Assignment:
Formal Verification of a Logging Daemon

Prof. Bart Jacobs

1 Motivation

Included with this assignment is the C program `logd.c`, which implements an extremely simple logging daemon, i.e. a server that allows clients to append messages to a log, to retrieve the current contents of a log starting from a given offset, and to "follow" a log, retrieving messages as they are appended by other clients.

It is generally very difficult to verify the absence of buffer overflows, use-after-free bugs, double frees, arithmetic overflows, and similar security vulnerabilities in C programs. Common approaches, such as testing and code review, are very bad at detecting the subtle corner cases that could lead to an exploitation. While bug finding tools are available commercially, these suffer from *false positives* (warnings that do not correspond to actual problems) and *false negatives* (the absence of a warning for an actual problem). Such tools cannot generally be used to achieve strong confidence in the absence of buffer overflows and similar security vulnerabilities from a C program.

In many cases, the best solution for this problem is to not write the program in C in the first place. In some cases, however, C is necessary to achieve the control, performance, portability, interoperability and meet the resource constraints desired.

Fortunately, you have learned VeriFast, a formal verification approach that does guarantee[1] that it detects all accesses of unallocated memory (such as buffer overflows), as well as certain other programming errors, such as data races. By applying VeriFast to `logd.c`, you will be able to expose `logd` to the Internet, confident that no hacker will be able to exploit it to infect your machine.

2 Assignment

The goal of this assignment is to prove that the C program `logd.c` does not access unallocated memory and does not have data races or integer arithmetic overflows. More specifically, you must add annotations to `logd.c` such that VeriFast accepts the program. Your solution is considered to be correct if you

[1] assuming the absence of bugs in VeriFast itself

did not modify the C code itself and the command

```
verifast stringBuffers.o threading.o sockets.o logd.c
```

outputs

```
logd.c
0 errors found
Linking...
Program linked successfully.
```

This command checks that the program verifies (with overflow checking enabled), that `main` has a sound contract and that the program contains neither lemmas without proofs nor `assume` statements. It is not necessary for specifications to be complete; it suffices to prove the absence of memory errors, data races, and integer arithmetic overflows. Note that the `leak` command may be used in your solution.

Your final solution must be submitted to bart.jacobs@cs.kuleuven.be no later than December 31st, 2021, 23h59. It suffices to send the annotated version of `logd.c`. If you get stuck, send an email describing your problem to bart.jacobs@cs.kuleuven.be and we will reply with a question that should get you unstuck, in Socratic style. In fact, we strongly encourage asking for help if you get stuck: if your final submission is incomplete, we will take into account the quantity and quality of your questions!

In your submission e-mail, specify your availability for the oral defense in the second to sixth working day after the day you submit. During the defense you will be asked to explain VeriFast's symbolic execution algorithm, by predicting the symbolic states encountered during the symbolic execution of a few of the functions of the assignment. The defense will take place in the office of Prof. Jacobs (200A 05.50).

You will also be asked about the material from the theory lectures, but these questions only count for three points out of twenty.

3 Hints

- Some annotations are already given. These are correct; do not remote them.
- Carefully consider at each point whether to assert full ownership or fractional ownership of a given resource. Do threads access the resource for reading without synchronization, or do they access it only while a mutex is held?

4 Compiling and Running the Program

Compiling

First, install `gcc`. (On Windows, install MSYS2 and then, in an MSYS2 MSYS shell, run `pacboy -S gcc:i`.)

Then run `gcc *.c` . (On Windows, run `gcc *.c -lws2_32` from inside an MSYS2 MINGW32 shell.)

Running

To run the program, first create some log files and then run the executable generated by the compiler, passing the names of the logs:

```
$ echo -n > testlog1
$ echo -n > testlog2
$ ./logd testlog1 testlog2
```

The server will listen on port 1234. Run any telnet client to use the program. For example:

To append "Hello world!" and "Bye for now!" to `testlog1`:

```
$ telnet localhost 1234
testlog1
APPEND
Hello world!
Bye for now!
```

(In most telnet clients, to end the session, press Ctrl+] and then enter the `close` command.)

To list the contents of log `testlog1`, starting from offset 6 and transferring at most 5 bytes:

```
$ telnet localhost 1234
testlog1
LIST
6
5
world
```

To follow log `testlog1`, starting from offset 13:

```
$ telnet localhost 1234
testlog1
FOLLOW
13
Bye for now!
```