1. Describe source of vulnerabilities in the original C code

C code vulnerabilities.

Out of Bounds Payload:

For each UserStruct within the database, the array for username is only 50 bytes long, defined by
#define MAX_NAME_LEN = 50.

When update_username() is called with a new_username char array that has more than 50 bytes, it copies this new_username into the username array through the copy_string() function without checking whether or not it exceeds the 50 byte limit, thus resulting in writing to memory beyond the username array.
Since the memory for the UserStruct_t objects are sequential within UserDatabase_t, this eventually results in the password array of another user being written into.

Double Free Payload:

In the loop, for 13 times, Alice and Bob are logged in and the database is updated with update_database_daily().

Past the INACTIVITY_THRESHOLD, inactive users are freed through free_user().
This means free_user() is called on Eve more than once, after the 10th day.
There is no check as to whether free() cas was already called on a pointer.

This in turns calls free() on the same region of memory/same pointer.

Use After Free Payload:

UserStruct_t pointers for Mallory and Eve freed once after crossing the INACTIVITY_THRESHOLD.
After that, object for Charlie is added and it is stored at the same region of memory as Mallory was.

Later on, when Mallory tries to update her password through the UserStruct_t pointer, that memory address stored by the pointer is for Charlie.
There is no check as to whether or not the UserStruct_t pointer for Mallory was freed, hence it is a user-after-free error and password for Charlie gets updated instead,

Also, the pointers for the users are still stored in the array.
Eve's freed pointer remains in the database array, so subsequent operations on Even also target arbitrary reallocated memory, potentially leaking or corrupting another user's password.

2. Describe process of translating to Rust, and difficulties

Converting to Rust code.

For each function: for each of the arguments passed into the function, need to consider whether or not the caller of the function wants to retain ownership of the variable.
If caller of function wants to maintain ownership, then need to pass in a reference.
If want to modify the reference, make it mutable.
If caller of function is okay with handing over ownership, then can pass by value instead.
Make the variable mutable if we want to then mutate it.

Similar logic applies for the return values of each function.
If we want to hand over ownership of the returned object, then we can just return the variable.
But if not, for example in find_user_by_id, where the UserDatabase continues to have ownership of the users,
we just return a reference to the variable. Once again, make the reference mutable if we want to modify it.

Also had to understand how Option<> and Box<> worked, as well as the associated syntax.
Accessing Option<Box<UserStruct>> requires unwrapping before use, and mutable borrows must not overlap with ofer references to the same element.

Functions that return references into data structures (e.g., finding a user in the database) must declare explicit lifetimes so that Rust knows the reference cannot outlive the database itself.
The 'a ties helped to tie the lifetime of returned reference to lifetime of database.

Sources:
https://www.w3schools.com/rust/rust_syntax.php
https://doc.rust-lang.org/rust-by-example/
https://users.rust-lang.org/t/what-is-a-in-rust-language/37378