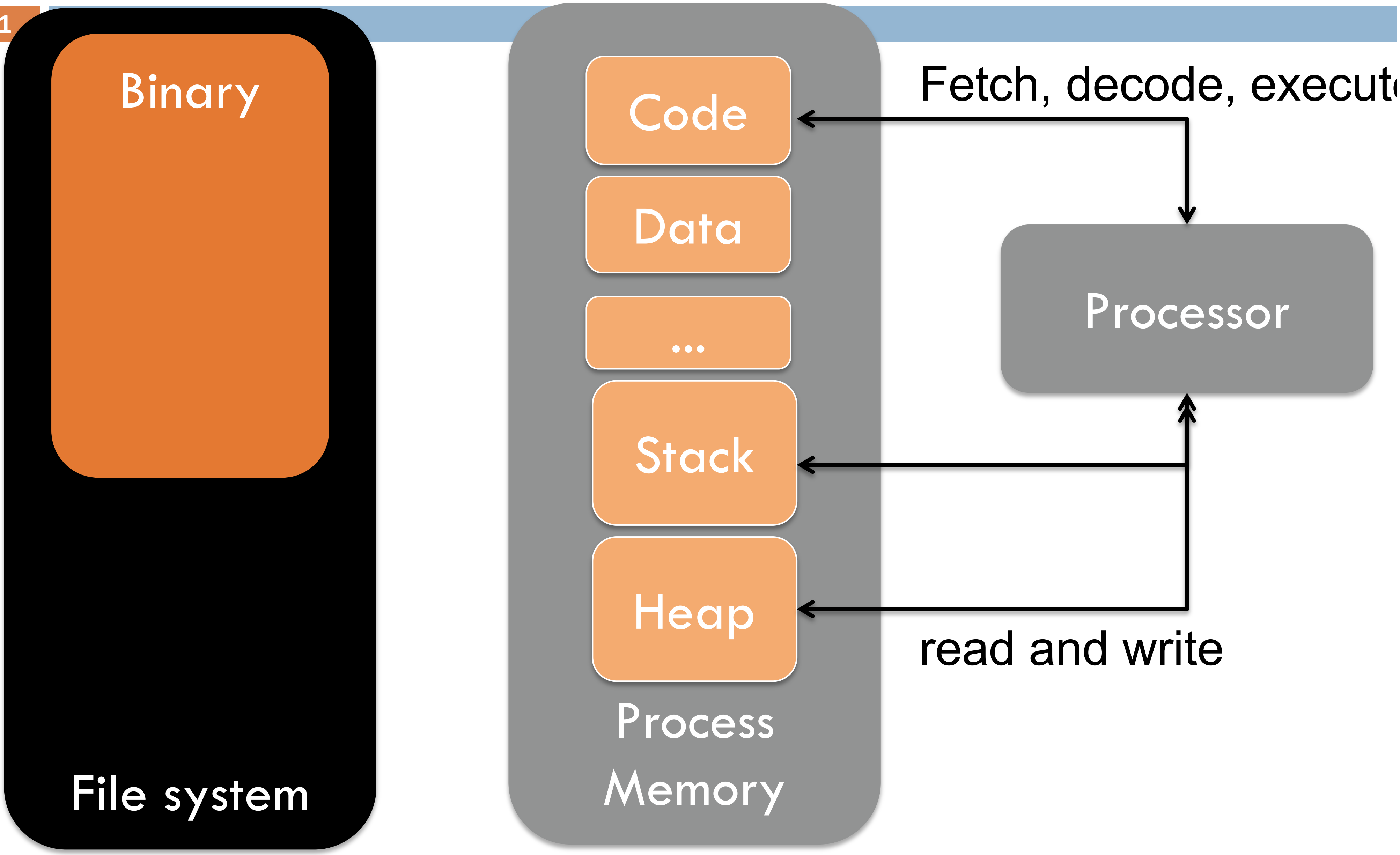
Basic Execution



Basic Execution

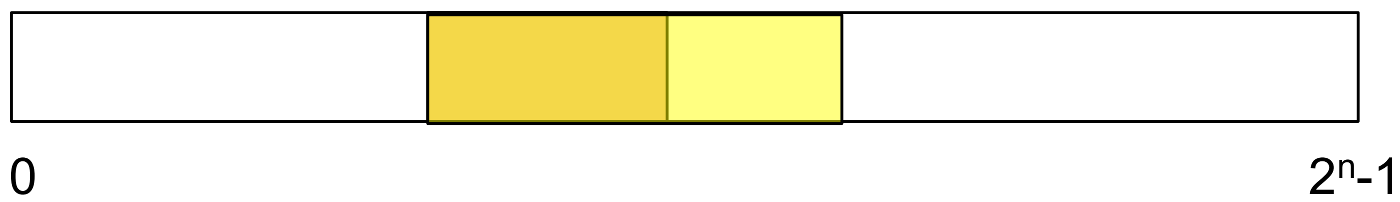


Basic Execution



What Can Be Exploited?

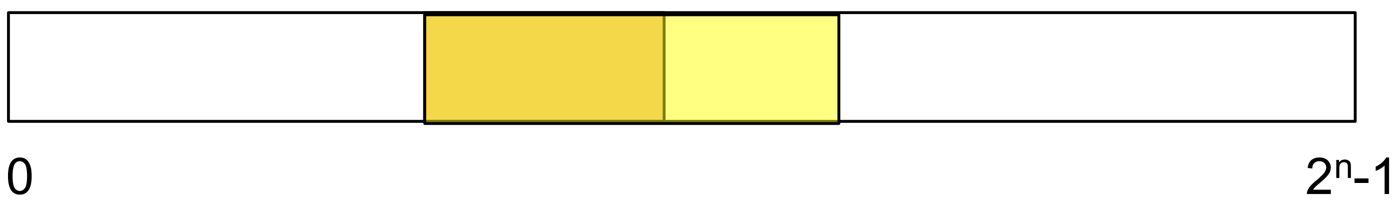
- What is in the yellow memory area
  - Stack memory



- What can be written illicitly by sscanf in this program?
  - Local variables: “buffer[10]” is the only one in this stack frame
  - No local variables in prior frame (main)

What Can Be Exploited?

- What is in the yellow memory area
  - Stack memory



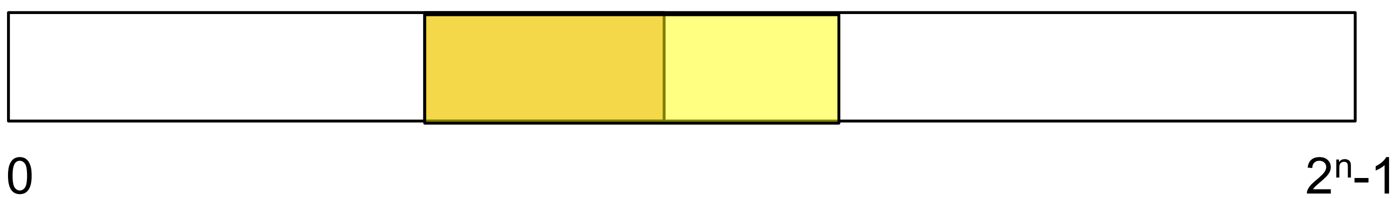
- What can be written illicitly by sscanf in this program?
  - Local variables: “buffer[10]” is the only one in this stack frame
  - But there is more – metadata controlling the program execution, such as
    - Frame pointer (ebp)
    - Return address

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

What Can Be Exploited?

- What is in the yellow memory area
  - Stack memory



- What happens if the return address is located in the yellow region of memory?
  - Change the return address
  - Change the address of the instruction run after the function returns
    - To any address the attacker wants
  - Can run arbitrary code of the program to violate (CIA)

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