# Memory Safety

## CS107 Additional Topics, 3/10/2023

Jerry Chen (with materials adapted from Will Crichton)

# What is memory safety?

# What is memory safety?

A ***memory safe*** program is one that has ***no memory errors***

# What are examples of memory errors?

# What are examples of memory errors?

- Buffer overflows

# What are examples of memory errors?

- Buffer overflows
- Invalid pointer

# What are examples of memory errors?

- Buffer overflows
- Invalid pointer
- Heap management

# What are examples of memory errors?

- Buffer overflows
- Invalid pointer
- Heap management
  - Use after free

# What are examples of memory errors?

- Buffer overflows
- Invalid pointer
- Heap management
    - Use after free
    - Double frees

# What are examples of memory errors?

- Buffer overflows
- Invalid pointer
- Heap management
  - Use after free
  - Double frees
  - Memory leaks (kind of)

# What are examples of memory errors?

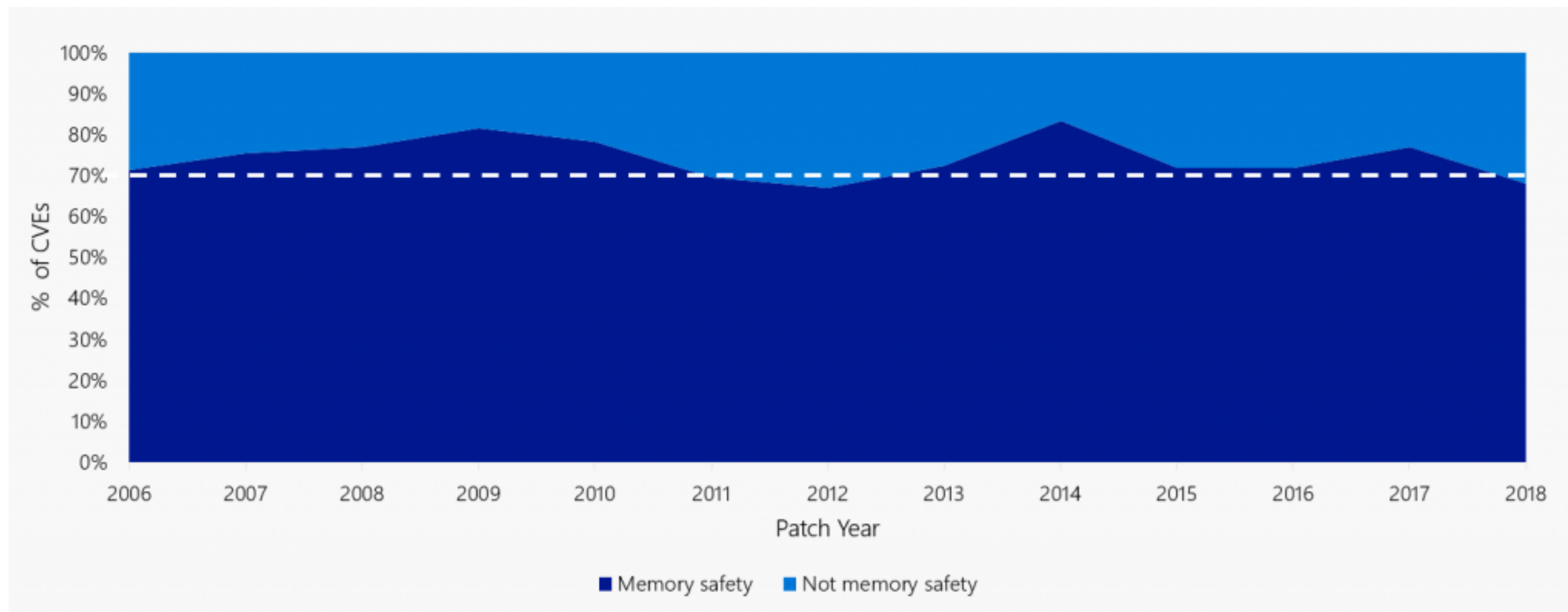
- Buffer overflows
- Invalid pointer
- Heap management
  - Use after free
  - Double frees
  - Memory leaks (kind of)

But you've taken CS107 and know how to find and prevent these errors!
Real-world systems can't *possibly* have these problems, right?

# Real-world systems can't *possibly* have these problems, right?

# Real-world systems can't *possibly* have these problems, right?

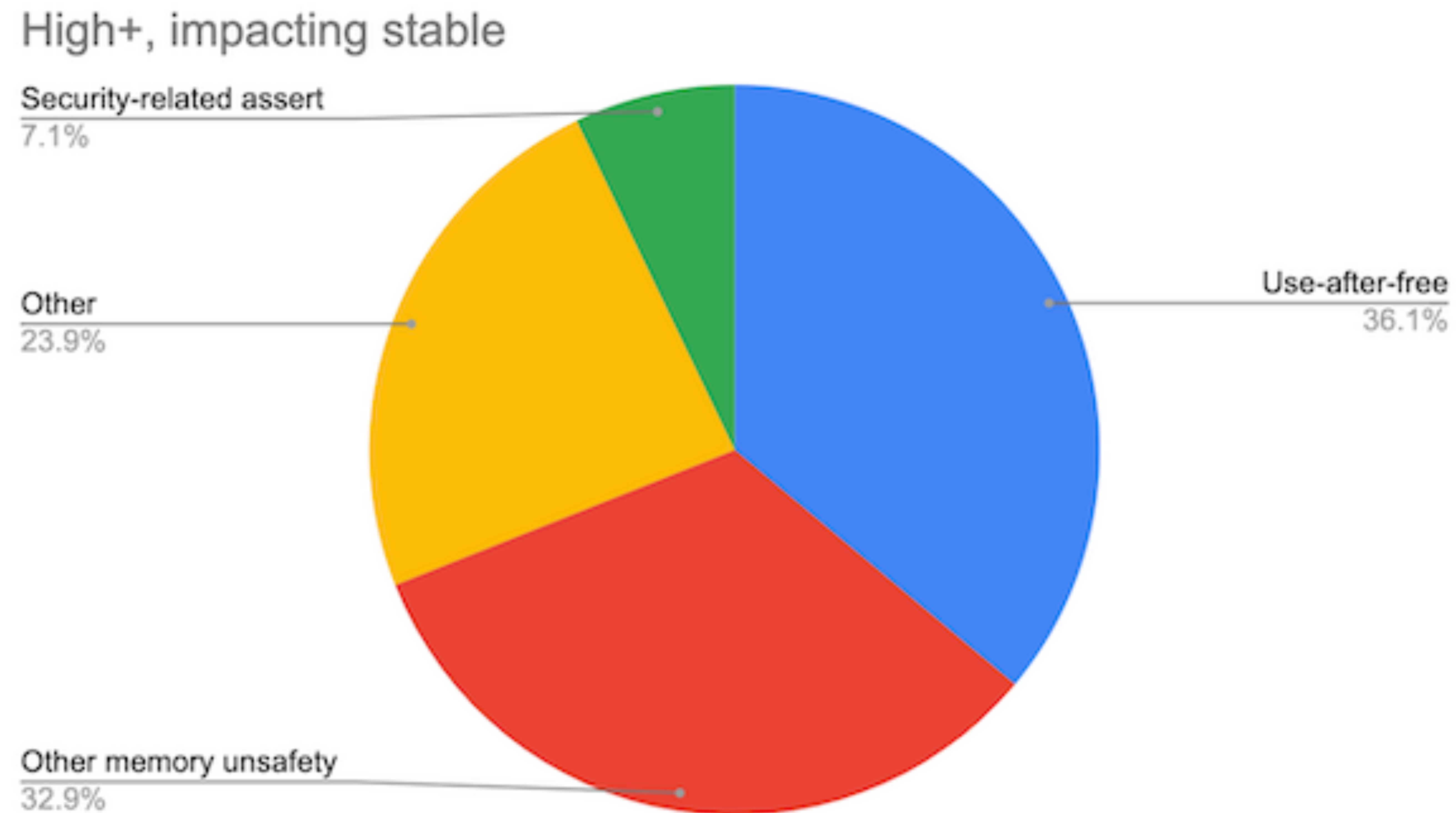
## 70% of all Microsoft vulnerabilities are memory errors




Microsoft Security Response Center. "A proactive approach to more secure code." 2019

# Real-world systems can't *possibly* have these problems, right?

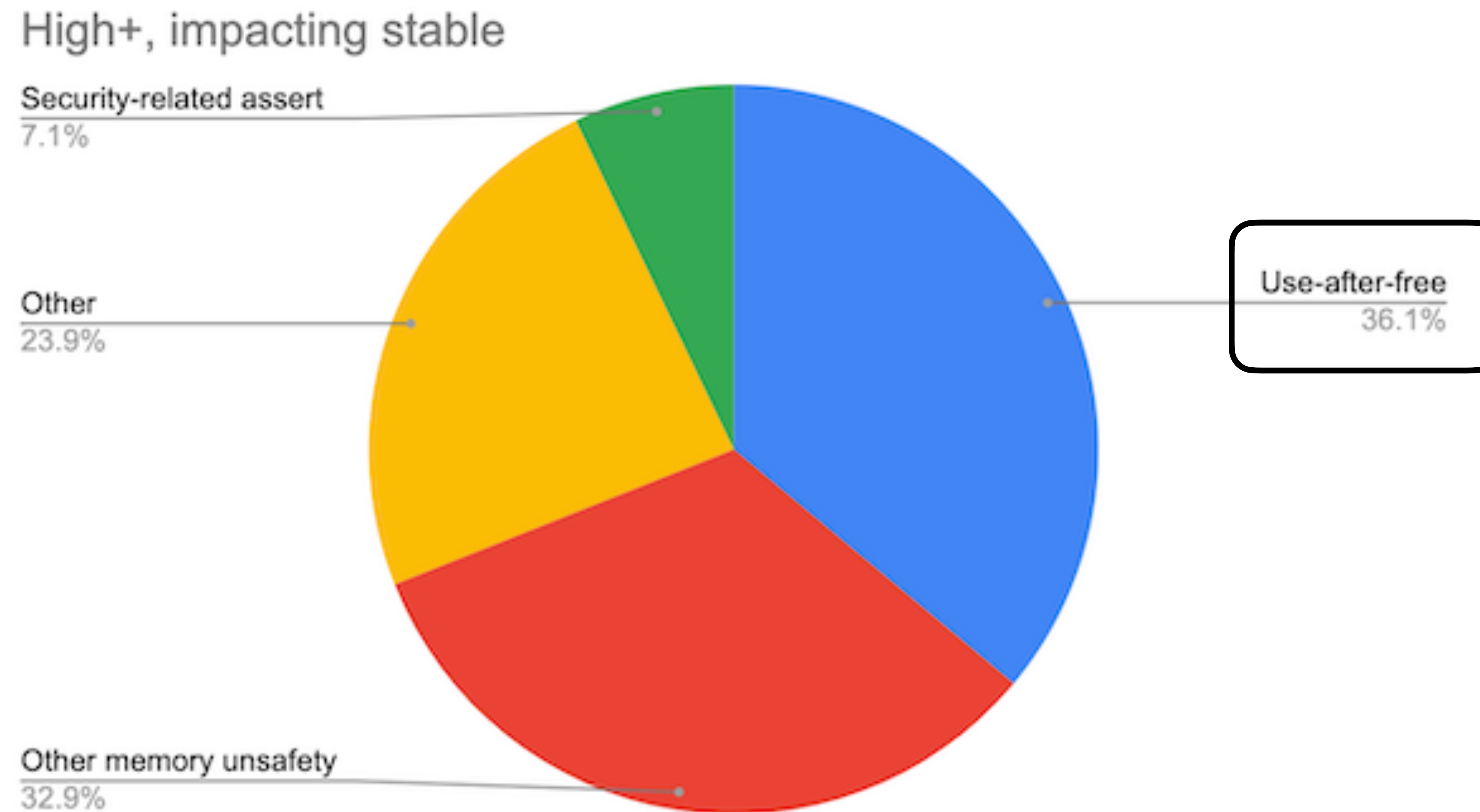
## 70% of severe security bugs in Google Chrome are memory errors




The Chromium Project. "Memory Safety."

# Real-world systems can't *possibly* have these problems, right?

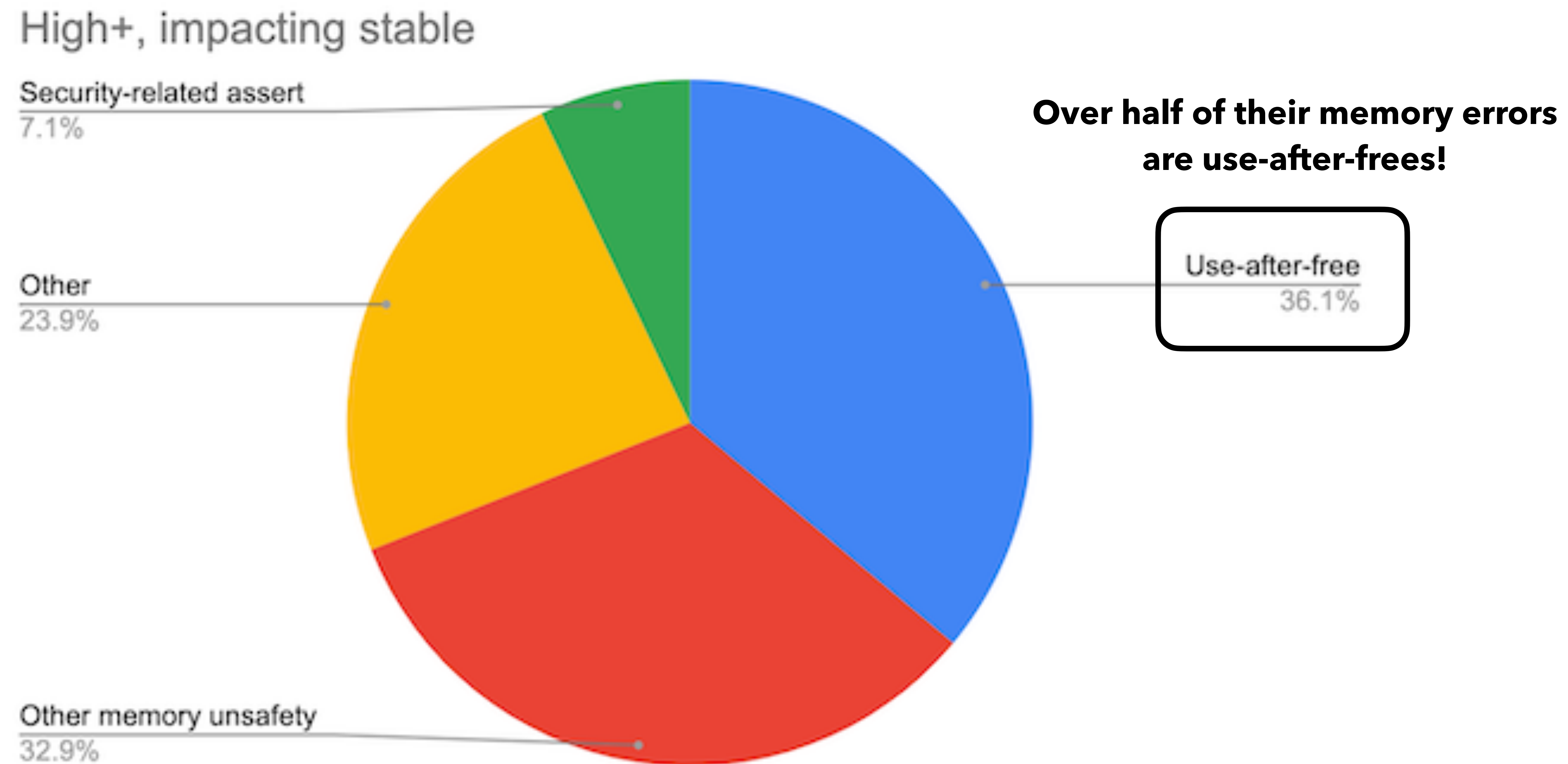
## 70% of severe security bugs in Google Chrome are memory errors




The Chromium Project. "Memory Safety."

# Real-world systems can't *possibly* have these problems, right?

## 70% of severe security bugs in Google Chrome are memory errors

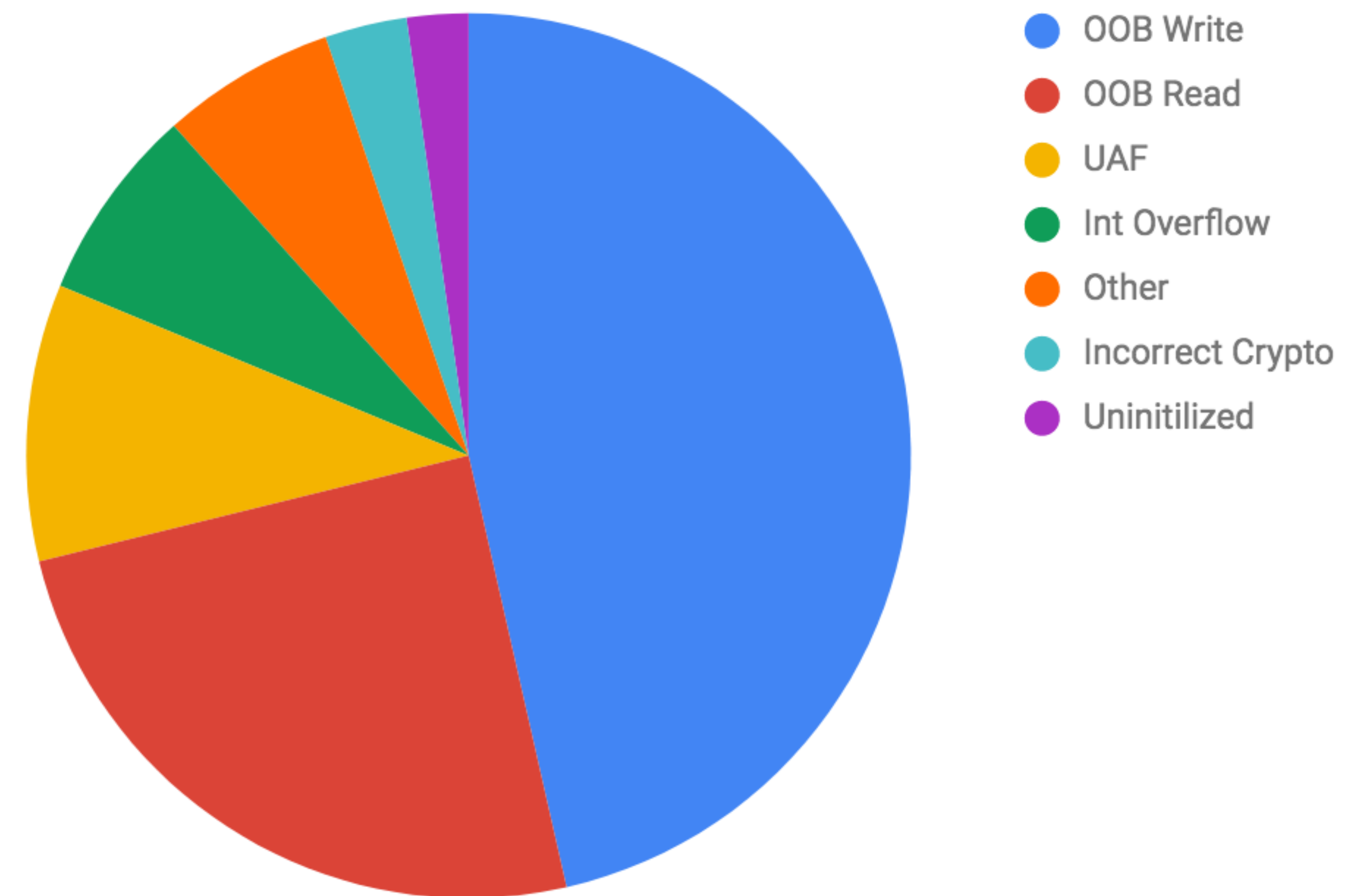



**Over half of their memory errors are use-after-frees!**

The Chromium Project. "Memory Safety."

# Real-world systems can't *possibly* have these problems, right?

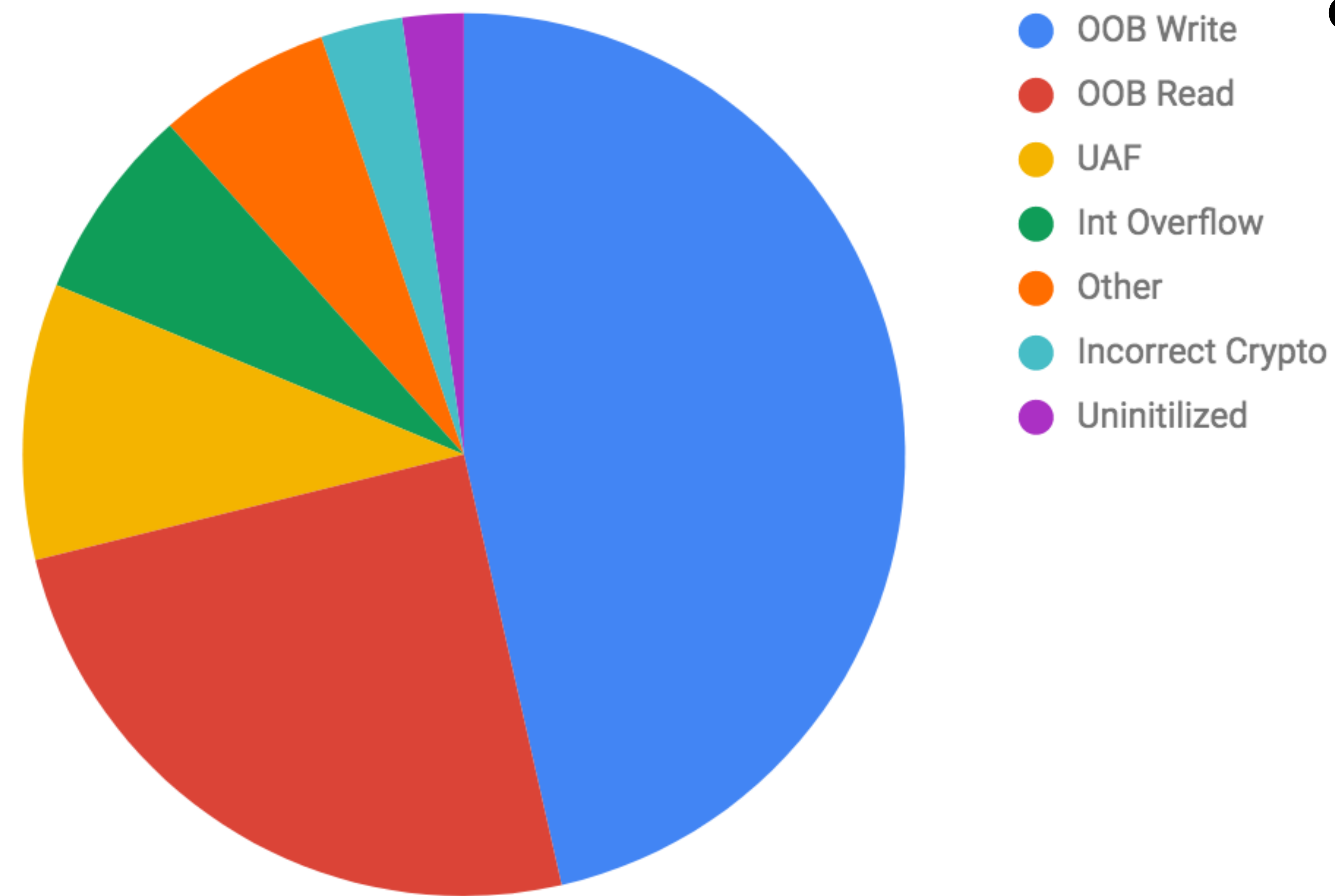
## 90% of Android vulnerabilities are memory errors




Google Security Blog. "Queue the Hardening Enhancements." 2019

# Real-world systems can't *possibly* have these problems, right?

## 90% of Android vulnerabilities are memory errors




**Out-of-bound writes comprise nearly half of all Android security bugs!**

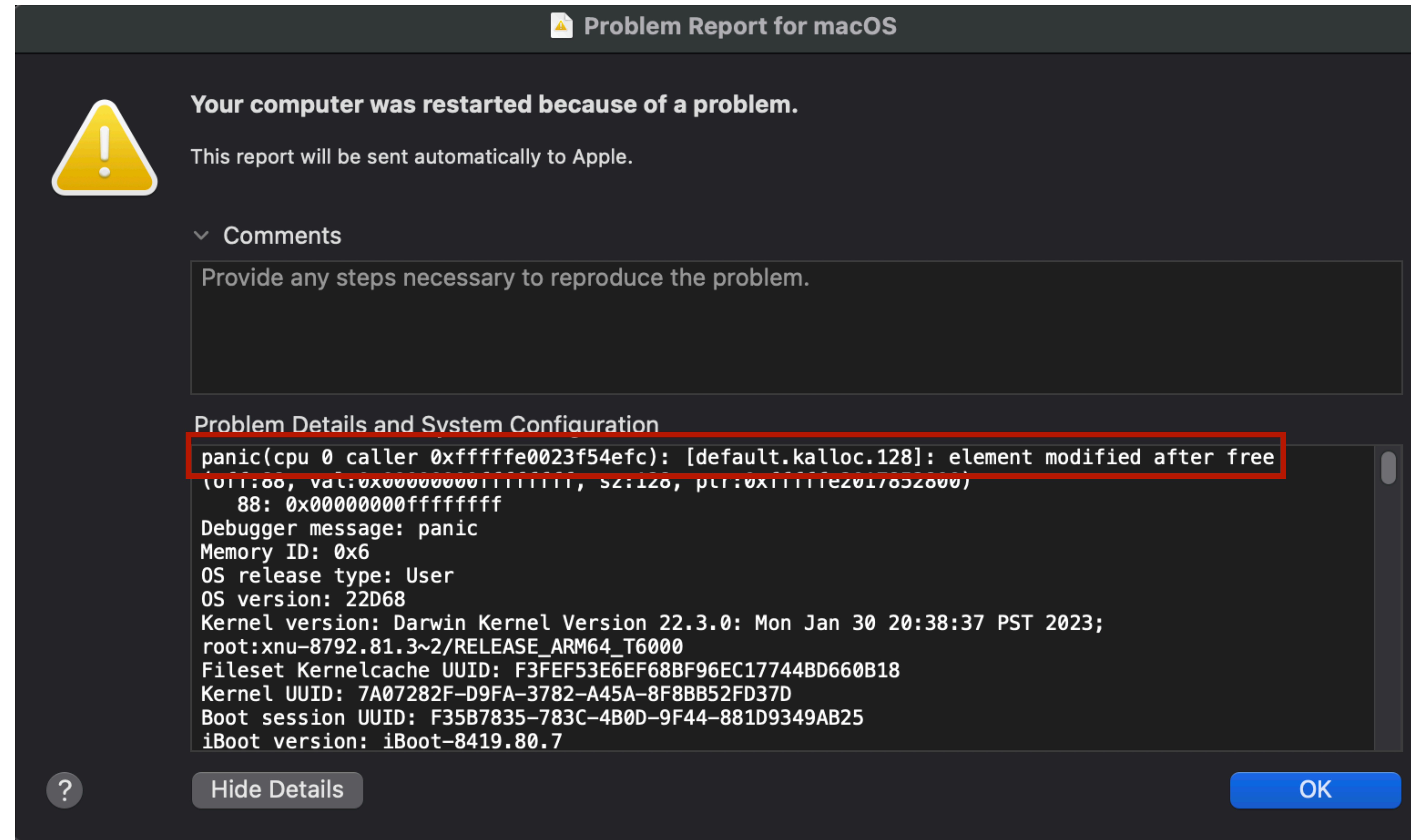
Google Security Blog. "Queue the Hardening Enhancements." 2019

## Real-world systems can't *possibly* have these problems, right?

**my laptop right as I was making the past few slides 😭**

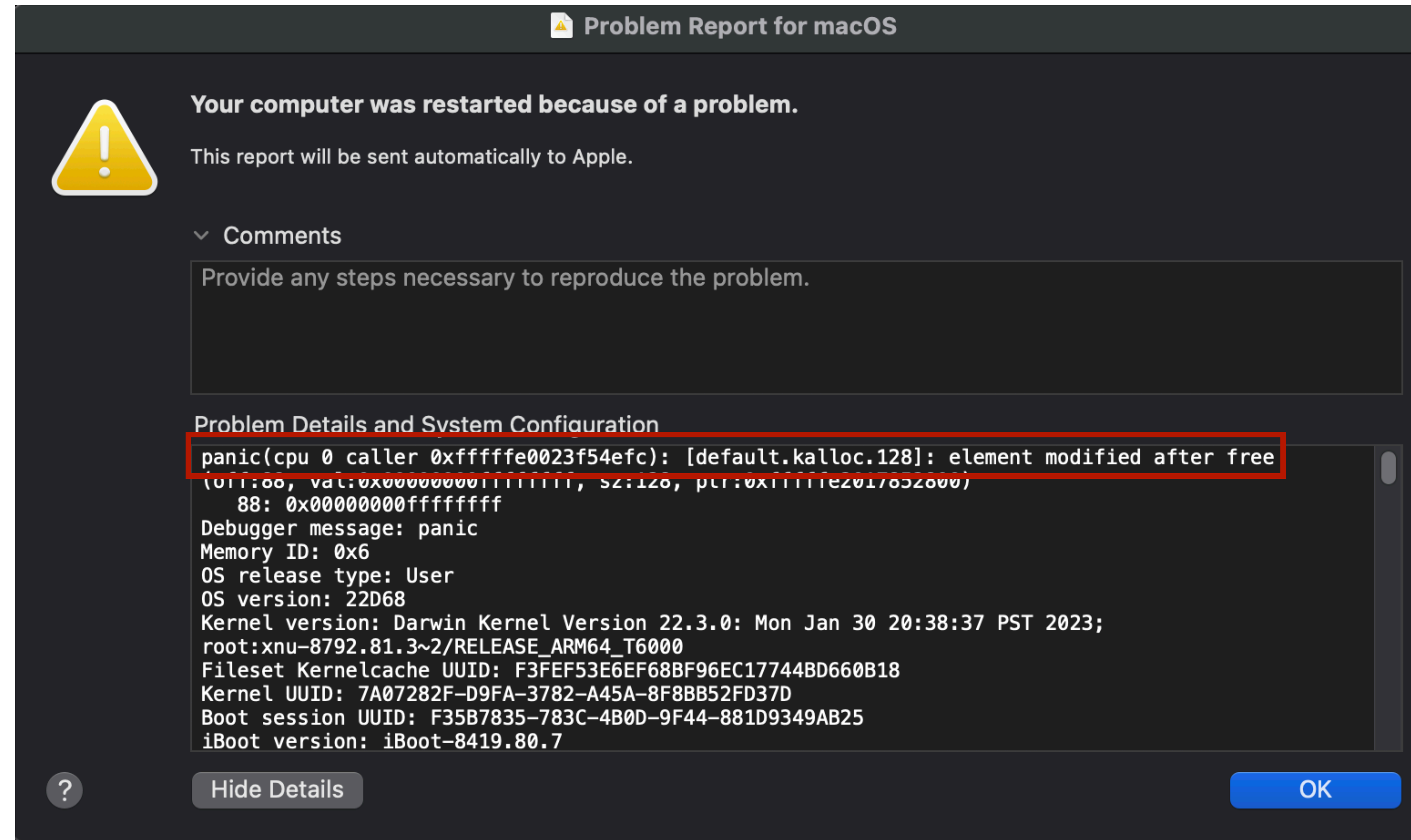
# Real-world systems can't *possibly* have these problems, right?

## my laptop right as I was making the past few slides 😭



# Real-world systems can't *possibly* have these problems, right?

## my laptop right as I was making the past few slides 😭




My laptop crashed because of a use-after-free bug!

# What can attackers do with memory errors?

**besides stealing money from an ATM**

# What can attackers do with memory errors?

## EternalBlue and WannaCry (2017)

# What can attackers do with memory errors?

## EternalBlue and WannaCry (2017)

- **EternalBlue:** vulnerability found and kept secret by the NSA until it was leaked in April 2017

# What can attackers do with memory errors?

## EternalBlue and WannaCry (2017)

- **EternalBlue:** vulnerability found and kept secret by the NSA until it was leaked in April 2017
- **WannaCry** was a May 2017 ransomware attack using EternalBlue
  - Shut down critical health services and infrastructure
  - Economic damages on the order of $1 billion

# What can attackers do with memory errors?

## EternalBlue and WannaCry (2017)

- **EternalBlue:** vulnerability found and kept secret by the NSA until it was leaked in April 2017
- **WannaCry** was a May 2017 ransomware attack using EternalBlue
  - Shut down critical health services and infrastructure
  - Economic damages on the order of $1 billion
- How? **Integer overflow** while determining much memory to allocate, leading to a **buffer overflow**

# What can attackers do with memory errors?

## Heartbleed (2014)

# What can attackers do with memory errors?

## Heartbleed (2014)

- **TLS (HTTPS)**: Internet protocol that keeps your internet activity private and secure

# What can attackers do with memory errors?

## Heartbleed (2014)

- **TLS (HTTPS)**: Internet protocol that keeps your internet activity private and secure
- **OpenSSL**: open-sourced implementation of TLS
    - “Heartbeat” mechanism where clients and servers periodically exchanged messages

# What can attackers do with memory errors?

## Heartbleed (2014)

- **TLS (HTTPS)**: Internet protocol that keeps your internet activity private and secure
- **OpenSSL**: open-sourced implementation of TLS
  - “Heartbeat” mechanism where clients and servers periodically exchanged messages
- **Heartbleed** vulnerability allowed attackers to read private data, including potentially passwords

# What can attackers do with memory errors?

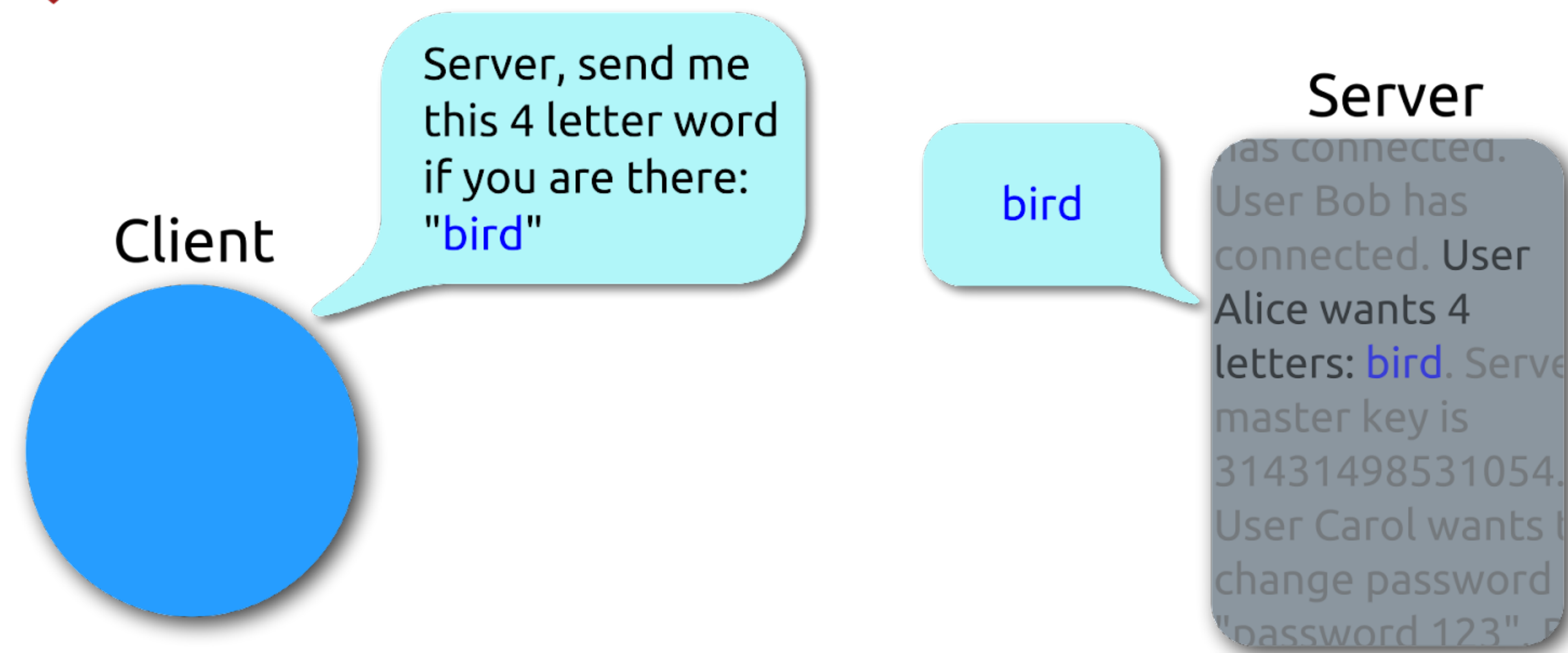
## Heartbleed (2014)

- **TLS (HTTPS)**: Internet protocol that keeps your internet activity private and secure
- **OpenSSL**: open-sourced implementation of TLS
  - “Heartbeat” mechanism where clients and servers periodically exchanged messages
- **Heartbleed** vulnerability allowed attackers to read private data, including potentially passwords
- How? **Missing bounds check,** leading to a **buffer over-read**

# What can attackers do with memory errors?

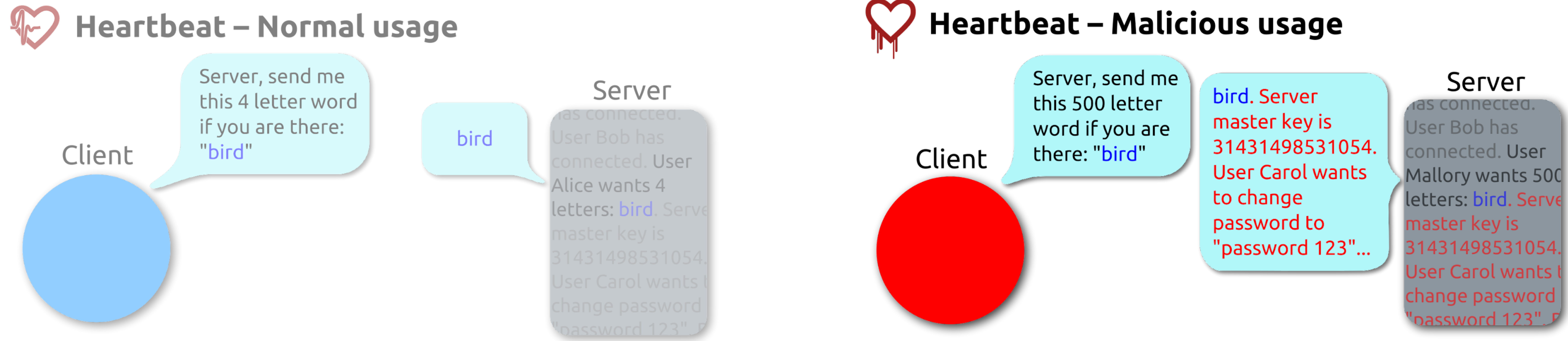
## Heartbleed (2014)

**Heartbeat – Normal usage**



# What can attackers do with memory errors?

## Heartbleed (2014)



# What can attackers do with memory errors?

## Heartbleed (2014)

Forbes

ENERGY

What Heartbleed Means for Critical Infrastructure

CNET Your guide to a better future

News > Privacy

'Heartbleed' bug undoes Web encryption, reveals Yahoo passwords

WIRED BACKCHANNEL BUSINESS CULTURE GEAR IDEAS SCIENCE SECURITY SIGN IN SUBSCRIBE

After 'Catastrophic' Security Bug, the Internet Needs a Password Reset

Security experts are calling Heartbleed, a bug in the internet's infrastructure, the worse thing they've seen in years. The bug is such problem, it may require what amounts to a massive password reset for the internet at large.

# Memory safety is difficult!

# Buggy Vector in C

## Struct Definition

```c
// Vec is short for "vector", a common term for a resizable array.
// For simplicity, our vector type can only hold ints.
typedef struct {
    int *data;     // Pointer to our array on the heap
    int length;    // How many elements are in our array
    int capacity; // How many elements our array can hold
} Vec;
```

# Buggy Vector in C

## What's wrong here?

```c
Vec *vec_create() {
    Vec vec;
    vec.data = malloc(sizeof(int));
    vec.length = 0;
    vec.capacity = 1;

    return &vec;
}
```

# Buggy Vector in C

## What's wrong here?

```c
Vec *vec_create() {
    Vec vec;
    vec.data = malloc(sizeof(int));
    vec.length = 0;
    vec.capacity = 1;

    return &vec;
}
```

**Our Vec is stack-allocated!**

We return a pointer, but that pointer will be invalid because the stack-allocated Vec will be destroyed when the function returns.

# Buggy Vector in C

## What's wrong here?

```
Vec *vec_create() {
    Vec vec;
    vec.data = malloc(sizeof(int));
    vec.length = 0;
    vec.capacity = 1;

    return &vec;
}
```

**Our Vec is stack-allocated!**

We return a pointer, but that pointer will be invalid because the stack-allocated Vec will be destroyed when the function returns.

```
Vec *vec_create() {
    Vec *vec = malloc(sizeof(Vec));
    vec->data = malloc(sizeof(int));
    vec->length = 0;
    vec->capacity = 1;

    return vec;
}
```

# Buggy Vector in C

## What's wrong here?

```
void vec_push(Vec *vec, int n) {
    // Double the capacity of our vector if it is full
    if (vec->length == vec->capacity) {
        vec->data = realloc(vec->data, vec->capacity * 2);
        assert(vec->data != NULL);

        vec->capacity *= 2;
    }

    // Append the element to the end of our vector
    vec->data[vec->length] = n;
    vec->length++;
}
```

# Buggy Vector in C

## What's wrong here?

```
void vec_push(Vec *vec, int n) {
    // Double the capacity of our vector if it is full
    if (vec->length == vec->capacity) {
        vec->data = realloc(vec->data, vec->capacity * 2);
        assert(vec->data != NULL);

        vec->capacity *= 2;
    }

    // Append the element to the end of our vector
    vec->data[vec->length] = n;
    vec->length++;
}
```

**Not realloc'ing with the correct size!**

realloc requires the number of bytes, not elements

Multiply by sizeof(int)

# Buggy Vector in C

## What's wrong here?

```
void main() {
    Vec *vec = vec_create();
    vec_push(vec, 107);
    int *n = &vec->data[0];

    vec_push(vec, 111);
    printf("%d\n", *n);

    free(vec);
}
```

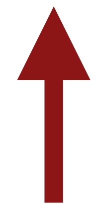
# Buggy Vector in C

## What's wrong here?

```
void main() {
    Vec *vec = vec_create();
    vec_push(vec, 107);
    int *n = &vec->data[0];

    vec_push(vec, 111);
    printf("%d\n", *n);

    free(vec);
}
```



**1. Memory Leak: need to free vec->data first**

# Buggy Vector in C

## What's wrong here?

```
void main() {
    Vec *vec = vec_create();
    vec_push(vec, 107);
    int *n = &vec->data[0];

    vec_push(vec, 111);
    printf("%d\n", *n);

    free(vec);
}
```

# Buggy Vector in C

## What's wrong here?

```
void main() {
    Vec *vec = vec_create();
    vec_push(vec, 107);
    int *n = &vec->data[0];

    vec_push(vec, 111);
    printf("%d\n", *n);

    free(vec);
}
```

**2. Is it safe to dereference the n pointer here?**

**No!** vec_push() will resize the data array by calling realloc(). This can **invalidate** the pointer to old memory!

**1. Memory Leak: need to free vec->data first**

# C/C++ are inherently unsafe

# C/C++ are inherently unsafe

# C/C++ are inherently unsafe

- **Extremely powerful**
  - Low-level systems programming
  - Manual memory management
  - Casting and reinterpreting raw bytes

# C/C++ are inherently unsafe

- **Extremely powerful**
  - Low-level systems programming
  - Manual memory management
  - Casting and reinterpreting raw bytes
- **Extremely dangerous**
  - Low-level systems programming
  - Manual memory management
  - Casting and reinterpreting raw bytes

# Memory safe languages

# Memory safe languages

## Run-time memory safety

# Memory safe languages

## Run-time memory safety

- There are many higher-level languages, such as Java, Python, and Go, that guarantee memory safety at *run-time*

# Memory safe languages

## Run-time memory safety

- There are many higher-level languages, such as Java, Python, and Go, that guarantee memory safety at *run-time*
- **Run-time checks**: crash the program if a memory error is ever detected

# Memory safe languages

## Run-time memory safety

- There are many higher-level languages, such as Java, Python, and Go, that guarantee memory safety at *run-time*
- **Run-time checks**: crash the program if a memory error is ever detected
  - Index out-of-bounds

# Memory safe languages

## Run-time memory safety

- There are many higher-level languages, such as Java, Python, and Go, that guarantee memory safety at *run-time*
- **Run-time checks**: crash the program if a memory error is ever detected
  - Index out-of-bounds
  - Type mismatches

# Memory safe languages

## Run-time memory safety

- There are many higher-level languages, such as Java, Python, and Go, that guarantee memory safety at *run-time*
- **Run-time checks**: crash the program if a memory error is ever detected
    - Index out-of-bounds
    - Type mismatches
    - Invalid pointer dereferences

# Memory safe languages

## Run-time memory safety

- There are many higher-level languages, such as Java, Python, and Go, that guarantee memory safety at *run-time*
- **Run-time checks**: crash the program if a memory error is ever detected
    - Index out-of-bounds
    - Type mismatches
    - Invalid pointer dereferences
- **Garbage collection**: memory management is abstracted away by the language

# Memory safe languages

## Run-time memory safety

- There are many higher-level languages, such as Java, Python, and Go, that guarantee memory safety at *run-time*
- **Run-time checks**: crash the program if a memory error is ever detected
  - Index out-of-bounds
  - Type mismatches
  - Invalid pointer dereferences
- **Garbage collection**: memory management is abstracted away by the language
  - As a program executes, it periodically sweeps through and reclaims memory that was previously allocated but no longer used

# Memory safe languages

## Run-time memory safety

- There are many higher-level languages, such as Java, Python, and Go, that guarantee memory safety at *run-time*
- **Run-time checks**: crash the program if a memory error is ever detected
  - Index out-of-bounds
  - Type mismatches
  - Invalid pointer dereferences
- **Garbage collection**: memory management is abstracted away by the language
  - As a program executes, it periodically sweeps through and reclaims memory that was previously allocated but no longer used
- **As the programmer, you never have to worry about memory errors!**

# Memory safe languages

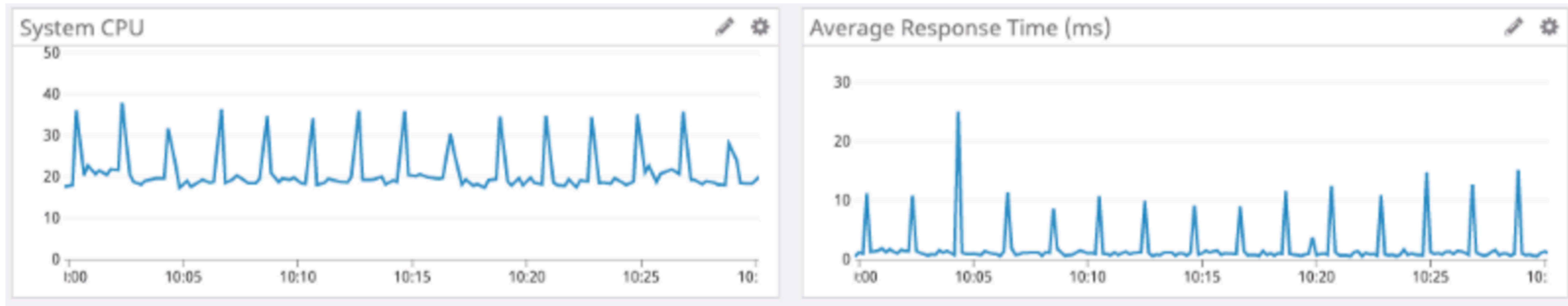
## Run-time memory safety

- There are many higher-level languages, such as Java, Python, and Go, that guarantee memory safety at *run-time*
- **Run-time checks**: crash the program if a memory error is ever detected
  - Index out-of-bounds
  - Type mismatches
  - Invalid pointer dereferences
- **Garbage collection**: memory management is abstracted away by the language
  - As a program executes, it periodically sweeps through and reclaims memory that was previously allocated but no longer used
- **As the programmer, you never have to worry about memory errors!**
  - Sometimes at the cost of performance…

# Memory safe languages

## Cost of runtime safety

This graph shows a couple performance metrics for one of Discord's core services to keep track of what messages a user has read (lower is better)

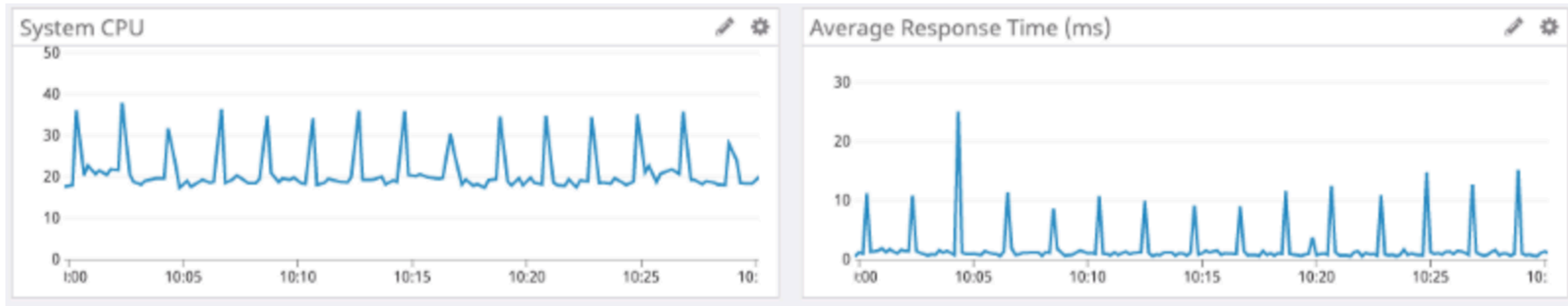



Jesse Howarth. "Why Discord is Switching from Go to Rust." 2020.

# Memory safe languages

## Cost of runtime safety

This graph shows a couple performance metrics for one of Discord's core services to keep track of what messages a user has read (lower is better)




**Each of the spikes represents the garbage collector running every couple minutes!**

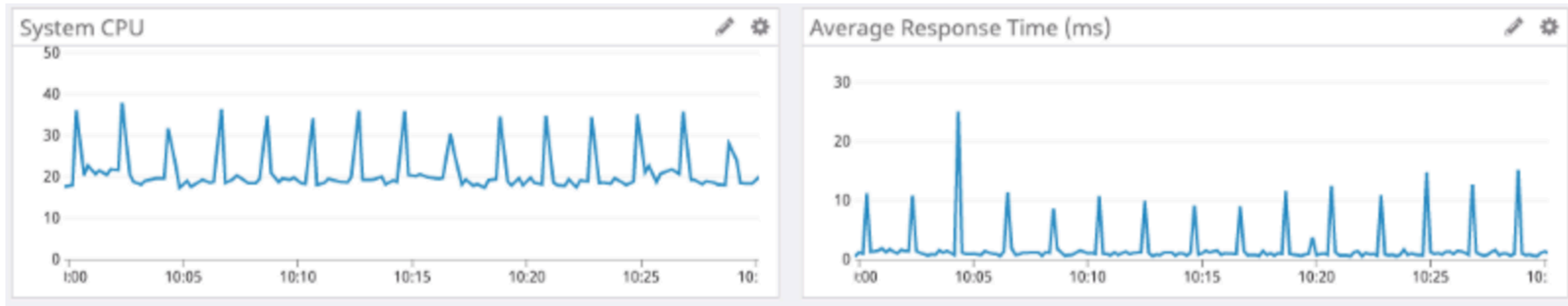
Can we achieve memory safety without sacrificing performance?

Jesse Howarth. "Why Discord is Switching from Go to Rust." 2020.

# Memory safe languages

## Cost of runtime safety

This graph shows a couple performance metrics for one of Discord's core services to keep track of what messages a user has read (lower is better)




**Each of the spikes represents the garbage collector running every couple minutes!**

Can we achieve memory safety without sacrificing performance?

Jesse Howarth. "Why Discord is Switching from Go to Rust." 2020.

# Memory safe languages

## Compile-time memory safety: Rust

# Memory safe languages

## Compile-time memory safety: Rust

- Safety is encoded directly into the syntax and grammar of the language

# Memory safe languages

## Compile-time memory safety: Rust

- Safety is encoded directly into the syntax and grammar of the language
- The compiler can verify whether code is safe to run before even trying to translate it to assembly for the computer to run

# Memory safe languages

## Compile-time memory safety: Rust

- Safety is encoded directly into the syntax and grammar of the language
- The compiler can verify whether code is safe to run before even trying to translate it to assembly for the computer to run
- **Minimal run-time cost**: memory management and safety is handled at compile-time

# Memory safe languages

## Compile-time memory safety: Rust

- Safety is encoded directly into the syntax and grammar of the language
- The compiler can verify whether code is safe to run before even trying to translate it to assembly for the computer to run
- **Minimal run-time cost**: memory management and safety is handled at compile-time
  - Comparable performance to C++ with none of the potential bugs!

# Memory safe languages

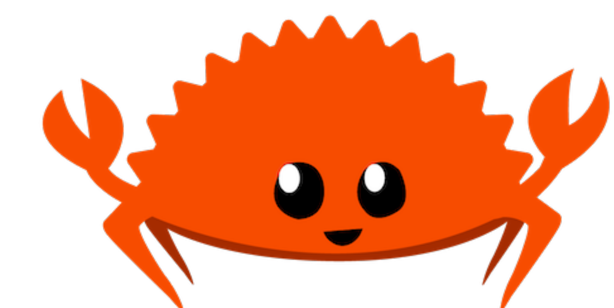
## Compile-time memory safety: Rust

- Safety is encoded directly into the syntax and grammar of the language
- The compiler can verify whether code is safe to run before even trying to translate it to assembly for the computer to run
- **Minimal run-time cost**: memory management and safety is handled at compile-time
  - Comparable performance to C++ with none of the potential bugs!
- Growing consensus that Rust should be the language of choice when building new performance- and safety-critical applications

# Memory safe languages

## Compile-time memory safety: Rust

- Safety is encoded directly into the syntax and grammar of the language
- The compiler can verify whether code is safe to run before even trying to translate it to assembly for the computer to run
- **Minimal run-time cost**: memory management and safety is handled at compile-time
    - Comparable performance to C++ with none of the potential bugs!
- Growing consensus that Rust should be the language of choice when building new performance- and safety-critical applications



# Rust Ownership Model

Rust adopts a unique approach to memory management known as ownership

# Rust Ownership Model

Rust adopts a unique approach to memory management known as ownership

- Each value in memory has a variable called its *owner*

# Rust Ownership Model

Rust adopts a unique approach to memory management known as ownership

- Each value in memory has a variable called its *owner*
- A value can only have one owner at a time

# Rust Ownership Model

Rust adopts a unique approach to memory management known as ownership

- Each value in memory has a variable called its *owner*
- A value can only have one owner at a time
- When the owner goes out of scope, the value is “dropped” (deallocated)

# Rust Ownership Model

```
{
    let s1: String = String::from("Hello");
    println!("{}", s1);
}
```

**Output:**

```
Hello
```

# Rust Ownership Model

```
{
    let s1: String = String::from("Hello");
    println!("{}", s1);
}
```

**Stack** **Heap**

**Output:**

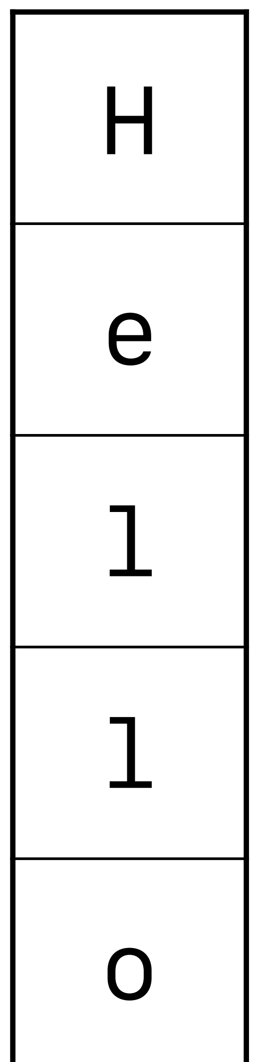
Hello

# Rust Ownership Model

```rust
{
    let s1: String = String::from("Hello");
    println!("{}", s1);
}
```

**Stack**

**Heap**



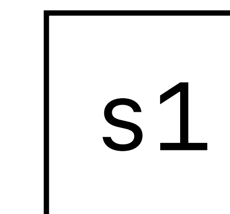

**Output:**

Hello

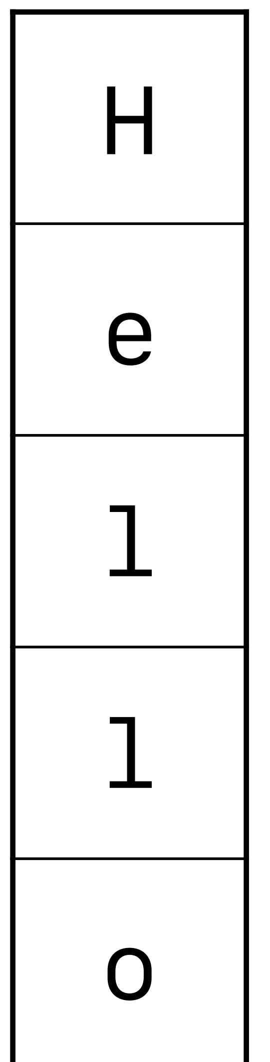
# Rust Ownership Model

```
{
    let s1: String = String::from("Hello");
    println!("{}", s1);
}
```

**Stack**




**Heap**

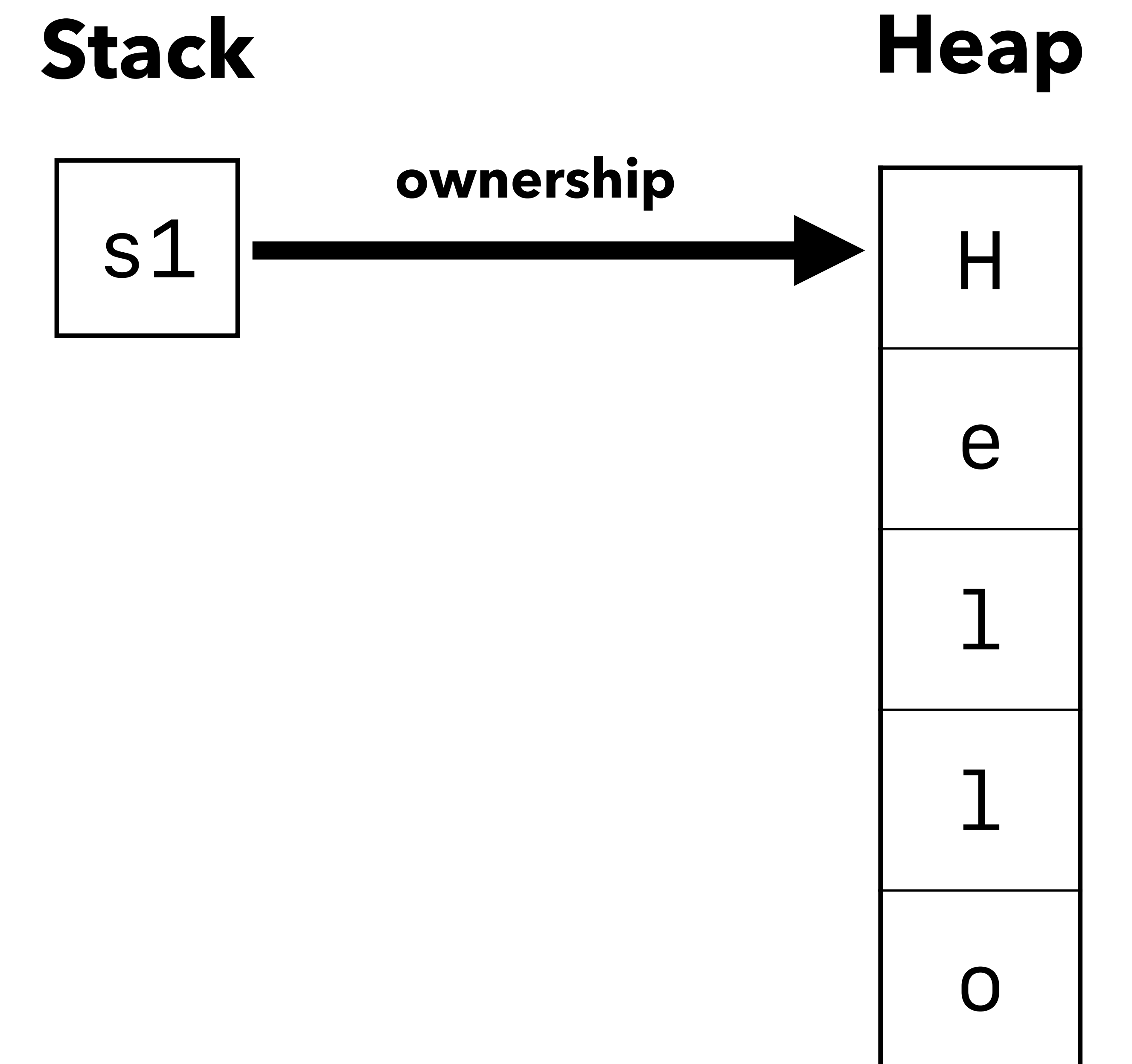



**Output:**

Hello

# Rust Ownership Model

```rust
{
    let s1: String = String::from("Hello");
    println!("{}", s1);
}
```

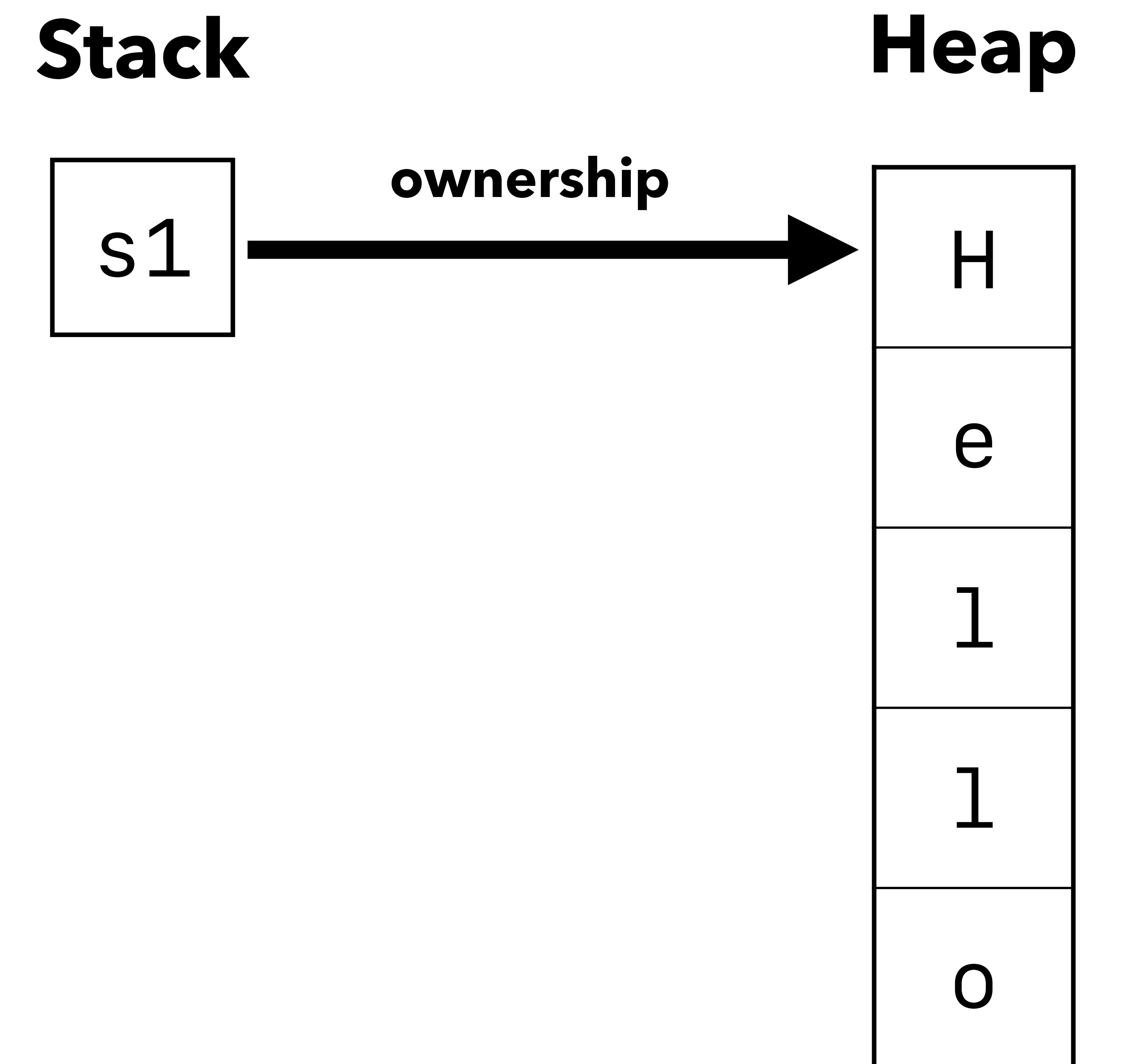



**Output:**
Hello

# Rust Ownership Model

```
{
    let s1: String = String::from("Hello");
    println!("{}", s1);
}
```

Compiler sees end of scope and drops s1 to deallocate the string. Hooray for no memory leaks!




**Output:**

```
Hello
```

# Rust Ownership Model

## Preventing use-after-free

```
{
    let s1: String = String::from("Hello");
    drop(s1);
    println!("{}", s1);
}
```

# Rust Ownership Model

## Preventing use-after-free

```
{
    let s1: String = String::from("Hello");
    drop(s1);
    println!("{}", s1);
}
```

```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:24
  |
3 |         let s1: String = String::from("Hello");
  |             -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
4 |         drop(s1);
  |              -- value moved here
5 |         println!("{}", s1);
  |                        ^^ value borrowed here after move
  |
```

# Rust Ownership Model

## This code won’t compile!

```
{
    let s1: String = String::from(“Hello");
    let s2: String = s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

# Rust Ownership Model

## This code won't compile!

```
{
    let s1: String = String::from(“Hello");
    let s2: String = s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:24
  |
3 |         let s1: String = String::from("Hello");
  |             -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
4 |         let s2: String = s1;
  |                          -- value moved here
5 |         println!("{}", s1);
  |                        ^^ value borrowed here after move
  |
```

# Rust Ownership Model

## This code won't compile!

```
{
    let s1: String = String::from(“Hello");
    let s2: String = s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

**Values are “moved” by default: ownership is transferred to s2**

```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:24
  |
3 |         let s1: String = String::from("Hello");
  |             -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
4 |         let s2: String = s1;
  |                          -- value moved here
5 |         println!("{}", s1);
  |                        ^^ value borrowed here after move
  |
```

# Rust Ownership Model

## This code won't compile!

```
{
    let s1: String = String::from("Hello");
    let s2: String = s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

Values are "moved" by default: ownership is transferred to s2

**Stack** **Heap**

```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:24
  |
3 |         let s1: String = String::from("Hello");
  |             -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
4 |         let s2: String = s1;
  |                          -- value moved here
5 |         println!("{}", s1);
  |                        ^^ value borrowed here after move
  |
```

# Rust Ownership Model

## This code won't compile!

```
{
    let s1: String = String::from("Hello");
    let s2: String = s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

Values are "moved" by default: ownership is transferred to s2

Stack

Heap

| H |
|---|
| e |
| l |
| l |
| o |

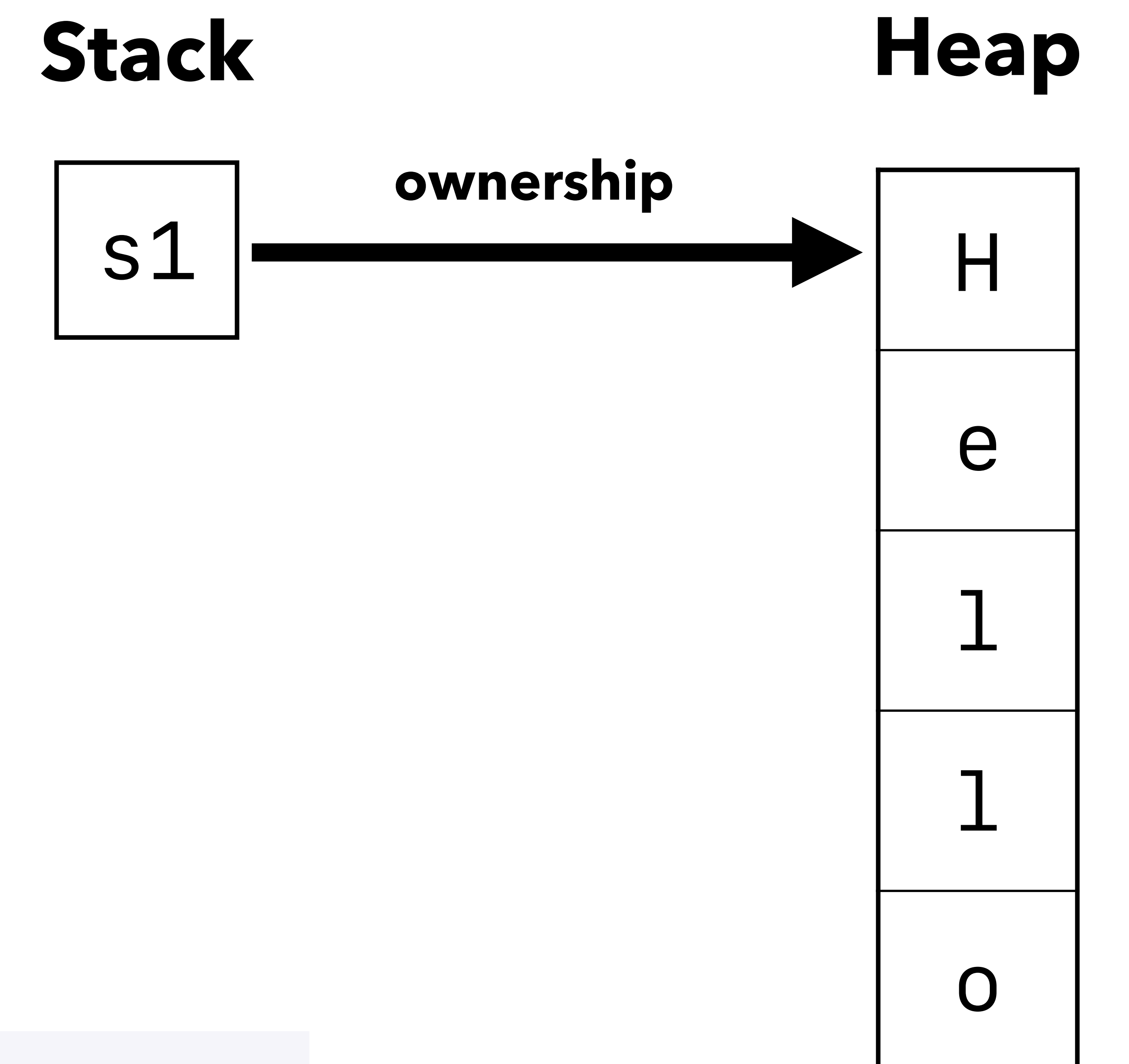
```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:24
  |
3 |         let s1: String = String::from("Hello");
  |             -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
4 |         let s2: String = s1;
  |                          -- value moved here
5 |         println!("{}", s1);
  |                        ^^ value borrowed here after move
  |
```

# Rust Ownership Model

## This code won’t compile!

```
{
    let s1: String = String::from(“Hello");
    let s2: String = s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

Values are “moved” by default: ownership is transferred to s2



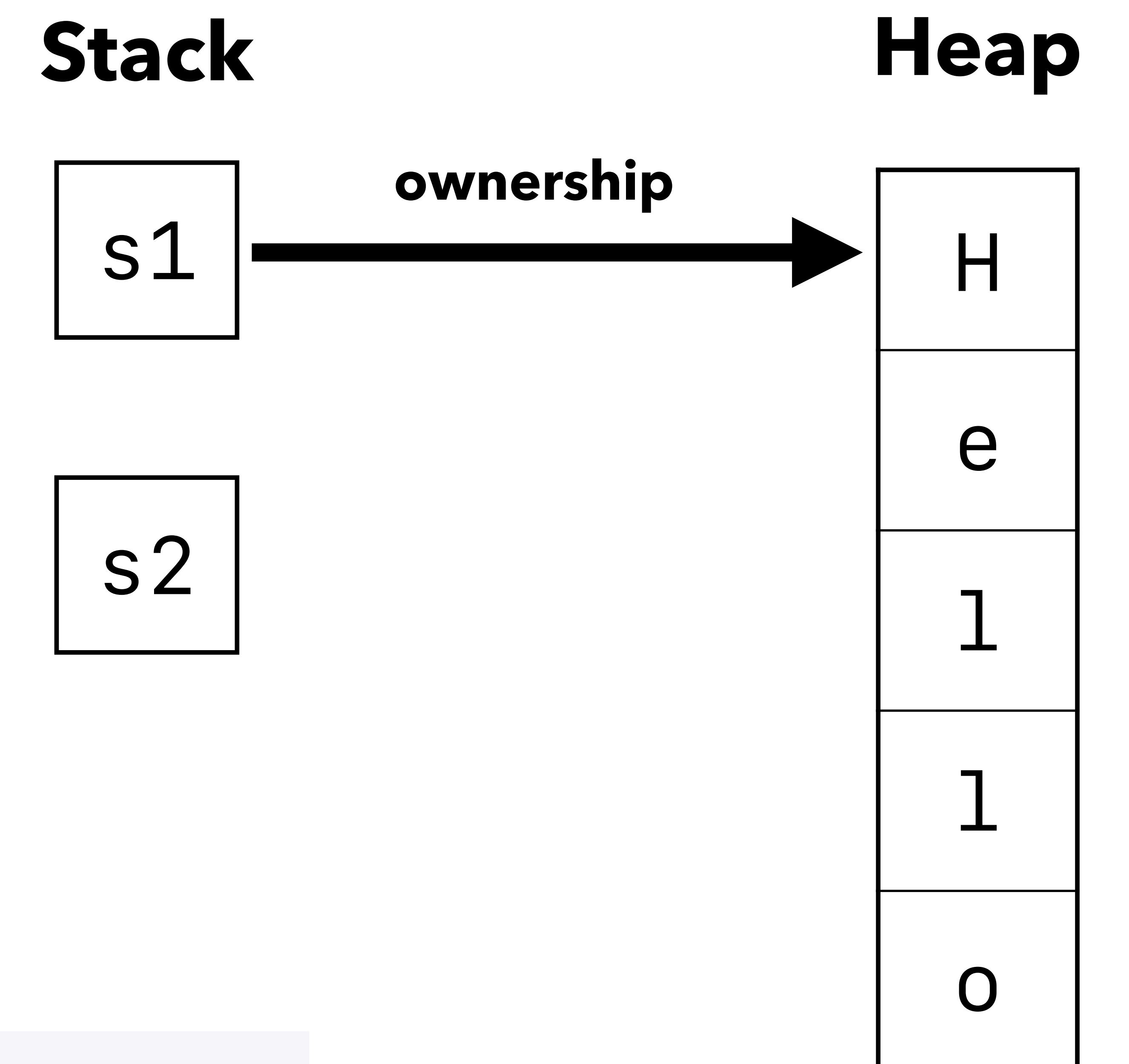

```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:24
  |
3 |         let s1: String = String::from("Hello");
  |             -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
4 |         let s2: String = s1;
  |                          -- value moved here
5 |         println!("{}", s1);
  |                        ^^ value borrowed here after move
  |
```

# Rust Ownership Model

## This code won't compile!

```
{
    let s1: String = String::from("Hello");
    let s2: String = s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

Values are "moved" by default: ownership is transferred to s2



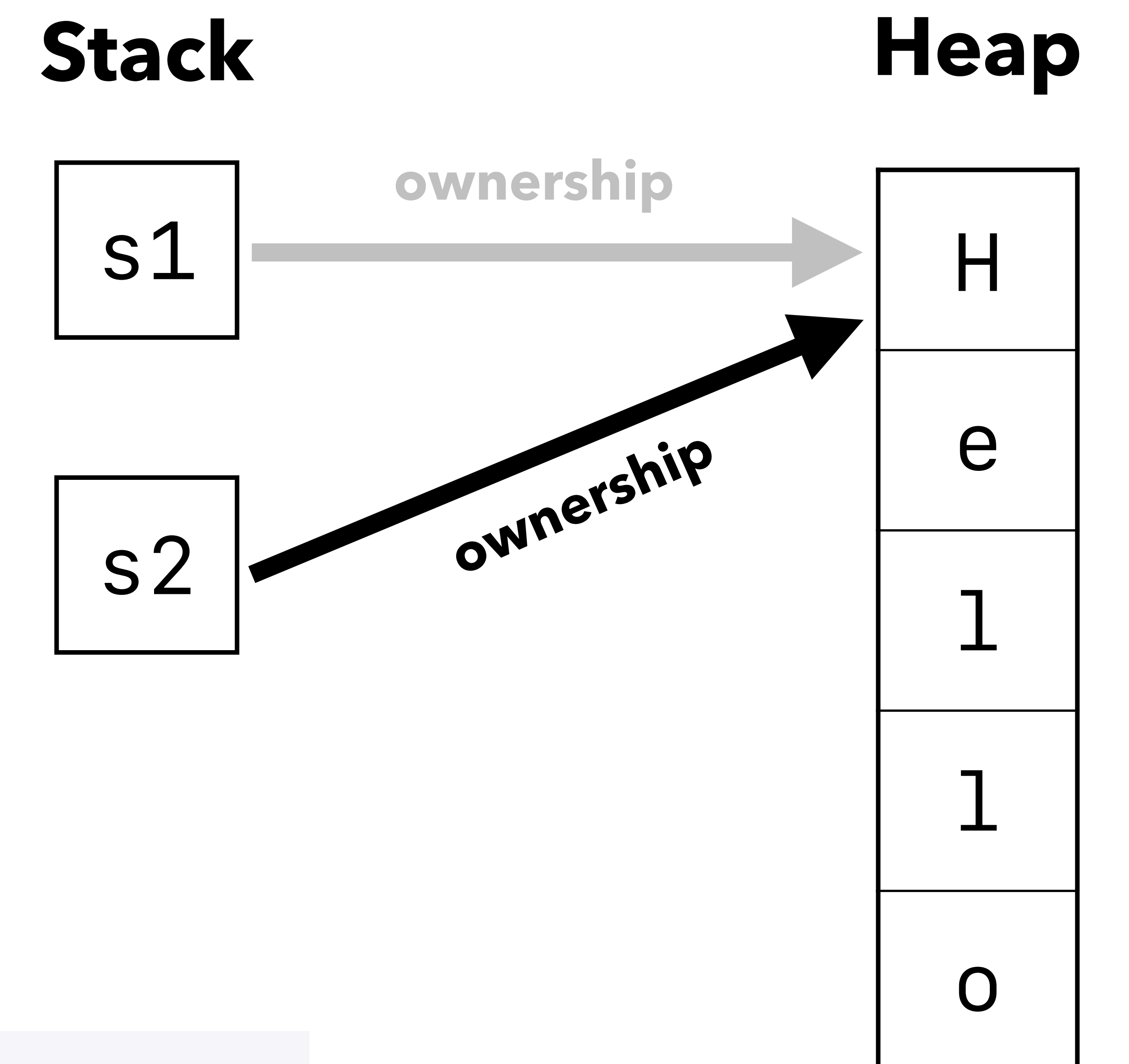

```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:24
  |
3 |         let s1: String = String::from("Hello");
  |             -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
4 |         let s2: String = s1;
  |                          -- value moved here
5 |         println!("{}", s1);
  |                        ^^ value borrowed here after move
  |
```

# Rust Ownership Model

## This code won’t compile!

```
{
    let s1: String = String::from(“Hello");
    let s2: String = s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

Values are “moved” by default: ownership is transferred to s2




```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:24
  |
3 |         let s1: String = String::from("Hello");
  |             -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
4 |         let s2: String = s1;
  |                          -- value moved here
5 |         println!("{}", s1);
  |                        ^^ value borrowed here after move
  |
```

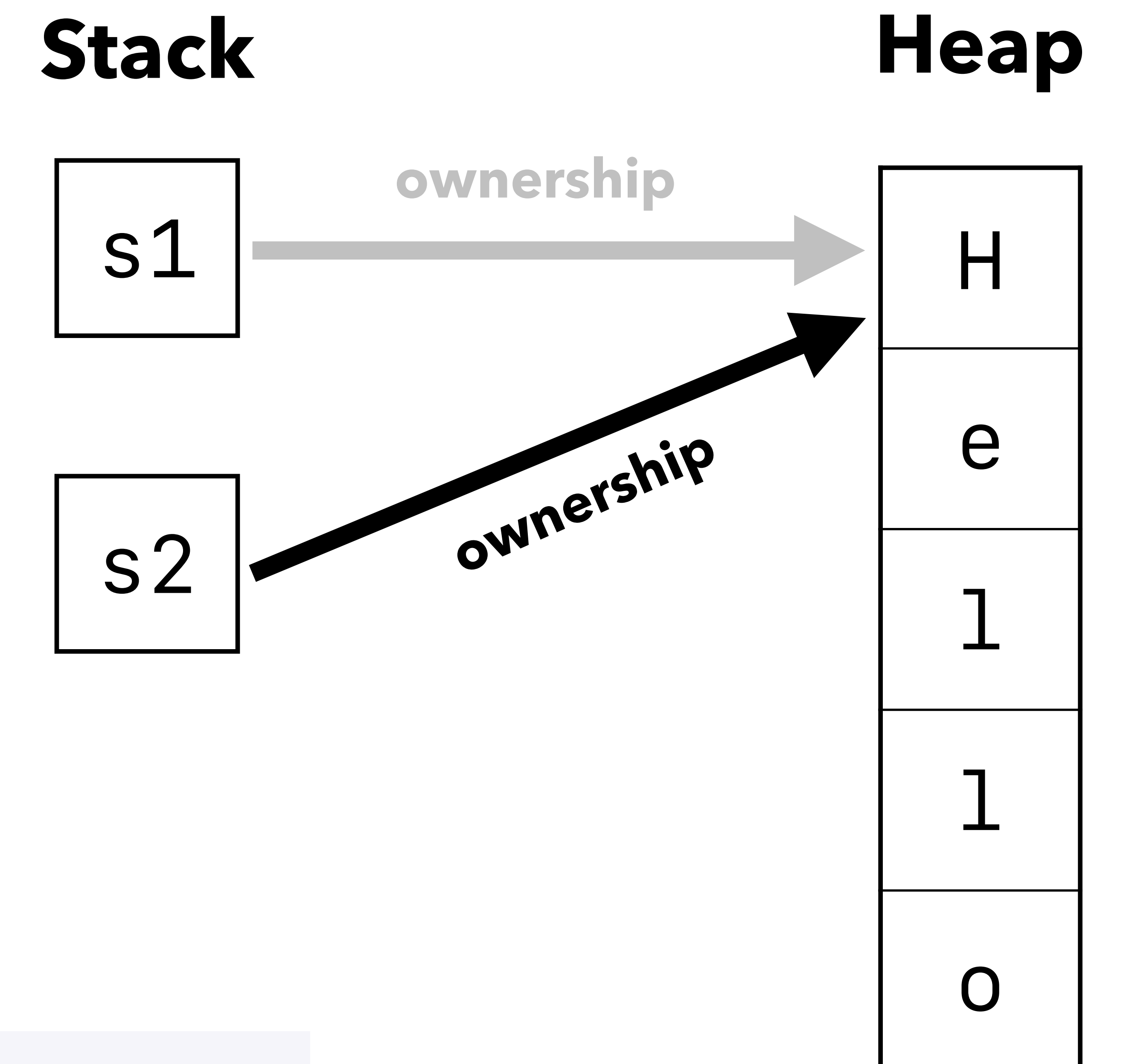
# Rust Ownership Model

## This code won't compile!

```
{
    let s1: String = String::from(“Hello");
    let s2: String = s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

Values are "moved" by default: ownership is transferred to s2

The compiler drops both s1 and s2. Without the ownership restriction, that would be a double free!




```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:24
  |
3 |         let s1: String = String::from("Hello");
  |             -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
4 |         let s2: String = s1;
  |                          -- value moved here
5 |         println!("{}", s1);
  |                        ^^ value borrowed here after move
  |
```

# Rust Ownership Model

## Borrowing

```
{
    let s1: String = String::from("Hello");
    let s2: &String = &s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

**Output:**

```
Hello
Hello
```

# Rust Ownership Model

## Borrowing

```
{
    let s1: String = String::from("Hello");
    let s2: &String = &s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

s2 “borrows” a reference to s1

**Output:**

```
Hello
Hello
```

# Rust Ownership Model

## Borrowing

```
{
    let s1: String = String::from("Hello");
    let s2: &String = &s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

s2 “borrows” a reference to s1

**Stack**

**Heap**

**Output:**

```
Hello
Hello
```

# Rust Ownership Model

## Borrowing

```
{
    let s1: String = String::from("Hello");
    let s2: &String = &s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

s2 “borrows” a reference to s1

**Stack**

**Heap**

| H |
|---|
| e |
| l |
| l |
| o |

**Output:**

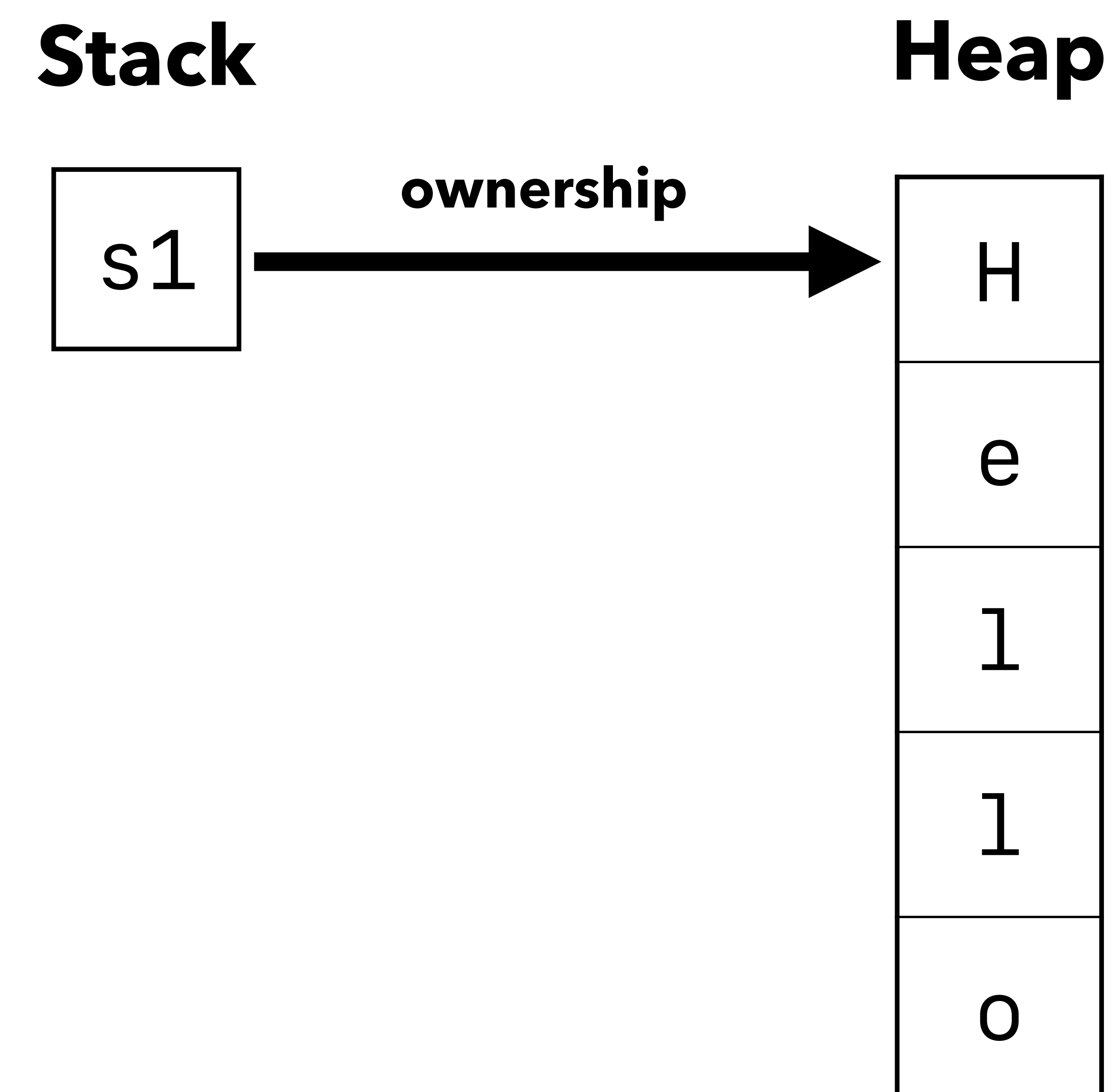
```
Hello
Hello
```

# Rust Ownership Model

## Borrowing

```
{
    let s1: String = String::from("Hello");
    let s2: &String = &s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

s2 "borrows" a reference to s1




**Output:**

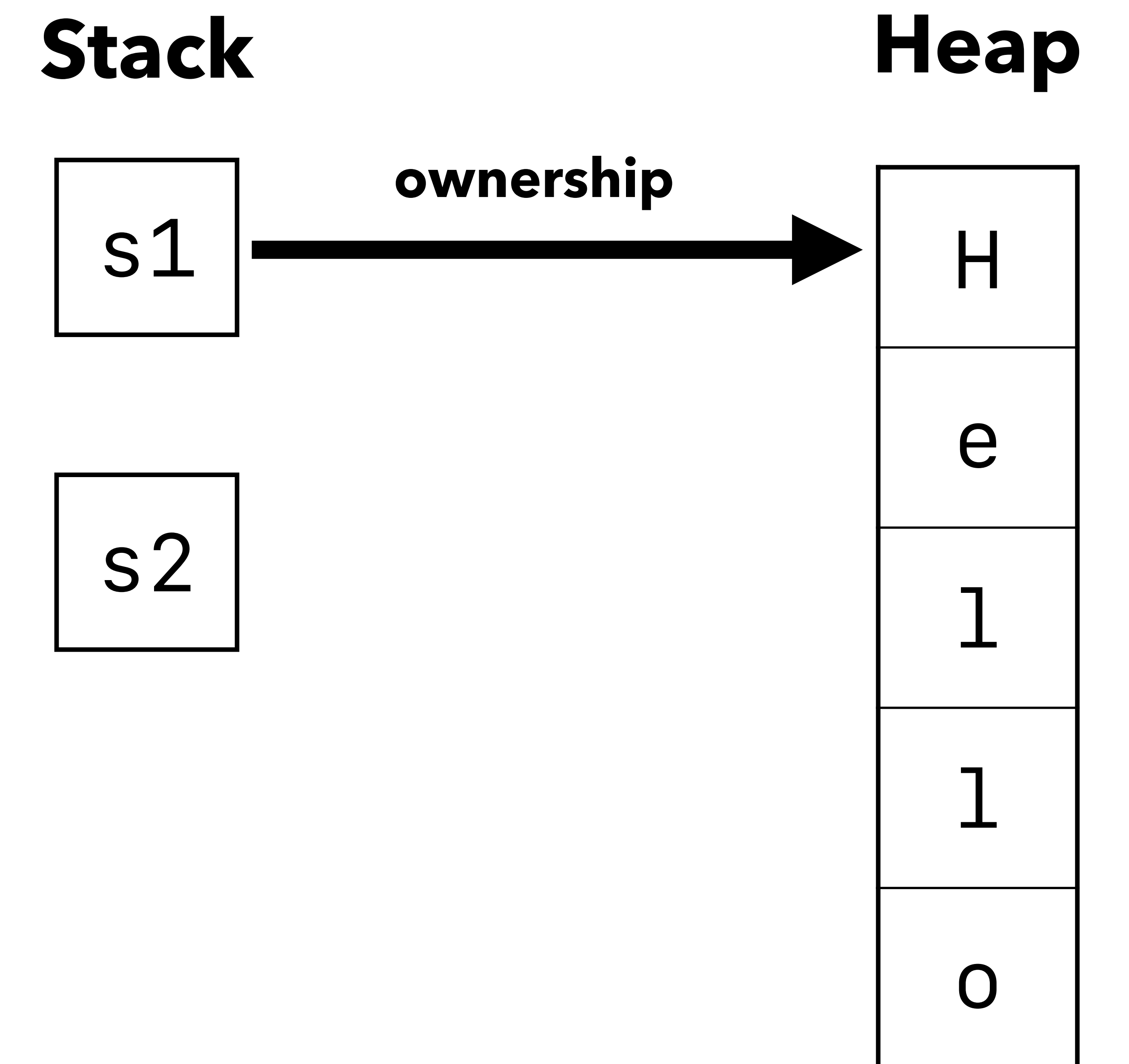
```
Hello
Hello
```

# Rust Ownership Model

## Borrowing

```
{
    let s1: String = String::from("Hello");
    let s2: &String = &s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

s2 “borrows” a reference to s1




**Output:**

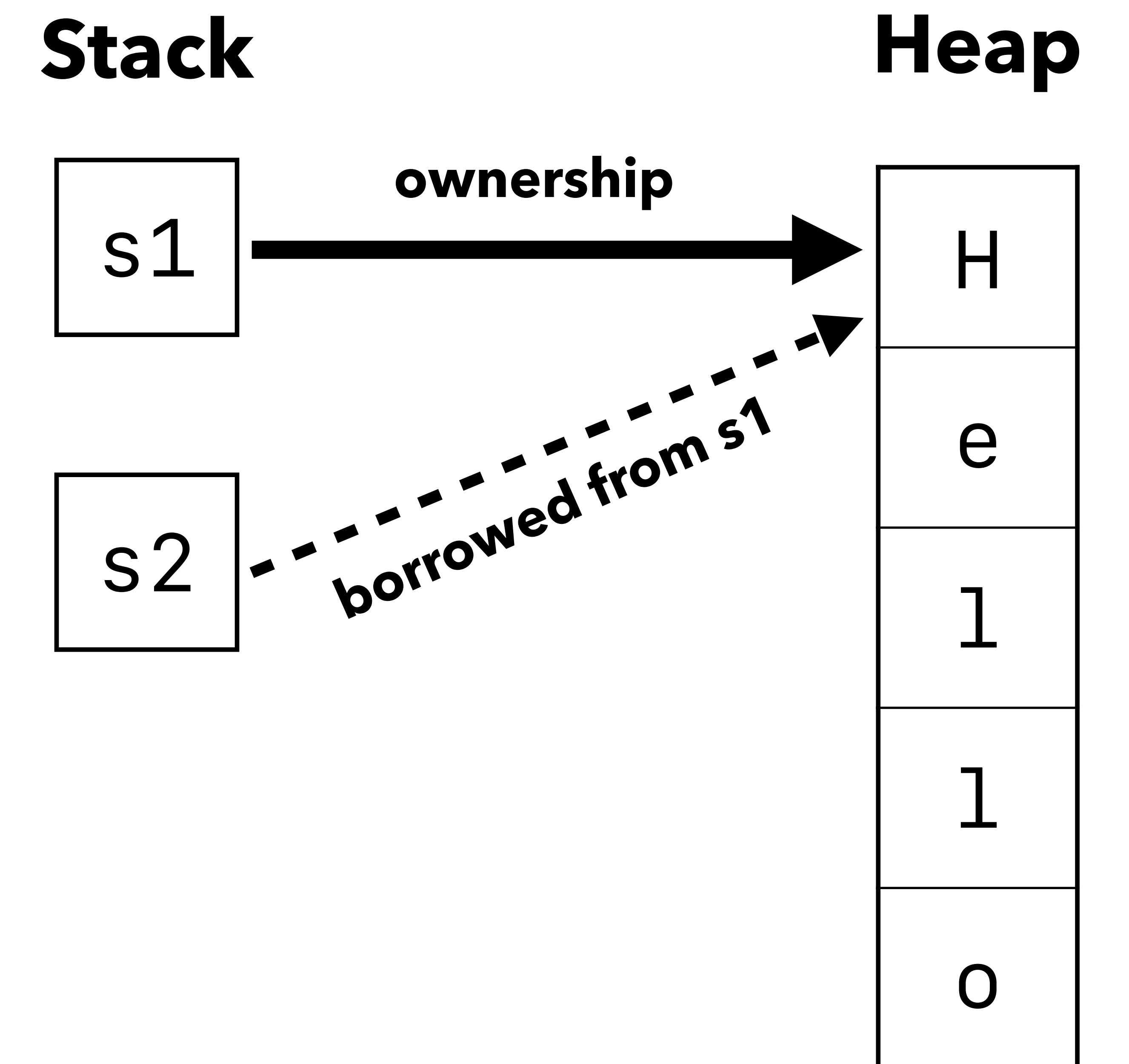
```
Hello
Hello
```

# Rust Ownership Model

## Borrowing

```
{
    let s1: String = String::from("Hello");
    let s2: &String = &s1;
    println!("{}", s1);
    println!("{}", s2);
}
```

s2 “borrows” a reference to s1




**Output:**

```
Hello
Hello
```

# Rust Ownership Model

## This code won't compile!

```
{
    let mut s1: String = String::from("Hello");
    let s2: &String = &s1;
    s1 = String::from(“World");
    println!("{}", s1);
    println!("{}", s2);
}
```

# Rust Ownership Model

## This code won’t compile!

```
{
    let mut s1: String = String::from("Hello");
    let s2: &String = &s1;
    s1 = String::from(“World");
    println!("{}", s1);
    println!("{}", s2);
}
```

```
error[E0506]: cannot assign to `s1` because it is borrowed
 --> src/main.rs:5:9
  |
4 |         let s2: &String = &s1;
  |                           --- borrow of `s1` occurs here
5 |         s1 = String::from("Hello");
  |         ^^ assignment to borrowed `s1` occurs here
6 |         println!("{}", s1);
7 |         println!("{}", s2);
  |                        -- borrow later used here
```

# Rust Ownership Model

## This code won't compile!

Stack

Heap

```
{
    let mut s1: String = String::from("Hello");
    let s2: &String = &s1;
    s1 = String::from("World");
    println!("{}", s1);
    println!("{}", s2);
}
```

```
error[E0506]: cannot assign to `s1` because it is borrowed
 --> src/main.rs:5:9
  |
4 |         let s2: &String = &s1;
  |                           --- borrow of `s1` occurs here
5 |         s1 = String::from("Hello");
  |         ^^ assignment to borrowed `s1` occurs here
6 |         println!("{}", s1);
7 |         println!("{}", s2);
  |                        -- borrow later used here
```

# Rust Ownership Model

## This code won't compile!

Stack

Heap

```
{
    let mut s1: String = String::from("Hello");
    let s2: &String = &s1;
    s1 = String::from("World");
    println!("{}", s1);
    println!("{}", s2);
}
```

```
error[E0506]: cannot assign to `s1` because it is borrowed
 --> src/main.rs:5:9
  |
4 |         let s2: &String = &s1;
  |                           --- borrow of `s1` occurs here
5 |         s1 = String::from("Hello");
  |         ^^ assignment to borrowed `s1` occurs here
6 |         println!("{}", s1);
7 |         println!("{}", s2);
  |                        -- borrow later used here
```

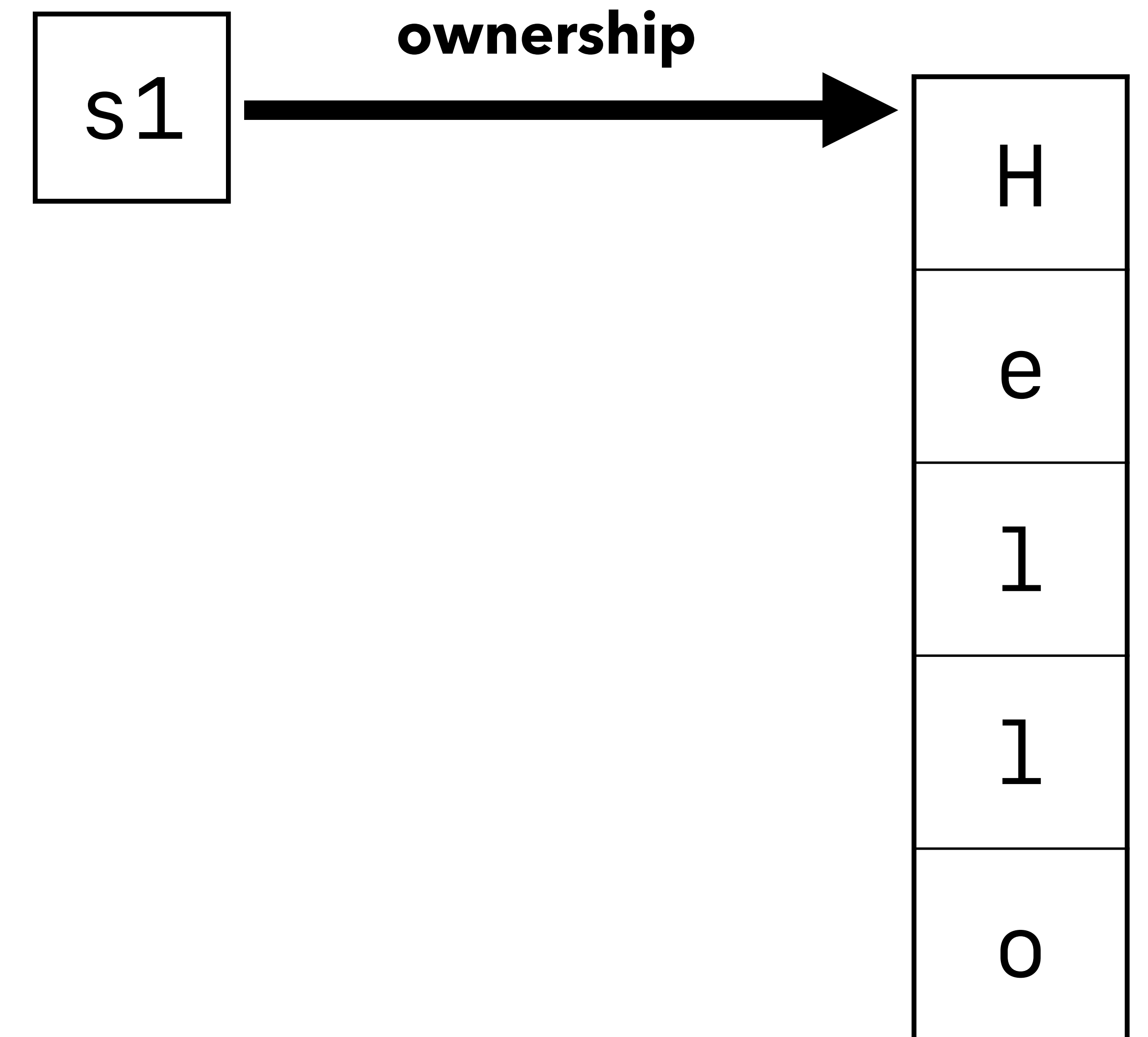
| H |
|---|
| e |
| l |
| l |
| o |

# Rust Ownership Model

## This code won’t compile!

Stack Heap

```
{
    let mut s1: String = String::from("Hello");
    let s2: &String = &s1;
    s1 = String::from(“World");
    println!("{}", s1);
    println!("{}", s2);
}
```




```
error[E0506]: cannot assign to `s1` because it is borrowed
 --> src/main.rs:5:9
  |
4 |         let s2: &String = &s1;
  |                           --- borrow of `s1` occurs here
5 |         s1 = String::from("Hello");
  |         ^^ assignment to borrowed `s1` occurs here
6 |         println!("{}", s1);
7 |         println!("{}", s2);
  |                        -- borrow later used here
```

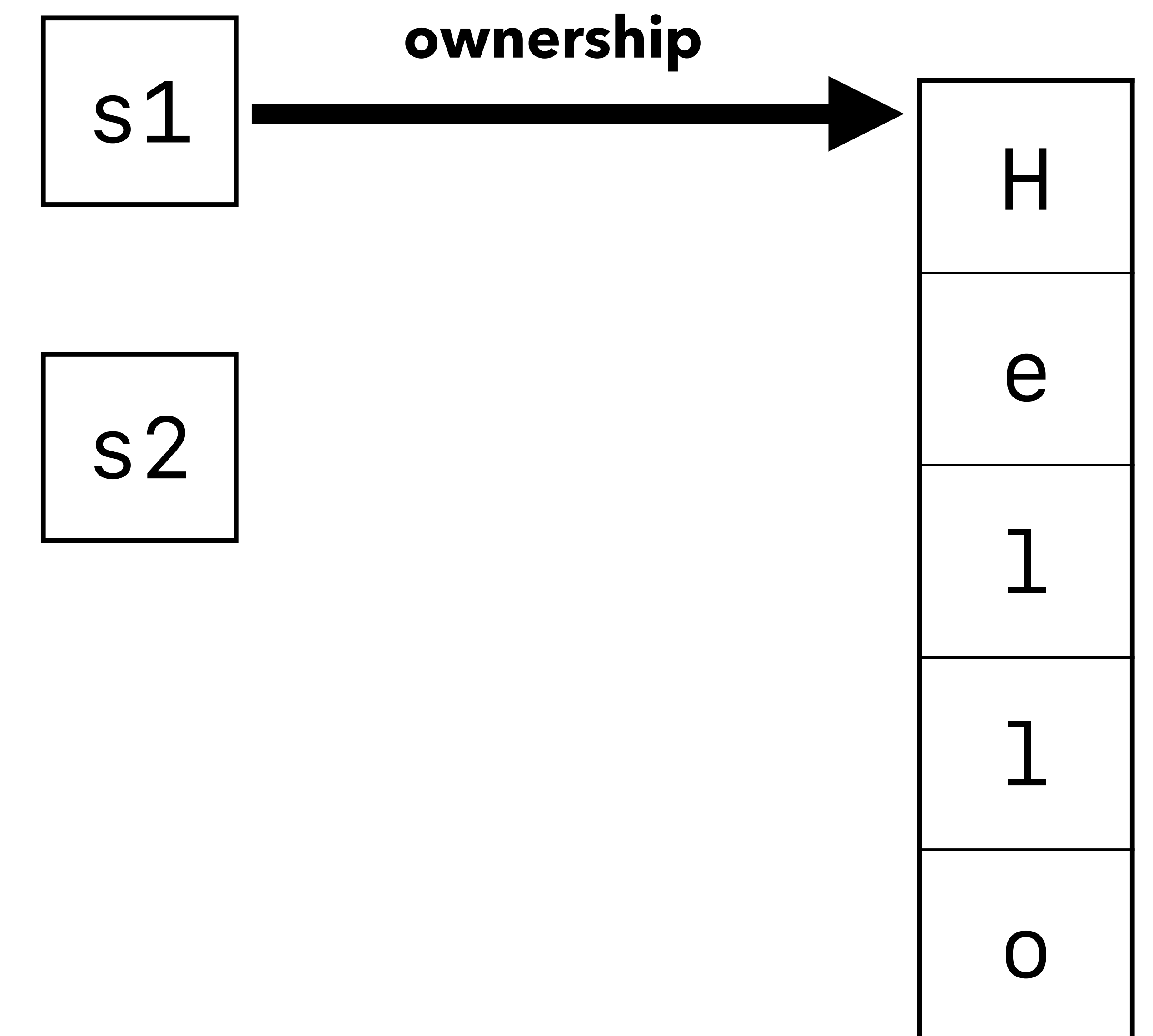
# Rust Ownership Model

## This code won't compile!

```
{
    let mut s1: String = String::from("Hello");
    let s2: &String = &s1;
    s1 = String::from(“World");
    println!("{}", s1);
    println!("{}", s2);
}
```

```
error[E0506]: cannot assign to `s1` because it is borrowed
 --> src/main.rs:5:9
  |
4 |         let s2: &String = &s1;
  |                           --- borrow of `s1` occurs here
5 |         s1 = String::from("Hello");
  |         ^^ assignment to borrowed `s1` occurs here
6 |         println!("{}", s1);
7 |         println!("{}", s2);
  |                        -- borrow later used here
```

**Stack** **Heap**



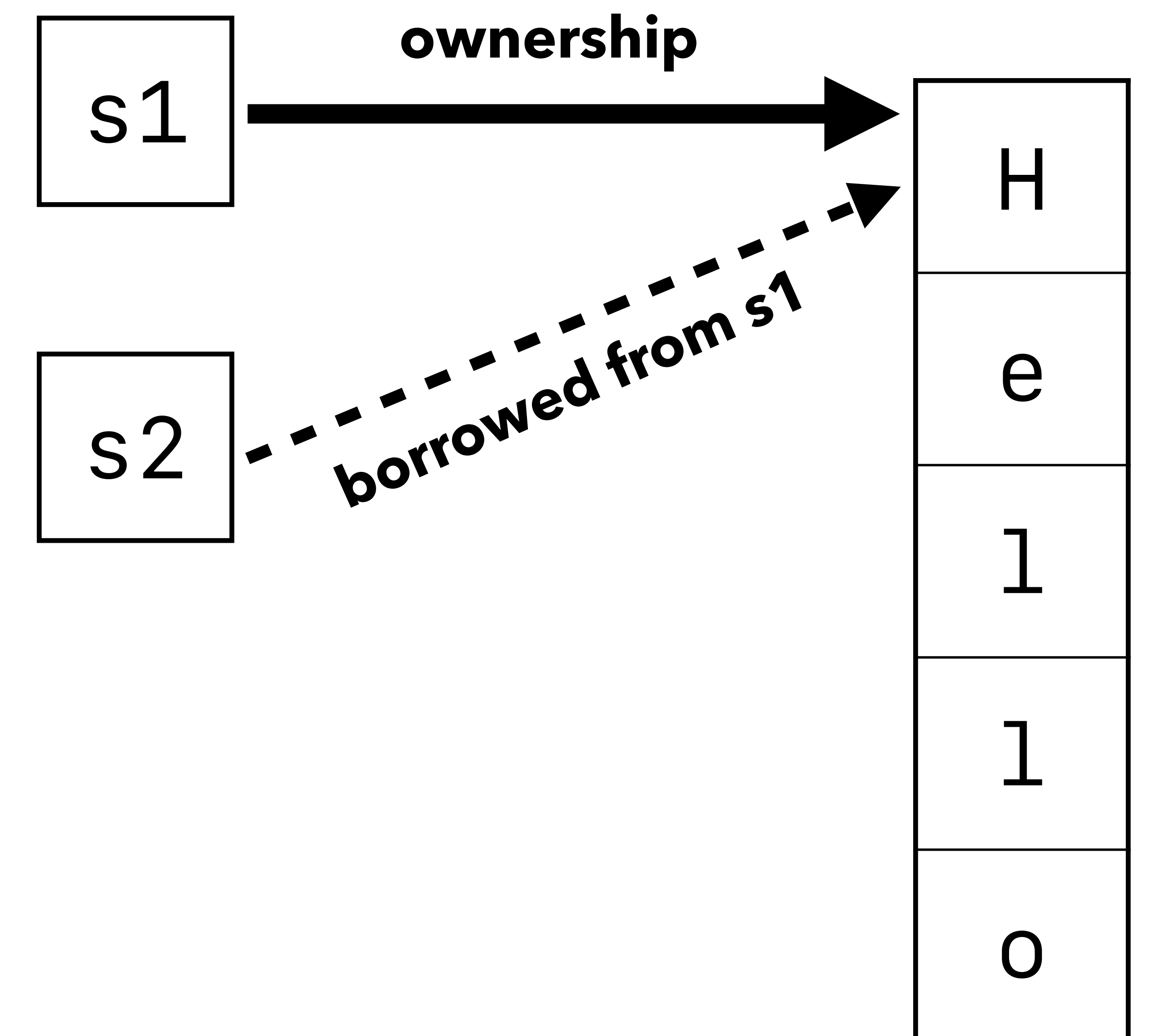
# Rust Ownership Model

## This code won't compile!

```
{
    let mut s1: String = String::from("Hello");
    let s2: &String = &s1;
    s1 = String::from(“World");
    println!("{}", s1);
    println!("{}", s2);
}
```

```
error[E0506]: cannot assign to `s1` because it is borrowed
 --> src/main.rs:5:9
  |
4 |         let s2: &String = &s1;
  |                           --- borrow of `s1` occurs here
5 |         s1 = String::from("Hello");
  |         ^^ assignment to borrowed `s1` occurs here
6 |         println!("{}", s1);
7 |         println!("{}", s2);
  |                        -- borrow later used here
```

**Stack** **Heap**

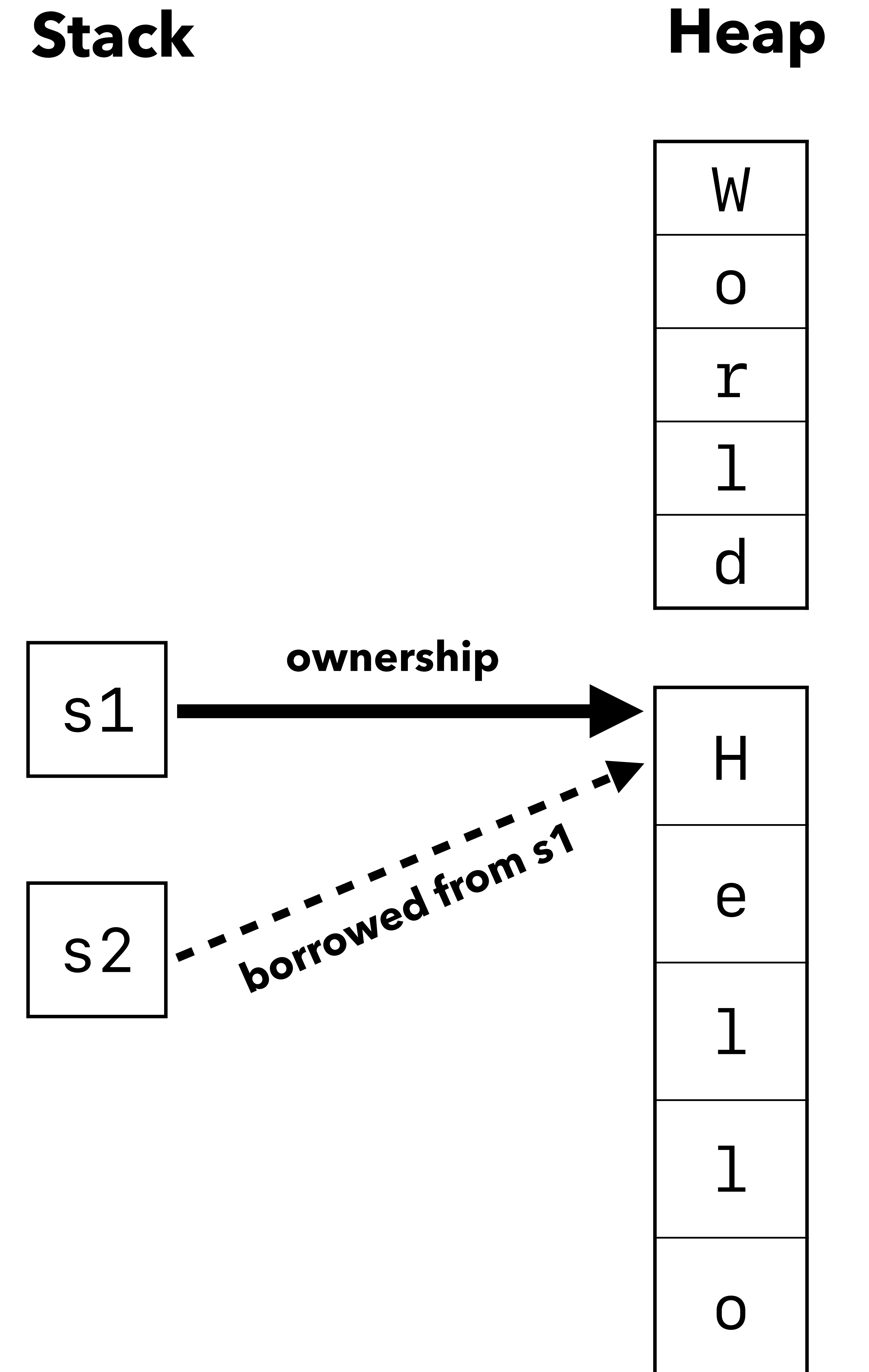


# Rust Ownership Model

## This code won't compile!

```
{
    let mut s1: String = String::from("Hello");
    let s2: &String = &s1;
    s1 = String::from(“World");
    println!("{}", s1);
    println!("{}", s2);
}
```

```
error[E0506]: cannot assign to `s1` because it is borrowed
 --> src/main.rs:5:9
  |
4 |         let s2: &String = &s1;
  |                           --- borrow of `s1` occurs here
5 |         s1 = String::from("Hello");
  |         ^^ assignment to borrowed `s1` occurs here
6 |         println!("{}", s1);
7 |         println!("{}", s2);
  |                        -- borrow later used here
```

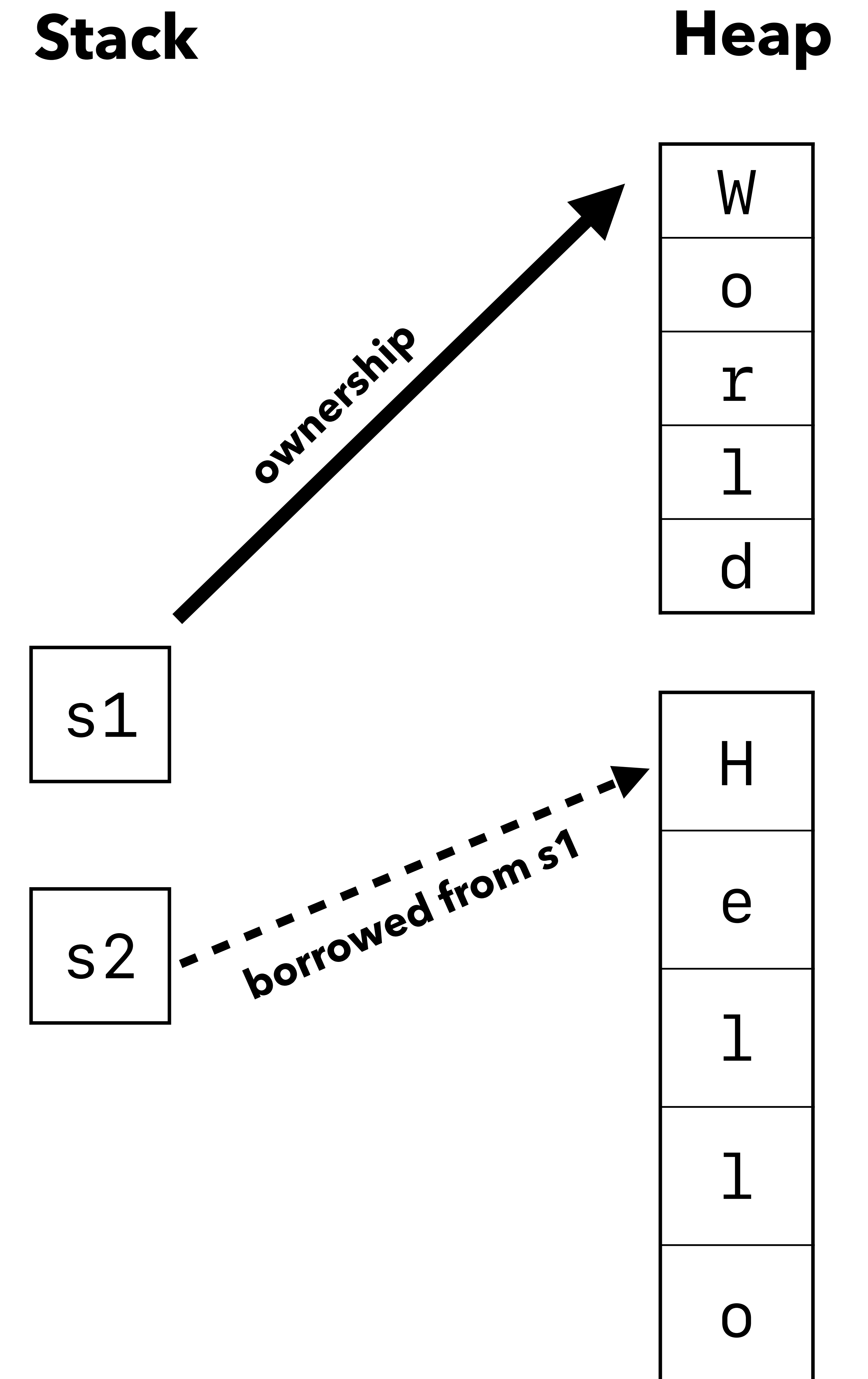


# Rust Ownership Model

## This code won't compile!

```
{
    let mut s1: String = String::from("Hello");
    let s2: &String = &s1;
    s1 = String::from(“World");
    println!("{}", s1);
    println!("{}", s2);
}
```

```
error[E0506]: cannot assign to `s1` because it is borrowed
 --> src/main.rs:5:9
  |
4 |         let s2: &String = &s1;
  |                           --- borrow of `s1` occurs here
5 |         s1 = String::from("Hello");
  |         ^^ assignment to borrowed `s1` occurs here
6 |         println!("{}", s1);
7 |         println!("{}", s2);
  |                        -- borrow later used here
```

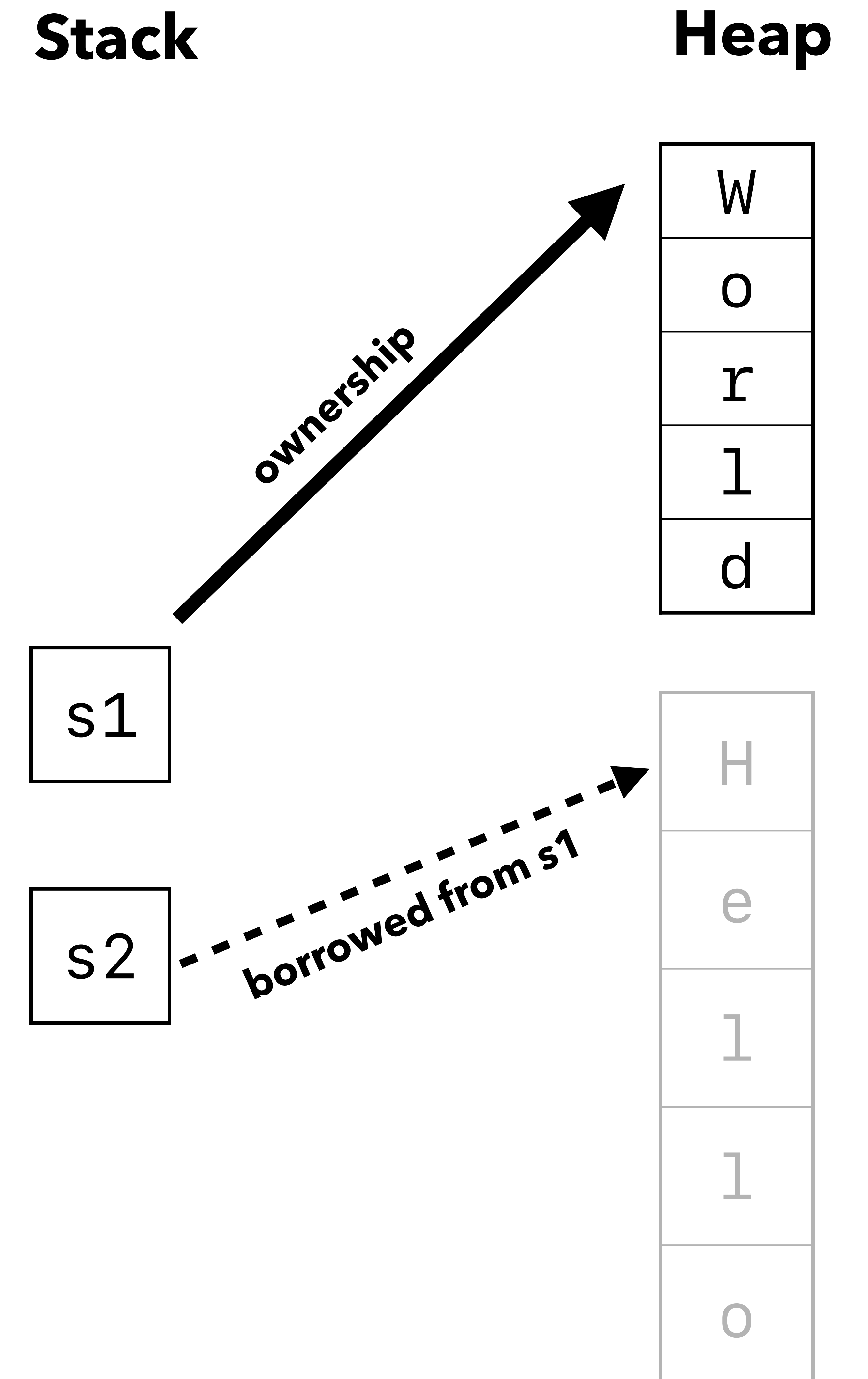


# Rust Ownership Model

## This code won't compile!

```
{
    let mut s1: String = String::from("Hello");
    let s2: &String = &s1;
    s1 = String::from("World");
    println!("{}", s1);
    println!("{}", s2);
}
```

```
error[E0506]: cannot assign to `s1` because it is borrowed
 --> src/main.rs:5:9
  |
4 |         let s2: &String = &s1;
  |                           --- borrow of `s1` occurs here
5 |         s1 = String::from("Hello");
  |         ^^ assignment to borrowed `s1` occurs here
6 |         println!("{}", s1);
7 |         println!("{}", s2);
  |                        -- borrow later used here
```

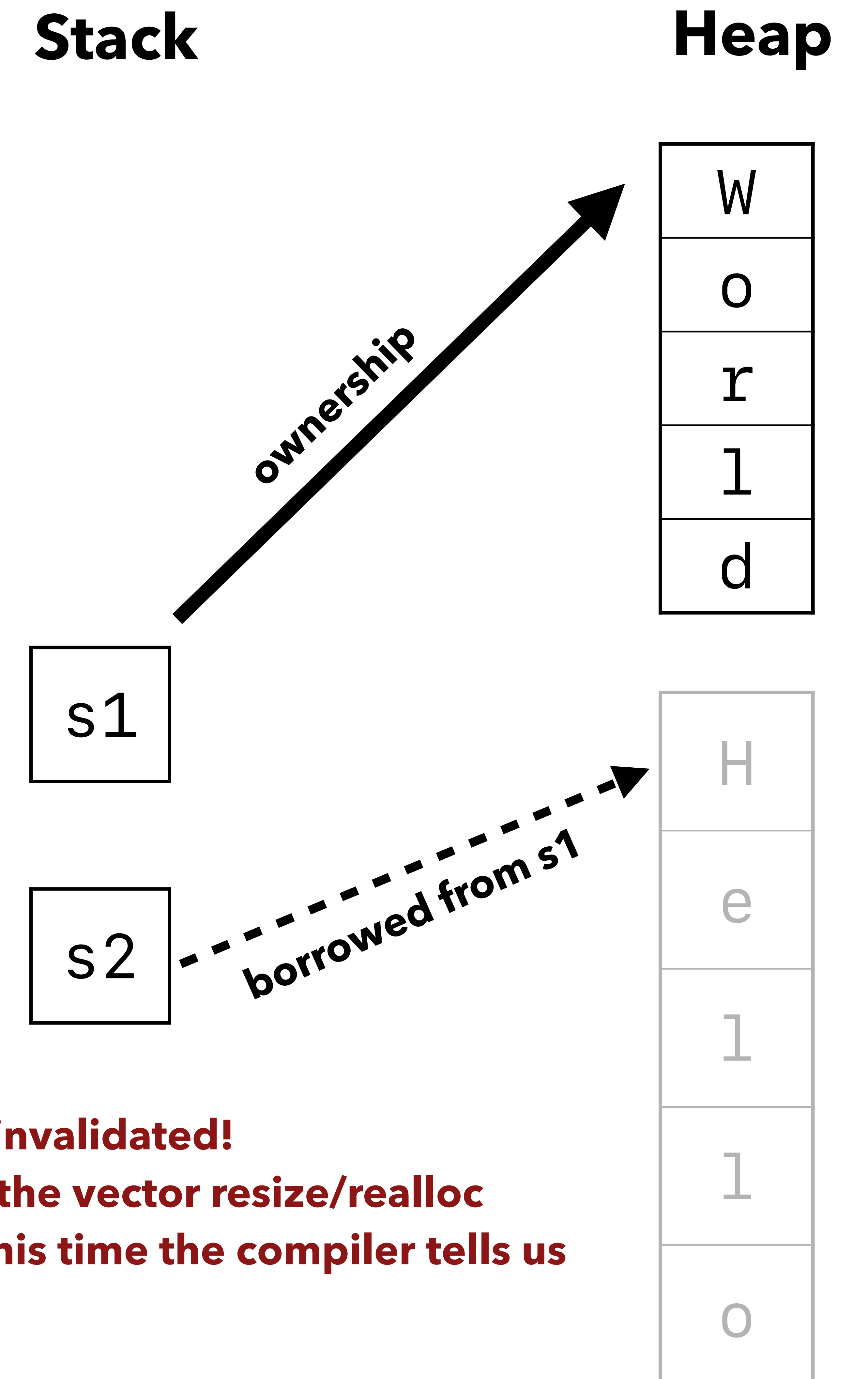


# Rust Ownership Model

## This code won't compile!

```
{
    let mut s1: String = String::from("Hello");
    let s2: &String = &s1;
    s1 = String::from(“World");
    println!("{}", s1);
    println!("{}", s2);
}
```

```
error[E0506]: cannot assign to `s1` because it is borrowed
 --> src/main.rs:5:9
  |
4 |         let s2: &String = &s1;
  |                           --- borrow of `s1` occurs here
5 |         s1 = String::from("Hello");
  |         ^^ assignment to borrowed `s1` occurs here
6 |         println!("{}", s1);
7 |         println!("{}", s2);
  |                        -- borrow later used here
```




s2 is now invalidated!
Similar to the vector resize/realloc bug, but this time the compiler tells us

# Additional Resources

- Interested in security and/or how memory errors can actually be exploited? Take CS155!
- Want to learn more Rust?
  - The Rust book online
  - CS110L used to be offered, but the materials are still available at cs110L.stanford.edu
- Curious about programming languages? CS242 and CS343D