Final Paper Proposal

Colin Samplawski, Malcolm Reid, and Keith Funkhouser

CS 736 - Fall 2016

1 Introduction

Many commonly used C library functions return some value. In most cases, these values indicate something that is relevant to the functions operations, such as a file descriptor, number of bytes processed, or a pointer to some data. Additionally, many functions that can fail return some predefined value that indicates that an error occurred during the function's execution. Unfortunately, many programmers have the bad habit of not checking these return values for such error conditions, which can lead to unexpected behaviors. In our work, we plan to analyze a suite of standard Linux applications and see how they handle errors returned by commonly used system/library calls.

2 Motivation and Goals

In their paper [1], Miller et al. modified the `malloc` library function to return `NULL` with some probability indicating that memory allocation has failed. They then tested a variety of Linux utilities using this version of `malloc` to test if the programs correctly handled this error case. Surprisingly, the majority of these utilities did not correctly handle this case and crashed in a variety of ways. This motivates us to expand on this idea and test significantly more system/library calls in a similar way.

In expanding on this work, we want to not only find crashes, but also investigate and analyze their sources. In doing so, we hope to determine where and why these calls are not being checked and whether there are any patterns. We would also like to develop a tool that others can use to test for return value errors. Likewise, programmers may use this tool as part of their test suite to confirm that their own programs handle error cases in the way expected. Table 1 displays a subset of calls which we plan to investigate.

3 Method

We do not expect our process to be terribly difficult to implement. Our main challenge is designing a way to wrap the system/library calls of interest. We need to be able to intercept calls in a way that is invisible and non-disruptive to the application being tested. To solve this problem, Miller et al. extracted the binary of the call of interest (in their case `malloc`) and used a binary rewriter to rename `malloc` to `_malloc`. They then wrote a new function called `malloc` which was called by the applications being tested. This new version

Table 1: First round of calls to test

System Calls	libc calls	Pthread calls
close	free	pthread_cond_init
creat	kmalloc	pthread_create
dup	malloc	pthread_mutex_init
fork	memccpy	
ioctl	printf	
mkdir	strto*	
mmap		
open		
pipe		
read		
write		

returned an error value with some probability or just passed the call along to `_malloc`. We have come up with a similar solution that has a lower startup cost.

Linux systems have a built in environment variable named `LD_PRELOAD` which allows users to load a shared library before starting an application. Most importantly, these preloaded libraries take precedence over any other libraries loaded by the application. Therefore if this preloaded library contains a function named, for example, `open`, any calls to `open` by the application will invoke the preloaded `open` and not the version found in the system library. This allows us to intercept any call and return an error message with some probability. In order to call the real version of `open`, the `dlsym` function is used. This function searches through dynamically loaded libraries and returns a function handle to a function whose name is given as a argument. This handle can then be used to call the original version of `open`. The code used to wrap `open` is given in code listing 1. This code is compiled into a position independent shared library file to be used with `LD_PRELOAD`. This implementation is based on a tutorial found at [2].

There are some uncertainties that arise when using `LD_PRELOAD`. Firstly, if a function from another library calls `open` it may bypass our wrapped version. Similarly, if the library containing `open` (`stdio`) is statically linked to a program, the original version may be called. We are currently investigating ways to handle these problems, and we feel that we can design an implementation using `LD_PRELOAD` that is robust to these situations. If not, we will use a method similar to the one found in Miller et al.

Listing 1: open wrapper

```
#define _GNU_SOURCE  //needed to compile as PIC
#include <dlfcn.h>   //dlsym
#include <stdio.h>

//function pointer for real open
static ssize_t (*real_open) (const char *pathname, int flags) = NULL;

//open wrapper
ssize_t open(const char *pathname, int flags) {
   printf("wrapped read\n");             //do something before calling real open
   real_open = dlsym(RTLD_NEXT, "open");  //get addr of real opean
   real_open(pathnae, flags);            //call real open
}
```

4 A Small Example

As a proof of concept, we used a wrapped call to `open` on a handful of common Unix utilities: `grep`, `gcc`, and `cat`. Our call to `open` set `errno` to `EACCES`, which indicates that permission to the file is denied. Interestingly, `grep` does not make any calls to `open`. A brief overview of the source code reveals that it calls `fopen` and `fts_open` instead. Furthermore, within the `fopen` source code, calls to `open` have a different function signature from our wrapper and therefore bypassed it. We will have to be cognizant of this overloading in the future.

We found that `gcc` and `cat` were more straightforward. We found that `gcc` made 70 calls to `open` when compiling a simple one source file C program. When we returned errors probabilistically, we found robust handling of errors in a variety of locations. In some situations when a header file could not be opened, we received the error message "(Permission denied) Bug not reproducible." This suggests that `gcc` reattempts to compile the source file at least once. This is an example of a behavior that we will need to further investigate. Finally, `cat` displayed the simplest behavior: it either executed successfully or terminated and reported "Permission denied."

5 Schedule

5.1 Important Dates

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Oct 24th	Oct 25th	Oct 26th	Oct 27th	Oct 28th	Oct 29th	Oct 30th
Oct 31st No class	Nov 1st	Nov 2nd No class	Nov 3rd	Nov 4th No class	Nov 5th	Nov 6th
Nov 7th	Nov 8th	Nov 9th	Nov 10th	Nov 11th	Nov 12th	Nov 13th
Nov 14th No class	Nov 15th	Nov 16th No class	Nov 17th	Nov 18th	Nov 19th	Nov 20th
Nov 21st	Nov 22nd	Nov 23rd	Nov 24th Thanksgiving recess	Nov 25th Thanksgiving recess	Nov 26th Thanksgiving recess	Nov 27th Thanksgiving recess
Nov 28th	Nov 29th	Nov 30th	Dec 1st	Dec 2nd	Dec 3rd	Dec 4th
Dec 5th No class	Dec 6th	Dec 7th No class	Dec 8th	Dec 9th **Paper draft due to referees**	Dec 10th	Dec 11th
Dec 12th	Dec 13th	Dec 14th **Paper reviews back to author**	Dec 15th	Dec 16th	Dec 17th	Dec 18th
Dec 19th **Final project papers due**	Dec 20th	Dec 21st **Project Poster Session**	Dec 22nd	Dec 23rd	Dec 24th	Dec 25th

5.2 Weekly Work Goals

Week 1 (10/17 - 10/23): Submit project proposal

Week 2 (10/24 - 10/31): Write wrapper code generation code, test one call

Week 3 (10/31 - 11/06): Analyze source code, investigate crashes

Weeks 4-6 (11/07 - 11/27): Expand scope - test more calls and more programs

Week 7 (11/28 - 12/4): Start writing paper, package project code as tool

Week 8 (12/5 - 12/11): Finish writing paper

Week 9 (12/12 - 12/18): Revise paper, work on poster

References

[1] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services", *Computer Sciences Technical Report #1268*, University of Wisconsin-Madison, April 1995.

[2] S. Barghi, "How to wrap a system call (libc function) in Linux", September 2014. `http://samanbarghi.com/blog/2014/09/05/how-to-wrap-a-system-call-libc-function-in-linux/`