Bounds Checking Deep Dive: Code Companion

Reference code for the Bounds Checking Deep Dive lecture. Sections correspond to the lecture document.

Section 1: The Vulnerability Rust Eliminates

```c
#include <string.h>

void vulnerable_copy(char* dest, const char* src) {
    strcpy(dest, src);  // NO BOUNDS CHECKING!
    // strcpy copies until null terminator, ignoring dest size
}

int main() {
    char buffer[10];  // Only 10 bytes allocated on stack
    char* long_string = "This is a very long string that will overflow";
    // This string is 46 characters - 36 bytes overflow!

    vulnerable_copy(buffer, long_string);  // BUFFER OVERFLOW!
    // Those extra 36 bytes overwrite the stack frame,
    // potentially corrupting the return address
    return 0;
}
```

This C code represents the class of vulnerability that Rust eliminates by design. The `strcpy` function has no way to know the destination buffer's size—it's just a raw pointer with no length information.

Section 2: Rust's Safe Copy Pattern

```rust
/// Rust's safe equivalent to strcpy
/// Note: slices are "fat pointers" containing (ptr, length)
pub fn safe_copy(dest: &mut [u8], src: &[u8]) -> Result<(), &'static str> {
    // Length information travels WITH the slice - can't be forgotten
    if dest.len() < src.len() {
        return Err("Destination buffer too small");
    }

    // dest[..src.len()] creates a sub-slice of exact size needed
    // copy_from_slice performs its own bounds verification
    dest[..src.len()].copy_from_slice(src);
    Ok(())  // Result type forces caller to handle potential failure
}
```

The signature `&mut [u8]` and `&[u8]` are slices—fat pointers that carry their length. The `Result` return type makes error handling explicit and non-ignorable.

Section 3: The Two Faces of Indexed Access

```rust
pub fn safe_array_access() {
    let array = [1, 2, 3, 4, 5];

    // Direct indexing: bounds-checked at runtime, panics if invalid
    let _first = array[0];  // OK - returns i32 directly

    // This would panic with a clear error message:
    // let _invalid = array[10];  // thread 'main' panicked at 'index out of
bounds'

    // Safe alternative: get() returns Option<&T>
    match array.get(10) {
        Some(value) => println!("Value: {}", value),  // Never reached here
        None => println!("Index out of bounds (safely handled)"),
    }
}
```

Access Method	Return Type	Out-of-Bounds Behavior	Use When
`array[i]`	`T`	Panics immediately	Bug if index invalid
`array.get(i)`	`Option<&T>`	Returns `None`	Invalid index is expected

Section 4: Vectors and Dynamic Safety

```rust
pub fn safe_vector_usage() {
    let mut vec = Vec::new();  // Dynamically-sized, heap-allocated
    vec.push(1);
    vec.push(2);
    vec.push(3);

    // Iterator-based access: bounds checking happens once at creation
    // Cannot iterate past the end - iterator knows exact length
    for item in &vec {
        println!("{}", item);
    }

    // get() works identically to arrays - same Option<&T> pattern
    if let Some(value) = vec.get(5) {
        println!("Value at index 5: {}", value);
    } else {
        println!("Index 5 doesn't exist");  // This branch executes
    }
}
```

The iterator `&vec` yields exactly `vec.len()` references. The `if let` syntax is a concise way to handle the `Some` case when you don't need the `None` branch to do anything special.

Section 5: Strings as Growable Buffers

```rust
pub fn safe_string_operations() {
    // Initial capacity is just a hint - not a hard limit
    let mut dest = String::with_capacity(10);
    let src = "This is a very long string that would overflow a fixed buffer";

    // In C: catastrophic overflow. In Rust: automatic reallocation
    dest.push_str(src);  // String grows to accommodate 61 characters

    println!("String length: {} (automatically managed)", dest.len());
    // Prints: String length: 61 (automatically managed)
}
```

`String::with_capacity(10)` pre-allocates 10 bytes but imposes no limit. When `push_str` needs more space, `String` automatically reallocates to a larger buffer.

Section 6: Compile-Time Size Enforcement

```
pub fn demonstration_bounds_checking() {
    let buffer: [u8; 10] = [0; 10];  // Fixed-size array: 10 bytes
    let data: [u8; 20] = [1; 20];    // Fixed-size array: 20 bytes

    // This would NOT compile - size mismatch caught at compile time:
    // buffer.copy_from_slice(&data);
    // error: source slice length (20) does not match destination slice length
(10)

    // Safe alternative: explicitly slice to matching size
    let safe_copy = &data[..buffer.len()];  // Take first 10 bytes only
    let mut mutable_buffer = buffer;
    mutable_buffer.copy_from_slice(safe_copy);  // Now sizes match

    println!("Safe copy completed: {:?}", mutable_buffer);
    // Prints: Safe copy completed: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
}
```

When array sizes are known at compile time, the compiler catches mismatches before the code ever runs. The slice expression `&data[..buffer.len()]` creates a view of exactly 10 elements.

Section 7: Testing Bounds Behavior

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_safe_copy_success() {
        let mut dest = [0u8; 10];
        let src = [1, 2, 3, 4, 5];

        assert!(safe_copy(&mut dest, &src).is_ok());
        assert_eq!(&dest[..5], &src);  // First 5 bytes copied
    }

    #[test]
    fn test_safe_copy_overflow_prevented() {
        let mut dest = [0u8; 5];
        let src = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

        // Overflow attempt returns Err, buffer unchanged
        assert!(safe_copy(&mut dest, &src).is_err());
    }

    #[test]
    #[should_panic]  // Test expects a panic - controlled failure
    #[allow(unconditional_panic, clippy::out_of_bounds_indexing)]
    fn test_out_of_bounds_panic() {
        let array = [1, 2, 3];
        let _ = array[10];  // Panics - deterministic, debuggable failure
    }

    #[test]
    fn test_safe_get() {
        let array = [1, 2, 3];
        assert_eq!(array.get(1), Some(&2));   // Valid index
        assert_eq!(array.get(10), None);       // Invalid index - no panic
    }
}
```

The `#[should_panic]` attribute lets you verify that dangerous operations fail predictably. Unlike C's undefined behavior, Rust's panics are consistent and testable.

Quick Reference

Slice Types

Type	Description	Length Known
`[T; N]`	Fixed-size array	Compile time
`&[T]`	Immutable slice (fat pointer)	Runtime
`&mut [T]`	Mutable slice (fat pointer)	Runtime
`Vec<T>`	Growable vector	Runtime
`String`	Growable UTF-8 string	Runtime

Access Methods

```
// Direct indexing - panics on out-of-bounds
let value: T = collection[index];

// Safe access - returns Option
let maybe_value: Option<&T> = collection.get(index);

// Slice creation - panics if range invalid
let sub: &[T] = &collection[start..end];

// Safe slice - returns Option
let maybe_sub: Option<&[T]> = collection.get(start..end);
```

Key Copy Operations

```
// Requires exact size match - panics otherwise
dest.copy_from_slice(src);

// Clone elements (requires T: Clone)
dest.clone_from_slice(src);

// Safe copying via slicing
dest[..src.len()].copy_from_slice(src);
```

C vs Rust Comparison

C Pattern	Rust Equivalent	Safety Mechanism

`strcpy(dest, src)`	`dest.copy_from_slice(src)`	Size match required
`array[i]`	`array.get(i)`	Returns `Option<&T>`
`malloc` + manual tracking	`Vec::new()`	Length stored with data
`char buffer[N]`	`String`	Automatic growth