CS527/599-SWS Project (2016 Spring)

Due dates:	
Development:	04/17/2016
Hacking:	04/27/2016
Patching:	04/30/2016

1 Introduction

In this programming project you have to deal with a develop, hack, patch, repeat (dehapa) challenge[1]. In these type of challenges you are given a program description and you have to develop that program (development phase). Then you inspect the code written from the other teams searching for flaws and vulnerabilities (hacking phase). Finally, you collect feedback about your code from other teams and you try to fix the bugs (patching phase).

2 Program specification

The target program will have low complexity and hopefully, you can keep your implementation simple as well. Remember, simplicity and good coding practices make programs safer. However, a straightforward implementation will probably have either security or performance issues, so be careful!

The target program implements an online, shared file editing service (we will call it *ONID* – ONline fIle eDiting service). Your task is to implement the functions to a pre-defined API. You are given all function and structure declarations as a basis. This API is then used both in a server and a client (and our test driver to test functionality). You are free to add any additional helper functions or data structures but you must stay binary compatible with whatever goes over the wire. The development platform is 64-bit x86 Linux and your program must compile with both LLVM and GCC on Ubuntu 15.10.

ONID consists of 2 parts: a client and a server that provides the remote service to the clients. A user may read a specific block from a file, make changes to it, and either save it or discard the changes. The client will interact with the user and will send the right commands to the server. The server will process these commands and will return the results back to the client.

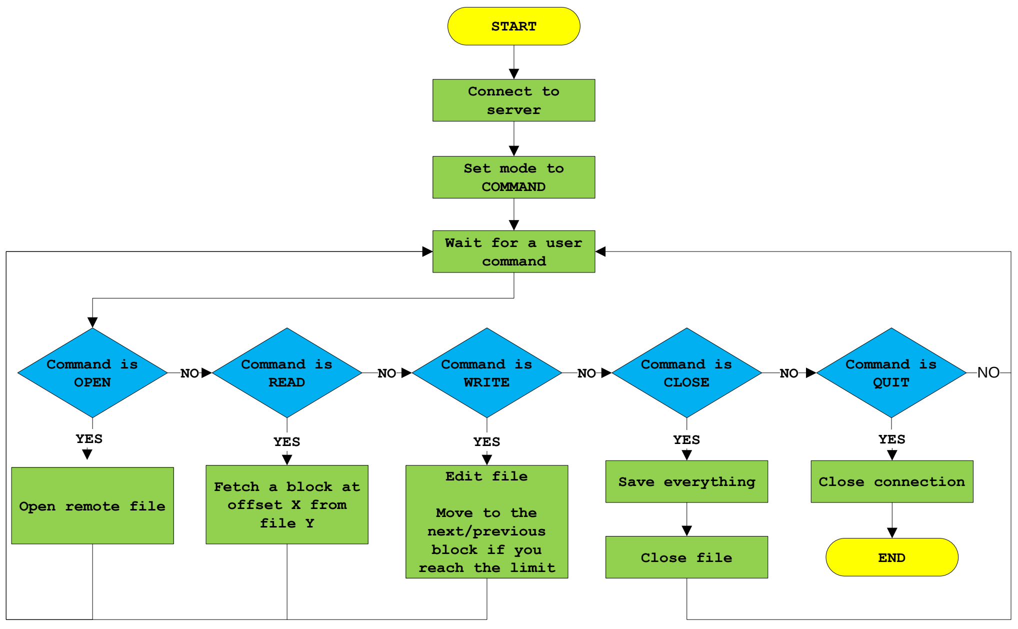
[1]Hat tip to both the *Applied Security Laboratory* lecture at ETH Zurich and the *Build it, break it, fix it* contest at the University of Maryland for the inspiration for this project.

2.1 Client

As you can see in the figure below, the client should support several operations. We are going to call these operations "modes". See **??** for an overview.

In OPEN mode (i.e., a file has been opened for reading), the client will open the remote file. In READ mode, the client retrieves a block from the server and will display it to the user. In WRITE mode, a user should be able to make changes to the file (she should be able to move the cursor on the screen and change individual bytes, comparable to the REPLACE mode in vim). In case that user moves before/beyond the current block, the next/previous must be fetched. When we fetch a different block, we should keep both the old and the new block available for any changes. Fortunately for you, files can only grow (there is no "delete" mode). Note that no changes should be done to the file, unless the user explicitly specifies it using the CLOSE command.

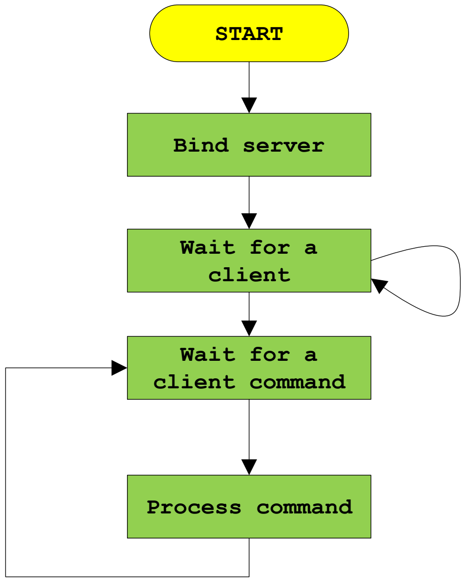
Once you enter in the WRITE mode, you can exit and save using ˆC (Ctrl + C), or exit without save using ˆX (Ctrl + X).



2.2 Server

The server has to interact with the client. As you can see in **??**, it has a much simpler flow chart, which means that you have more freedom on how to implement it. However this does not mean that it is simpler. File permissions and bound checking on files are some things that you have to consider.

An easy way to implement this is by taking a backup of the original file and then directly applying any changes to it. If a user saves the file, you delete the backup file. Otherwise (if the user discards the changes, you remove the new file and rename the old file). The problem with this approach is that it can be very slow due to the heavy I/O and can be very inefficient for many clients. So you need a more efficient way to implement it.



3 Additional Notes

The client should communicate with the server using special packets. Take a look at the source file for more information. You are not allowed to send data in a different format. Even if you want to send a single bit, you have to use this packet format.

Note there is no output format. This means that we will check correctness of your code, based on the “results”, i.e., we will check whether your code can successfully modify the files without any errors.

The deadlines are at 11:59pm Eastern Time and the archived (tar.gz) version of your projects must be sent to the TA by the deadline. For the hacking contest, you have to send a report (description and proof of concept) for each bug you find. For the patching part, you have to submit an updated archive of your project and a report on how you addressed the individual bugs.

4 Scoring

To score your developed implementation, we will test the functionality of your code by using your API in a set oftest programs. Your code must be functional and consistent with the specification. The maximum possible score is

100, whereas both conformance to the specification and performance will be evaluated.

For every bug/problem that is found in your code during the hacking phase, you will lose between 5 and 15 points. You lose points for every bug you have. This means that the number of students that report the same bug, does not affect the number of points you lose.

Finally at patching stage, you can get some of your lost points back by fixing your bugs. For any bug you patch, you recover 50% of the lost points. For example, if you lost 10 points for a bug, you will get 5 points for fixing it. You will get no points if you fix bugs that were not reported during hacking phase. After all phases, the maximum possible score is 100 for the development and patching phases and up to X points for finding vulnerabilities in other people's code. The minimum amount of points is 0 (obviously). During the hacking phase you do not have to actually exploit the bug, but a proof of concept attack is enough. However, bonus points will be given for those who'll come with successful exploits.