`error[E0382]: borrow of moved value`

Secure Systems Engineering Spring 2024

🛡️ EE G7701

February 13, 2024

Tushar Jois

Recap

- Buffer overflows can be used to hijack the control flow of a program
- Either shellcode or a return-to-libc can spawn a shell afterward
- Defenses against buffer overflows exist, but have shortcomings

Lesson objectives

- Become familiar with basic Rust syntax and types
- Apply Rust languages features when writing code
- Intuitively understand Rust’s borrowing and ownership semantics

Assignment 1 reflection

- How do/did we feel about it?
  - It’s OK if you don’t have as much experience with code
  - But, it’s still a graduate-level course
- Got a lot of emails from folks asking about how to do things
  - Emailing the professor is typically a last resort
- Problem solving is part of the assignment
  - Be specific about what you did and why you did it
  - Always start by closely doing all of the reading
  - Use online resources to help you learn
  - Help each other out
    - Work with your partner -- pair programming
    - Discussion board -- anonymously ask/give help
- Deadline extension -- Assignment 1 now due 10p Feb 14
  - One-time -- people were having some initial trouble with the VM
  - Don’t ask for deadline extensions 6 hours before the due date
    - Not fair to those of your classmates who prioritized effectively

Attribution

This lecture is adapted from material developed for a different course, CIS 198: Rust Programming, Spring 2016, at the University of Pennsylvania.

Rust?

“A language empowering everyone to build reliable and efficient software.”

- rust-lang.org

Rust is:

- Fast
- Safe
- Functional
- Zero-cost

Rust is.. fast

- Rust compiles to native code
- Rust has no garbage collector
- Most abstractions have zero cost
- Fine-grained control over lots of things
- Pay for exactly what you need...
- ...and pay for most of it at compile time

Rust is... safe

- No null
- No uninitialized memory
- No dangling pointers
- No double free errors
- No manual memory management!

Rust is... functional

- First-class functions
- Trait-based generics
- Algebraic datatypes
- Pattern matching

Rust is... zero-cost

- Zero-cost, 100% safe abstractions form Rust's defining feature
- Strict compile-time checks remove need for runtime
- Big concept: Ownership

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

Basic Rust Syntax

Variable Bindings

- Variables are bound with `let`:

```
let x = 17;
```

- Bindings are implicitly-typed: the compiler infers based on context.
- The compiler can't always determine the type of a variable, so sometimes you have to add type annotations.

```
let x: i16 = 17;
```

- Variables are inherently immutable:

```
let x = 5;
x += 1; // error: re-assignment of immutable variable x
let mut y = 5;
y += 1; // OK!
```

Variable Bindings

- Bindings may be shadowed:

```
let x = 17;
let y = 53;
let x = "Shadowed!";
// x is not mutable, but we're able to re-bind it
```

- The shadowed binding for `x` above lasts until it goes out of scope.
- Above, we've effectively lost the first binding, since both `x`s are in the same scope.
- Patterns may also be used to declare variables:

```
let (a, b) = ("foo", 12);
```

Expressions

- (Almost!) everything is an expression: something which returns a value.
  - Exception: variable bindings are not expressions.
- The “nothing” type is called “unit”, which is written `()`.
  - The *type* `()` has only one value: `()`.
  - `()` is the default return type.
- Discard an expression’s value by appending a semicolon. Now it returns `()`.
  - Hence, if a function ends in a semicolon, it returns `()`.

```
fn foo() -> i32 { 5 }
fn bar() -> () { () }
fn baz() -> () { 5; }
fn qux()       { 5; }
```

Expressions

- Because everything is an expression, we can bind many things to variable names:

```
let x = -5;
let y = if x > 0 { "greater" } else { "less" };
println!("x = {} is {} than zero", x, y);
```

- Aside: `"{}"` is Rust's (most basic) string interpolation operator
  - Similar to Python, Ruby, C#, and others; like `printf`'s `"%s"` in C/C++.

Comments

```rust
/// Triple-slash comments are docstring comments.
///
/// `rustdoc` uses docstring comments to generate
/// documentation, and supports **Markdown** formatting.
fn foo() {
    // Double-slash comments are normal.

    /* Block comments
     * also exist /* and can be nested! */
     */
}
```

Types

Primitive Types

- `bool`: spelled `true` and `false`.
- `char`: spelled like `'c'` or `'😺'` (`char`s are Unicode!).
- Numerics: specify the signedness and size.
  - `i8`, `i16`, `i32`, `i64`, `isize`
  - `u8`, `u16`, `u32`, `u64`, `usize`
  - `f32`, `f64`
  - `isize` & `usize` are the size of pointers (and therefore have machine-dependent size)
  - Literals are spelled like `10i8`, `10u16`, `10.0f32`, `10usize`.
  - Type inference for non-specific literals default to `i32` or `f64`:
    - e.g. `10` defaults to `i32`, `10.0` defaults to `f64`.
- Arrays, slices, `str`, tuples.
- Functions.

Arrays

- Arrays are generically of type `[T; N]`.
  - N is a compile-time *constant*. Arrays cannot be resized.
  - Array access is bounds-checked at runtime.
- Arrays are indexed with `[]` like most other languages:
  - `arr[3]` gives you the 4th element of `arr`

```
let arr1 = [1, 2, 3]; // (array of 3 elements)
let arr2 = [2; 32];   // (array of 32 `2`s)
```

Slices

- Generically of type `&[T]`
- A “view” into an array by reference
- Not created directly, but are borrowed from other variables
- Mutable or immutable
- How do you know when a slice is still valid? Coming soon...

```
let arr = [0, 1, 2, 3, 4, 5];
let total_slice = &arr;          // Slice all of `arr`
let total_slice = &arr[..];      // Same, but more explicit
let partial_slice = &arr[2..5]; // [2, 3, 4]
```

Strings

- Two types of Rust strings: `String` and `&str`.
- `String` is a heap-allocated, growable vector of characters.
- `&str` is a type[1] that's used to slice into `String`s.
- String literals like `"foo"` are of type `&str`.

```rust
let s: &str = "galaxy";
let s2: String = "galaxy".to_string();
let s3: String = String::from("galaxy");
let s4: &str = &s3;
```

[1]`str` is an unsized type, which doesn't have a compile-time known size, and therefore cannot exist by itself.

Tuples

- Fixed-size, ordered, heterogeneous lists
- Index into tuples with `foo.0`, `foo.1`, etc.
- Can be destructured in `let` bindings

```
let foo: (i32, char, f64) = (72, 'H', 5.1);
let (x, y, z) = (72, 'H', 5.1);
let (a, b, c) = foo; // a = 72, b = 'H', c = 5.1
```

Casting

- Cast between types with `as`:

```
let x: i32 = 100;
let y: u32 = x as u32;
```

- Naturally, you can only cast between types that are safe to cast between.
    - No casting `[i16; 4]` to `char`! (This is called a “non-scalar” cast)
    - There are unsafe mechanisms to overcome this, if you know what you’re doing.

Vec<T>

- A standard library type: you don't need to import anything.
- A `Vec` (read "vector") is a heap-allocated growable array.
  - (cf. Java's `ArrayList`, C++'s `std::vector`, etc.)
- `<T>` denotes a generic type.
  - The type of a `Vec` of `i32`s is `Vec<i32>`.
- Create `Vec`s with `Vec::new()` or the `vec!` macro.
  - `Vec::new()` is an example of namespacing. `new` is a function defined for the `Vec` struct.

Vec<T>

```rust
// Explicit typing
let v0: Vec<i32> = Vec::new();

// v1 and v2 are equal
let mut v1 = Vec::new();
v1.push(1);
v1.push(2);
v1.push(3);

let v2 = vec![1, 2, 3];
// v3 and v4 are equal
let v3 = vec![0; 4];
let v4 = vec![0, 0, 0, 0];
```

Vec<T>

```rust
let v2 = vec![1, 2, 3];
let x = v2[2]; // 3
```

- Like arrays, vectors can be indexed with `[]`.
  - You can't index a vector with an i32/i64/etc.
  - You must use a `usize` because `usize` is guaranteed to be the same size as a pointer.
  - Other integers can be cast to `usize`:

    ```rust
    let i: i8 = 2;
    let y = v2[i as usize];
    ```
- Vectors have an extensive stdlib method list, which can be found at the official Rust documentation.

References

- Reference *types* are written with an `&`: `&i32`.
- References can be taken with `&` (like C/C++).
- References can be *dereferenced* with `*` (like C/C++).
- References are guaranteed to be valid.
  - Validity is enforced through compile-time checks!
- These are *not* the same as pointers!
- Reference lifetimes are pretty complex, as we'll explore later.

```rust
let x = 12;
let ref_x = &x;
println!("{}", *ref_x); // 12
```

Control Flow

If Statements

```
if x > 0 {
    10
} else if x == 0 {
    0
} else {
    println!("Not greater than zero!");
    -10
}
```

- No parens necessary.
- Entire if statement evaluates to one expression, so every arm must end with an expression of the same type.
    - That type can be unit `()`:

```
if x <= 0 {
    println!("Too small!");
}
```

Loops

- Loops come in three flavors: `while`, `loop`, and `for`.
  - `break` and `continue` exist just like in most languages
- `while` works just like you'd expect:

```rust
let mut x = 0;
while x < 100 {
    x += 1;
    println!("x: {}", x);
}
```

Loops

- `loop` is equivalent to `while true`, a common pattern.
    - Plus, the compiler can make optimizations knowing that it’s infinite.

```
let mut x = 0;
loop {
    x += 1;
    println!("x: {}", x);
}
```

Loops

- `for` is the most different from most C-like languages
  - `for` loops use an *iterator expression*:
  - `n..m` creates an iterator from n to m (exclusive).
  - Some data structures can be used as iterators, like arrays and `Vec`s.

```
// Loops from 0 to 9.
for x in 0..10 {
    println!("{}", x);
}

let xs = [0, 1, 2, 3, 4];
// Loop through elements in a slice of `xs`.
for x in &xs {
    println!("{}", x);
}
```

Functions

```rust
fn foo(x: T, y: U, z: V) -> T {
    // ...
}
```

- `foo` is a function that takes three parameters:
  - `x` of type `T`
  - `y` of type `U`
  - `z` of type `V`
- `foo` returns a `T`.
- Must explicitly define argument and return types.
  - The compiler is actually smart enough to figure this out for you, but Rust’s designers decided it was better practice to force explicit function typing.

Functions

- The final expression in a function is its return value.
  - Use `return` for *early* returns from a function.

```
fn square(n: i32) -> i32 {
    n * n
}

fn squareish(n: i32) -> i32 {
    if n < 5 { return n; }
    n * n
}

fn square_bad(n: i32) -> i32 {
    n * n;
}
```

- The last one won't even compile!
  - Why? It ends in a semicolon, so it evaluates to `()`.

Function Objects

- Several things can be used as function objects:
  - Function pointers (a reference to a normal function)
  - Closures (more advanced)
- Much more straightforward than C function pointers:

```rust
let x: fn(i32) -> i32 = square;
```

- Can be passed by reference:

```rust
fn apply_twice(f: &Fn(i32) -> i32, x: i32) -> i32 {
    f(f(x))
}

// ...

let y = apply_twice(&square, 5);
```

Macros!

- Macros are like functions, but they're named with `!` at the end.
- Can do generally very powerful stuff.
  - They actually generate code at compile time!
- Call and use macros like functions.
- You can define your own with `macro_rules! macro_name` blocks.
  - These are *very* complicated.
- Because they're so powerful, a lot of common utilities are defined as macros.

print! & println!

- Print stuff out. Yay.
- Use `{}` for general string interpolation, and `{:?}` for debug printing.
  - Some types can only be printed with `{:?}`, like arrays and `Vec`s.

```
print!("{}, {}, {}", "foo", 3, true);
// => foo, 3, true
println!("{:?}, {:?}", "foo", [1, 2, 3]);
// => "foo", [1, 2, 3]
```

format!

- Uses `println!`-style string interpolation to create formatted `String`s.

```
let fmted = format!("{}, {:x}, {:?}", 12, 155,
Some("Hello"));
// fmted == "12, 9b, Some("Hello")"
```

panic!(msg)

- Exits current task with given message.
- Don't do this lightly! It is better to handle and report errors explicitly.

```
if x < 0 {
    panic!("Oh noes!");
}
```

assert! & assert_eq!

- `assert!(condition)` panics if `condition` is `false`.
- `assert_eq!(left, right)` panics if `left != right`.
- Useful for testing and catching illegal conditions.

```
#[test]
fn test_something() {
    let actual = 1 + 2;
    assert!(actual == 3);
    assert_eq!(3, actual);
}
```

unreachable!()

- Used to indicate that some code should not be reached.
- `panic!`s when reached.
- Can be useful to track down unexpected bugs (e.g. optimization bugs).

```
if false {
    unreachable!();
}
```

unimplemented!()

- Shorthand for `panic!("not yet implemented")`
- You'll probably see this in your assignments a lot!

```
fn sum(x: Vec<i32>) -> i32 {
    // TODO
    unimplemented!();
}
```

Match statements

```
let x = 3;

match x {
    1 => println!("one fish"),  // <- comma required
    2 => {
        println!("two fish");
        println!("two fish");
    },  // <- comma optional when using braces
    _ => println!("no fish for you"), // "otherwise" case
}
```

- `match` takes an expression (`x`) and branches on a list of `value => expression` statements.
- The entire match evaluates to one expression.
    - Like `if`, all arms must evaluate to the same type.
- `_` is commonly used as a catch-all (cf. Haskell, OCaml).

Match statements

```rust
let x = 3;
let y = -3;

match (x, y) {
    (1, 1) => println!("one"),
    (2, j) => println!("two, {}", j),
    (_, 3) => println!("three"),
    (i, j) if i > 5 && j < 0 => println!("On guard!"),
    (_, _) => println!(":<"),
}
```

- The matched expression can be any expression (l-value), including tuples and function calls.
    - Matches can bind variables. _ is a throw-away variable name.
- You *must* write an exhaustive match in order to compile.
- Use `if`-guards to constrain a match to certain conditions.
- Patterns can get very complex, as we'll see later.

Tooling

Cargo

- Rust’s package manager & build tool
- Create a new project:
  - `cargo new project_name` (library)
  - `cargo new project_name --bin` (executable)
- Build your project: `cargo build`
- Run your tests: `cargo test`
- Magic, right? How does this work?

Cargo.toml

- Cargo uses the `Cargo.toml` file to declare and manage dependencies and project metadata.
  - TOML is a simple format similar to INI.

```toml
[package]
name = "Rust"
version = "0.1.0"
authors = ["Ferris <ferris@rust-lang.org>"]

[dependencies]
uuid = "0.1"
rand = "0.3"

[profile.release]
opt-level = 3
debug = false
```

cargo test

- A test is any function annotated with `#[test]`.
- `cargo test` will run all annotated functions in your project.
- Any function which executes without crashing (`panic!`ing) succeeds.
- Use `assert!` (or `assert_eq!`) to check conditions (and `panic!` on failure)

```
#[test]
fn it_works() {
    // ...
}
```

Activity interlude

Ownership & Borrowing

Ownership & Borrowing

- Explicit ownership is the biggest new feature that Rust brings to the table!
- Ownership is all[1] checked at compile time!
- Newcomers to Rust often find themselves “fighting with the borrow checker” trying to get their code to compile

[1]*mostly*

Ownership

- A variable binding *takes ownership* of its data.
  - A piece of data can only have one owner at a time.
- When a binding goes out of scope, the bound data is released automatically.
  - For heap-allocated data, this means de-allocation.
- Data *must be guaranteed* to outlive its references.

```rust
fn foo() {
    // Creates a Vec object.
    // Gives ownership of the Vec object to v1.
    let mut v1 = vec![1, 2, 3];

    v1.pop();
    v1.push(4);

    // At the end of the scope, v1 goes out of scope.
    // v1 still owns the Vec object, so it can be cleaned up.
}
```

Move Semantics

```rust
let v1 = vec![1, 2, 3];

// Ownership of the Vec object moves to v2.
let v2 = v1;

println!("{}", v1[2]); // error: use of moved value `v1`
```

- `let v2 = v1;`
  - We don’t want to copy the data, since that’s expensive.
  - The data cannot have multiple owners.
  - Solution: move the Vec’s ownership into `v2`, and declare `v1` invalid.
- `println!("{}", v1[2]);`
  - We know that `v1` is no longer a valid variable binding, so this is an error.
- Rust can reason about this at compile time, so it throws a compiler error.

Move Semantics

- Moving ownership is a compile-time semantic; it doesn’t involve moving data during your program.
- Moves are automatic (via assignments); no need to use something like C++’s `std::move`.
  - However, there are functions like `std::mem::replace` in Rust to provide advanced ownership management.

Ownership

- Ownership does not always have to be moved.
- What would happen if it did? Rust would get very tedious to write:

```rust
fn vector_length(v: Vec<i32>) -> Vec<i32> {
    // Do whatever here,
    // then return ownership of `v` back to the caller
}
```

- You could imagine that this does not scale well either.
  - The more variables you had to hand back, the longer your return type would be!
  - Imagine having to pass ownership around for 5+ variables at a time :(

Borrowing

- Obviously, this is not the case.
- Instead of transferring ownership, we can *borrow* data.
- A variable's data can be borrowed by taking a reference to the variable; ownership doesn't change.
  - When a reference goes out of scope, the borrow is over.
  - The original variable retains ownership throughout.

```
let v = vec![1, 2, 3];

// v_ref is a reference to v.
let v_ref = &v;

// use v_ref to access the data in the vector v.
assert_eq!(v[1], v_ref[1]);
```

Borrowing

- Caveat: this adds restrictions to the original variable.
- Ownership cannot be transferred from a variable while references to it exist.
  - That would invalidate the reference.

```rust
let v = vec![1, 2, 3];

// v_ref is a reference to v.
let v_ref = &v;

// Moving ownership to v_new would invalidate v_ref.
// error: cannot move out of `v` because it is borrowed
let v_new = v;
```

Borrowing

```rust
/// `length` only needs `vector` temporarily, so it is borrowed.
fn length(vec_ref: &Vec<i32>) -> usize {
    // vec_ref is auto-dereferenced when you call methods on it.
    vec_ref.len()
    // you can also explicitly dereference.
    // (*vec_ref).len()
}

fn main() {
    let vector = vec![];
    length(&vector);
    println!("{:?}", vector); // this is fine
}
```

- Note the type of `length`: `vec_ref` is passed by reference, so it's now an `&Vec<i32>`.
- References, like bindings, are *immutable* by default.
- The borrow is over after the reference goes out of scope (at the end of `length`).

Borrowing

```rust
/// `push` needs to modify `vector` so it is borrowed mutably.
fn push(vec_ref: &mut Vec<i32>, x: i32) {
    vec_ref.push(x);
}

fn main() {
    let mut vector: Vec<i32> = vec![];
    let vector_ref: &mut Vec<i32> = &mut vector;
    push(vector_ref, 4);
}
```

- Variables can be borrowed by *mutable* reference: `&mut vec_ref`.
  - `vec_ref` is a reference to a mutable `Vec`.
  - The type is `&mut Vec<i32>`, not `&Vec<i32>`.
- Different from a reference which is variable.

Borrowing

```rust
/// `push` needs to modify `vector` so it is borrowed mutably.
fn push2(vec_ref: &mut Vec<i32>, x: i32) {
    // error: cannot move out of borrowed content.
    let vector = *vec_ref;
    vector.push(x);
}

fn main() {
    let mut vector = vec![];
    push2(&mut vector, 4);
}
```

- Error! You can't dereference `vec_ref` into a variable binding because that would change the ownership of the data.

Borrowing

- Rust will auto-dereference variables...
  - When making method calls on a reference.
  - When passing a reference as a function argument.

```rust
/// `length` only needs `vector` temporarily, so it is borrowed.
fn length(vec_ref: &&Vec<i32>) -> usize {
    // vec_ref is auto-dereferenced when you call methods on it.
    vec_ref.len()
}

fn main() {
    let vector = vec![];
    length(&&&&&&&&&&&&vector);
}
```

Borrowing

- You will have to dereference variables...
  - When writing into them.
  - And other times that usage may be ambiguous.

```
let mut a = 5;
let ref_a = &mut a;
*ref_a = 4;
println!("{}", *ref_a + 4);
// ==> 8
```

`Copy` Types

- Rust defines a trait[1] named `Copy` that signifies that a type may be copied instead whenever it would be moved.
- Most primitive types are `Copy` (`i32`, `f64`, `char`, `bool`, etc.)
- Types that contain references may not be `Copy` (e.g. `Vec`, `String`).

```rust
let x: i32 = 12;
let y = x; // `i32` is `Copy`, so it's not moved :D
println!("x still works: {}, and so does y: {}", x, y);
```

[1] Like a Java interface or Haskell typeclass

Borrowing Rules

The Holy Grail of Rust

Learn these rules, and they will serve you well.

You can't keep borrowing something after it stops existing.

One object may have many immutable references to it (`&T`).

OR *exactly one* mutable reference (`&mut T`) (not both).

That's it!

Borrowing Prevents...

- Iterator invalidation due to mutating a collection you're iterating over.
- This pattern can be written in C, C++, Java, Python, Javascript...
  - But may result in, e.g, `ConcurrentModificationException` (at runtime!)

```
let mut vs = vec![1,2,3,4];
for v in &vs {
    vs.pop();
    // ERROR: cannot borrow `vs` as mutable because
    // it is also borrowed as immutable
}
```

- `pop` needs to borrow `vs` as mutable in order to modify the data.
- But `vs` is being borrowed as immutable by the loop!

Borrowing Prevents...

- Use-after-free
- Valid in C, C++...

```rust
let y: &i32;
{
    let x = 5;
    y = &x; // error: `x` does not live long enough
}
println!("{}", *y);
```

- The full error message:

```
error: `x` does not live long enough
note: reference must be valid for the block suffix following statement
    0 at 1:16
...but borrowed value is only valid for the block suffix
    following statement 0 at 4:18
```

- This eliminates a *huge* number of memory safety bugs *at compile time*.

Example: Vectors

- You can iterate over `Vec`s in three different ways:

```rust
let mut vs = vec![0,1,2,3,4,5,6];

// Borrow immutably
for v in &vs { // Can also write `for v in vs.iter()`
    println!("I'm borrowing {}.", v);
}
// Borrow mutably
for v in &mut vs { // Can also write `for v in vs.iter_mut()`
    *v = *v + 1;
    println!("I'm mutably borrowing {}.", v);
}
// Take ownership of the whole vector
for v in vs { // Can also write `for v in vs.into_iter()`
    println!("I now own {}! AHAHAHAHA!", v);
}
// `vs` is no longer valid
```

Looking ahead

- Assignment 1 now due 10p Feb 14
- Assignment 2 will be out right after
    - Read it before coming to class
    - It’s a longer assignment, but you have more time to do it
    - Use the discussion board for this assignment
- Make sure you submit Lab 1 to Blackboard
- Read the Rust book chapters for next time
    - Follow along by typing code into your Rust environment

Lesson objectives

- Become familiar with basic Rust syntax and types
- Apply Rust languages features when writing code
- Intuitively understand Rust’s borrowing and ownership semantics