Segmentation Fault vs panic in Rust:

In C, the behaviour for accessing an array outside of it's bounds is technically speaking undefined. The result is usually that the program will then try to access memory outside the area that is allocated for the array. That might have no effects such as when the memory that is accessed is for example just used as padding so that data structures on the heap/stack can be better aligned. Also, it might cause memory corruption, so memory that is used for other variables as well is altered in an unintentional way. This might also result in the modification of a return address that is stored on the stack and when the program processes an malicious input, this might be used to build an exploit that allows code execution. When as a result of either this memory access is an access to a memory addresn in an illegal way (the address is not mapped or the access is a write but the address is mapped read only), the program will be stop with a segmentation fault. The segmentation fault might also happen later due to memory corruption, but it's not certain that a segmentation fault will happen and a code execution might also be the result of the memory corruption or the out of bounds array access.

In Rust, the behaviour for accessing an array outside of it's bounds is defined behaviour, so there is always a panic and the program (strictly speaking the thread) will terminate in a controled way. No memory corruption occurs.

C type safety:

- C is not a type safe language since it doesn't enforce that only operations on variables are performed that are permitted for the type of the variable. For example, you can write to the 5th position in an int array of length 4 or you can do pointer arithmetics resulting in a string pointer that suddenly points to an int in memory.

Strlcpy:

- It takes the length of the destination buffer as argument, not the maximum number of bytes/characters to copy. This is a more consistent way of dealing with the memory allocated for that string.
- It makes sure that the resulting string is always NULL terminated, so the maximum length of that string is l-1. Using strncpy, the resulting string might not be NULL terminated, so that when the resulting string is later used by a function that assumes that it's NULL terminted, this might lead to an array out of bounds memory access.

Calloc:

- Calloc intizializes the allocated memory with zeros. Therefor, accessing the freshly allocated memory by reading from it is defined behaviour.
- Calloc tecnically takes two arguments, but malloc only takes one. Malloc is often called like "malloc(n * sizeof(x))", but the multiplication may overflow and not enough memory is allocated. The corresponding calloc call is then "calloc(n, sizeof(x))", and calloc performs the multiplication by itself, checking for a potential overflow. (There was a problem on one of the lecture slides, so that problem wasn't illustrated very well on them)

Valgrind:

- Valgrind will in general find missing free statements. Missing free statements in general just cause memory leaks, which might result in the program running out of available memory when they occur very often. However, when just a small amount of memory is leaked, they will have no noticeable

effect when the program is tested. Valgrind will find them anyway. Still the missing free statements are security relevant when they can be triggered by an advresary so that the program runs out of memory and the service becames unavailables. (DoS)

Rust fuzzing:

- Rust programs without unsafe in general cannot crash with a segmentation fault since Rust is a memory safe language. However when certain events occur (.unwrap( ) is called on a result that was an Err, calling unimplemented, accessing an array out of bounds), the corresponding code will panic and this can potentially be detected by a fuzzer such as ALF.

Rust grade feedback:

The match statement needs to be exhaustive, so adding the following line to the match statement fixes that:

```
_ => "Sorry, that's not a valid grade"
```

Rust memory management:

This assignment is a bit tricky. To actually return the vector, we can use “std::mem”, and then replace the function body with “mem::replace(&mut self.l, Vec::new())”. This will replace self. With a new vector and return the old one.

A slightly worse solution would be to use a clone-like approach. This way, the entire vector needs to be copied, which costs ressources. Also, technically not “the vector” is returned, but a new one.

ASAN:

- It will detect forgotten “free” statements
- Usually a off-by-one array access does not lead to a crash immediately, but this can be detected with AFL
- Use-After-Free bugs can also be spotted
- Read access to not initialized memory

Not that great answers:
- It willl detect memory corruption bugs – Some of those bugs lead to crashes pretty soon, and not all kind of memory corruption bugs can be found with AFL
- Access to an address withins the programs virtual memory – When the access is a write but the memory region is mapped read only, this will result in a direct crash
- Access to an invalid address – Will usually be caught by the OS and result in a crash when the address is not mapped
- Freeing an address twice – This might be detected by the malloc/free implementation easily, but it’s a valid answer as well

Constant timing:

You can use a disassembler/debugger such as gdb or objdump to inspect the generated machine code. Conditional jumps are easy to recognize there. Memory access patterns can be spotted this way as well but require a bit more effort.

CBMC is not useful for this task (but for other things).

AFL in a security pipeline:

- You can test many different configurations
- It doesn’t block the developers machine
- You can never forget to run AFL since that happens automatically
- It will definitely run on the version of the code that has been committed to the server and not on some local one that was maybe not fully included in the commit/push.
- It can run for much longer on a dedicated server than on a developer machine
- It will run in a precisely defined environment
- The version of AFL used for the fuzzing can be centrally managed and updated
- Crash artifacts are available for all team members immediately
- Tickets/bugs can be automatically created when the fuzzer detects a crash
- It’s also possible to merge merge-requests automatically when the fuzzer finds no errors

Cross side scripting:

Sanitize your output is the key to prevent cross side scripting attacks. Whenever you emit HTML code to the client, make sure that everything that is supposed to be text is properly escaped/encoded.

Other patterns here are valid as well.

Web application security:

SQL Statements are build using simple string concatination, which makes them vulnerable to SQL injection. To counter this problem, prepared statement can be used. Instead of the input values, placeholders (such as ?) are placed where the input values belong in the query string and then later, those are linked with the input through an API that takes care about escaping those properly.