Chad Miller
CS 5959
1/24/2014

GNU Coreutil ls

The goal of this assignment is to critically evaluate *ls*, i.e., to determine if it is reliable enough to be used in a program that guides a space ship down to the surface of the Moon. Naturally, one may ask, how is it possible to determine that a program is 99% correct? One answer may be that it is a several step process, each step building the possibility of a reliable program. The steps that I'll follow will be tested on Coreutils-8.20 and include: compiling the program with Clang using options *-fsanitize=undefined* and *-fsanitize=integer* (note that these check for undefined behavior and integer overflow errors), using Valgrind to search for memory leaks, using Clang's static analyser and, finally, using Coreutil's own test suite.

Many arithmetic errors in a program may be identified using compiler options. When building Coreutils with the option *-fsanitize=integer* we immediately get the following:
make-prime-list.c:45:25: runtime error: unsigned integer overflow: 514273206 - 198357697807138827 cannot be represented in type 'unsigned long'
*make-prime-list.c:45:22: runtime error: unsigned integer overflow: 2 * 18248386376416685995 cannot be represented in type 'unsigned long'*
*make-prime-list.c:45:28: runtime error: unsigned integer overflow: 18248386376416685995 * 18248386376416685995 cannot be represented in type 'unsigned long'*
*make-prime-list.c:45:30: runtime error: unsigned integer overflow: 13112262871895739961 * 3 cannot be represented in type 'unsigned long'*
*make-prime-list.c:104:30: runtime error: unsigned integer overflow: 5001 * 12297829382473034411 cannot be represented in type 'unsigned long'*
With a little further examination we see that this is used by the test suite and may or may not imply an error in the program. However, with 'ls' we get the following warning:
ls.c:1981:24: warning: adding 'enum indicator_style' to a string does not append to the string [-Wstring-plus-int]
for (p = "=>@|" + indicator_style - file_type; *p; p++)*
When stepping through the code with GDB, we find that this section is ran if *ls* is passed an indicator style, such as, *ls –indicator-style=classify.* Both *indicator_style* and *file_type* are enums and global variables and hence this arithmetic on *p* may seem awkward. However, when printing *p* we see that it is removing from *p* one of the indicator symbols "*=>@|" each iteration, presumably classifying each of the file types with a symbol. Hence they are not trying to append to *p* but just do arithmetic with it, so the warning can be ignored.

A second step in building a reliable program, is that it doesn't cause critical memory leaks. For example, imagine a concurrent server that never freed memory allocated for each connection. In running Valgrind on *ls* we get the following results:
==7142== LEAK SUMMARY:
==7142== definitely lost: 0 bytes in 0 blocks
==7142== indirectly lost: 0 bytes in 0 blocks
==7142== possibly lost: 0 bytes in 0 blocks
==7142== still reachable: 35,204 bytes in 59 blocks
==7142== suppressed: 0 bytes in 0 blocks

We see that when *ls* exits there are 59 blocks that are still allocated. However, these blocks do not remain allocated, they are freed by the kernel when the process exits. In an article titled *Understanding*

Valgrind Memory Leak Reports by Aleksander Morgado: *'Still Reachable' memory leaks are not considered harmful in the standard development... instead of explicitly freeing memory before the return of the main function, it's just easier and faster to leave the kernel do it automatically when the process ends.* Hence it depends on a reliable kernel.

Static analysis may be thought as: *Analyzing code without executing it and an automated code review*[1]. When building Coreutils with Clang's static analyzer we get a longer list of warnings and errors spun out as a web page:

Bug Type	*Quantity*	*Display?*
All Bugs	*43*	
API		
Argument with 'nonnull' attribute passed null	*9*	
Dead store		
Dead assignment	*4*	
Logic error		
Dereference of null pointer	*14*	
Result of operation is garbage or undefined	*2*	
Uninitialized argument value	*5*	
Memory Error		
Memory leak	*6*	
Use-after-free	*3*	

None of these errors pertained to *ls*. So we will assume that it passed.

Finally, the last way we'll test is by using Coreutil's test suite. To run the tests, I changed directory to *tests* and ran *make check*. I couldn't figure out how to only run tests for *ls*, so this tested every command. The results were in test-suite.log:

```
==================================================
  GNU coreutils 8.20: ./tests/test-suite.log
==================================================
```

All 439 tests passed. (93 tests were not run).

The only tests that were not run were considered too expensive by default. Of the 32 tests for *ls* 31 were ran and passed.

To summarize, we examined compiler warnings using both *-fsanitize=undefined* and *-fsanitize=integer* on *Coreutils-8.20*. Our results were a test suite with five overflow errors and a total of ten warnings overall, only one warning for *ls*. We also examined *ls* with Valgrind and found it to have 59 still reachable blocks when the program exited. However, we learned that these are cleaned up by the kernel and don't cause critical leaks. Next we used Clang's static analyzer and although it stated 43 bugs overall, none of them pertained to *ls*. Finally, we used Coreutils own test suite, which passed 31 tests. It speaks very highly of the GNU Coreutils team for having such a large project giving so few errors and warnings. In *ls* alone, there are nearly 5000 lines of code. Since every level of testing passed I would recommend this version of *ls.*

1. http://stackoverflow.com/questions/49716/what-is-static-code-analysis