Fuzzing System Call Return Values

Sripradha Karkala
skarkala@cs.wisc.edu

Kavin Mani
kavin@cs.wisc.edu

Abstract

In this paper, we have challenged the robustness of common UNIX utility programs in our project by testing the issue of unchecked return values. We ran checks on several UNIX utilities and applications by failing the system calls pertaining to different domains including memory, file, process management, and network operations. To conduct such an experiment, we developed a shared library which returned the standard error values of UNIX system calls probabilistically on execution, which means that when a system call is invoked, instead of being sure that the call will be executed by the system, we deliberately fail the call and return failed return value. We adopted library interposition techniques to manipulate the target applications to use the overridden wrapper functions rather than the standard system calls. The result of our experiment was that 25% of the programs that were tested (30 out of 120) crashed. We managed to crash several popular UNIX utilities such as ps, netstat, ifconfig, and some GUI-based (Graphical User Interface based) applications like bitmap, firefox, and evince (PDF viewer). The most commonly observed reasons for crashes were segmentation fault, deadlock, failure in shared libraries, incorrect loading of user interface (UI) and trace/breakpoint traps. We have analysed them to identify the exact cause and the location of the failure. Evaluation of the crashes unearthed bugs in popular programs like gdb, vim, and in shared libraries like glibc.

Keywords

Fuzz Testing, library interposition, stacktrace, crashes, segmentation fault, shared libraries, return value, deadlocks.

1.Introduction

Bad programming practices can often lead to failure of a program, unexpected behaviors, loss/corruption of application data or deadlocks. One such common practice during program development is the execution of a program without checking the return value of function calls. Return values are an indication of success or failure of the execution of that function. Ignoring the return values can lead to crashes, incorrect/partial results, infinite loops or undesirable execution paths in the program. Hence, it is critical to validate return values as they define the reliability of applications.

We intend to test the robustness of programs by modifying the return values of commonly used system calls. System calls in UNIX are extensively used in a variety of utilities and applications. We want to identify and analyze the behavior of these utilities and applications when the system calls return values are deliberately modified. We adopt library interposition techniques to manipulate the return values.

Library interposition is the process of placing a new or different library function between the application and its reference to a library function [1]. This technique is mainly used to intercept function calls to code in shared libraries. It is achieved by preloading user specified code and then making the dynamic linker reference the user's function definition without using the normal library search path. In order to successfully interpose a program, the LD_PRELOAD environment variable must be initialized with the intended shared object that would interpose the program. Setting the LD_PRELOAD variable makes the dynamic linker first check for definitions of the interposed function in the objects initialized by this variable. Figure 1 demonstrates the working of library interposition technique [1]. When the target application makes a call to a function redefined in the interposed library, instead of calling the original function, the call is first intercepted by the interposed library. The original function is then executed depending on the logic defined in the interposed library.

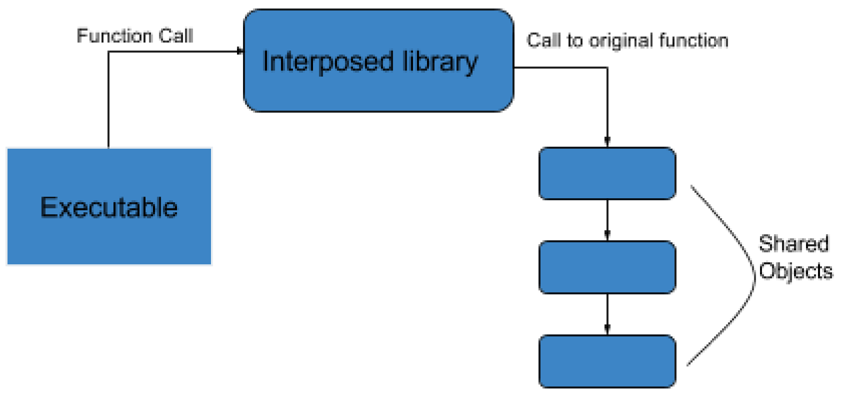

Figure 1 : Library Interposition

Interposition is a useful technique because it permits the interception of function calls and collection of information between such calls without needing the recompilation of the shared objects, enabling us to create a custom library that generates that required information while not modifying the shared libraries. It is effective, as a lot of commercial operating systems and proprietary softwares do not open source the source code of their products. This interposed library can be created without the requirement of the source code of actual shared libraries on which the interposition is to be performed [2]. When an interposing library intercepts a function call, it has the ability to record information or even modify the arguments that are passed to the original function call. The interposed function can be manipulated in many ways. It can be made to directly execute the original function call and return a different return value, execute only a portion of the original function call or even execute a completely different function. The interposing library can also collect data such as the number of times a specific function was called, monitoring file access requests to obtain file usage statistics, detecting race conditions and recording memory and data usage during the execution of the program [3].

Library interposition technique do have a few caveats. First, LD_PRELOAD is ignored for programs with the SUID permission bits set for security reasons. Since library interposition allows a program to do almost anything one wants it to, Linux prevents the program from modifying the behavior of a program running on behalf of another user or group. Second, one cannot interpose internal library function calls, since these are resolved at compile time. For instance, if some function in libc calls malloc, it will never call a wrapper function from a different library.

1.1 Other function call tracing methods

Many flavors of UNIX such as Linux, MacOS, and BSD provide tools that allow the tracing of kernel calls. Some of the most popular tools are strace and truss [1]. These commands intercept and record the system calls that are called by a process and the signals that are received by a process. They provide a trace of the system calls executed by the application, showing the arguments and return values of each system call and the time spent in each. Both these commands are implemented using the trap mechanism by the operating system. Unfortunately, the trap mechanism incurs a high overhead and is not available for user level library functions [4]. Hence, they are preferred for statically linked applications over dynamically linked ones.

Static analysis is another useful tool for program tracing and bug identification at compile time [5], but it does not give us any program information during the execution of a program. Static program analysis requires access to the complete source code [6]. The analyzing tool will have to expand all the macros and function definitions across the various files involved prior to analysis and it becomes unwieldy when the complete set of shared libraries associated with the application code is expanded. It becomes impossible when we work with commercial software applications where source code is not available [7]. On the contrary, interposing library gathers data at runtime, making it possible to inform at the exact time when favorable conditions (for instance, return value checking) occur during the execution of a program. By logging sufficient data at this instant of time, it might be possible to trace down to the root cause of a problem (segmentation fault, deadlock) after its occurrence. Considering these facts, static analysis is still a valuable tool, but it does not meet our objective of extracting the amount of application level data needed for program analysis.

The rest of the paper is structured as follows: Section 2 describes the earlier work that has been done in this field. Section 3 describes the testing framework and set up for the experiment. Section 4 gives the details of implementation. Section 5 provides a summary and detailed analysis of the results. Section 6 gives a possible direction for future work and Section 7 concludes the paper.

2. Related Work

Fuzz Testing has been an important technique in automated or semi-automated software testing that involves providing invalid, unexpected or random data to the inputs of a computer program. In a study conducted by Miller et. al [8] in 1990, the authors primarily focused on testing the reliability of UNIX utilities. They managed to crash 25-33% of the tested UNIX utilities and presented a summary of the program bugs that caused the crashes. Analysis of these crashes included not checking array index out of bounds exception, not checking function return codes, bad error handling, and race conditions as some of the possible causes.

In another study conducted by Miller et al. [9] in 1995, they tested the reliability of a large collection of basic UNIX utility programs, X-Window applications and servers, and network services. They reported which programs failed on which systems, and identified and categorized the causes of these failures. The result of their testing was that they crashed (with core dump) or hanged (infinite loop) over 40% (in the worst case) of the basic programs and over 25% of the X-Window applications. Their study parallelled their previous study in 1990 and all utilities that were compared between 1990 and 1995 noticeably improved in reliability, but still had significant rates of failure. They also observed that reliability of the basic utilities from GNU and Linux were noticeably better than those of the commercial systems. A significant difference of this work over their previous one was that they also tested how utility programs behaved when bad return codes from the memory allocation library routines (malloc, calloc, realloc) were returned indicating the unavailability of virtual memory. They managed to crash almost half of the programs that were tested in this way.

According to Curry [4], library interposition is a demonstrated powerful technique to collect workload profiles and function trace information. It can be accomplished by having the linker resolve library functions to special wrapper functions that collect statistics before and after calling the real library function, leaving both the application and real library unaltered.

We have found the amount of detailed information that can be gathered has been useful in many stages of our project lifecycle including design, development, and debugging.

3.Testing framework

We developed a simple testing framework using C, which included the modified system call library and a bash script for automating the execution of the selected set of programs using the new library. Additionally, we logged the sequence of events that led to failure for reproducibility and generated core dumps for analysis of the programs that crashed.

For our experiments, we required a set of functions that are standard and are extensively used across a wide range of applications. A perfect candidate for this is the UNIX system call library. Table 1 lists the functions that were chosen to be overridden in the newly written shared library. The functions were picked to span across different use cases.

Class	System Calls
File Management	open, read, write, seek, dup2
Memory management	malloc, calloc, realloc, brk, sbrk
Network Operations	socket, accept, connect

Table 1 : System calls overridden in the new wrapper library

The programs subjected to testing were chosen from a diverse background based on the popularity and complexity of the applications. In the course of the project we tested a total of 120 programs. It included

- Simple UNIX utilities (e.g. man, cat, grep, ps, less, echo) that are widely used by users on a daily basis. Since these programs have gained immense popularity and stability over the years of usage, we decided it would be interesting to test the robustness of such programs.
- Larger UNIX utilities such as gcc, gdb, vim, ftp. New features are constantly added to such programs and they offer large code bases for testing.
- UI based applications gedit, firefox, evince, eog.

The programs were tested on a 64-bit, 1.2 GHz processor, and 1GB RAM virtual machine running Ubuntu stable version 14.04 and the stable version of the programs compatible with Ubuntu 14.04 were compiled with debug symbols.

3.1 Method for analysis

3.1.1 Code Inspection

Code inspection is a trivial testing method. For simple utilities that involve only a few hundreds of lines of code, we manually inspected the code to check if the return values of all systems calls under consideration are checked. It is also useful to inspect code to reach the code paths that are normally not executed. Eg. Error handling code, less known flags in UNIX utilities. We discovered a few instances in the code where the return values were not checked and the programs were bound to fail. For instance, for the ps utility, we found that there were several places where the return value of malloc was not checked. If malloc had failed, then trying to access the corresponding variables (that had memory allocated by failed malloc) can cause a segmentation fault. The segmentation fault issue with sort was quite unusual; upon inspecting the source code, we observed that all malloc calls except a couple that were issued were using a function xmalloc, whose definition executed malloc and checked for return value. The other two calls were issued as such without checking for return values. When we deliberately failed this function call, the sort utility crashed at the following line with a segmentation fault (sort.c : 2134).

3.1.2 Generating core dumps and stack trace evaluation

Core dumps were generated for all the programs that crashed. However, the core dumps by themselves were insufficient to understand the cause of the crash as the target programs were stripped off their debug symbols. Hence, we resort to obtain the source code of the failed applications by recompiling (-g flag in gcc) the programs with debug symbols. The recompiled applications were installed and tested again for failures. Evaluation of the stack traces produced by these crashes gave us an insight on the exact line where the program failed. Also, using stack traces did bring into light that some crashes/deadlocks were in the shared libraries rather than in the program itself. In case where the failures were in the shared libraries, we installed the debug symbol packages available to inspect the stack traces.

4. Implementation

We implemented a new shared library that included the wrapper functions for the system calls mentioned in Table 1. Each of the functions were purposely made to fail with a probability based on a random integer generated. The random integer is generated using the *rand* function. In any application, a system call will be called at multiple places. In order to ensure that we don't just observe failures happening at the first or last instance of the system call, we require a uniform distribution of failures in the function call for a given program. To handle this condition, we introduced a failure threshold value. Failure threshold value was calculated by dividing the random integer generated by RAND_MAX (2147483647). We arrived at the optimum percentage for failure threshold by testing the system calls for a variety of failure threshold values ranging from from 30% to 75%. We observed a uniform distribution in failures when the failure threshold range was between 45% to 50%. Hence, for all the future tests, we maintained a failure threshold of 50% as the ideal value. If the calculated failure threshold value was above the ideal value of 50%, the wrapper function failed and returned a bad return value. Otherwise, the original function was executed. To capture as many failures as possible in each of these programs, the programs were executed twenty times using a script. We decided the optimal iteration number to be twenty after increasing the number of iterations step by step until no new errors were encountered during execution. Although we didn't find any new errors after fifteen iterations, we wanted our results to be reliable and hence set the iteration count to twenty.

The wrapper functions were designed in such a way that they returned the standard failure return value when the random number was above the failure threshold. Otherwise, the function executed the actual system call.

For instance, the open system call was re-written as below

```
int open(const char * file, int oflag) {
// initialization and logging information
// If random seed > failure threshold
return -1
//else
return open(file, oflag); // call to the original function - open()
}
```

We make use of the dlsym function to obtain the address of the actual system call implemented in glibc and the compiler resolves the symbol at the point when glibc is loaded. This is informed to the compiler using the RTLD_NEXT flag. Recording the value of random seed and the name of the failed function was essential, since this enabled us to reproduce the crash for further analysis. The execution of the program for testing included linking the program with our shared library before any other libraries were loaded. This was accomplished using the environment variable LD_PRELOAD. For instance, if we tested *ls* program with *open.so* as shared library*,* it will be executed as below:

```
LD_PRELOAD=/path_to_new_lib/open.so /bin/ls
```

Thus, the function calls in glibc were replaced with the wrapper function implemented.

5. Results

Among the 120 utilities that were tested, 25% of the utilities crashed. Most of these programs crashed for more than one system call and for multiple reasons. Table 2 lists down the programs that crashed for a given system call.

Function Name	**Failed programs**
malloc	Netstat, ps, lsmod, gdb, cancel, corelist, cpan, dirsplit, ifconfig, vi,vim, telnet, gedit
calloc	sort, gedit, vi, vim, wish, hostnamectl, host, ghostscript, evince, dirsplit, dm-tool, eog, dig, cpan, corelist, cheese, checkbox-gui, cancel, gdb, loginctl
realloc	corelist, cheese, eog, evince, dmtool
open	cheese, firefox, eog, checkbox-gui, evince, gedit
brk, sbrk	No failures
socket	cheese, checkbox-gui, firefox, bitmap
read	checkbox-gui, cheese, gdb

write	checkbox-gui, cheese, eog , evince, firefox, wish, gedit
lseek	No failures
dup2	less, calendar, red
accept	calendar, less
connect	cheese

Table 2 : List of programs that crashed for given system calls

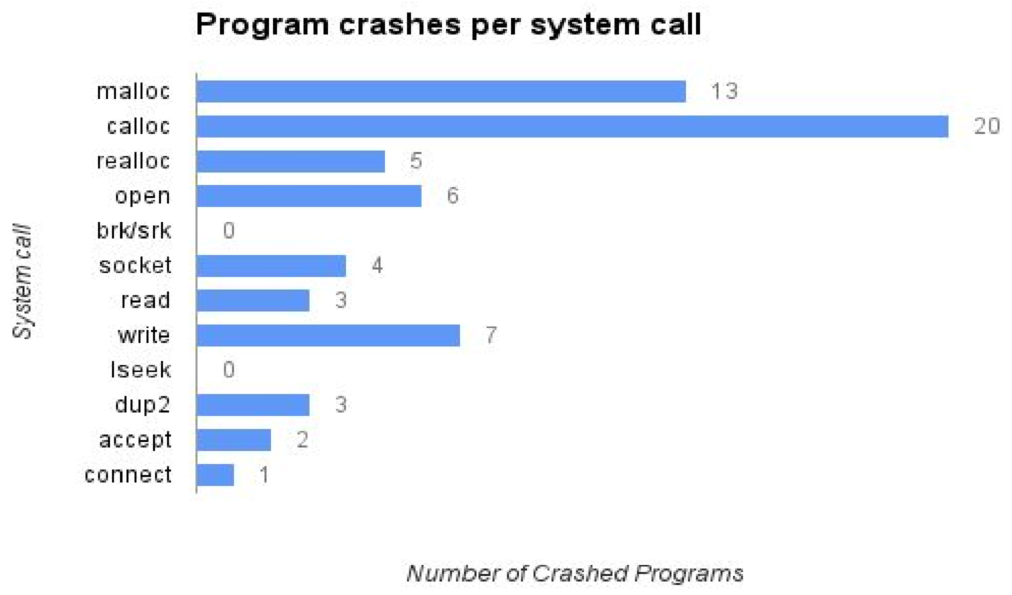

Figure 2 : Program crashes per system call

Figure 2 provides information on the number of system calls that failed per system call. We can see that most of the failures occur for memory allocation system calls, that is, malloc family. File management system calls such as open, read, write cause the next highest number of crashes followed by network management system calls .

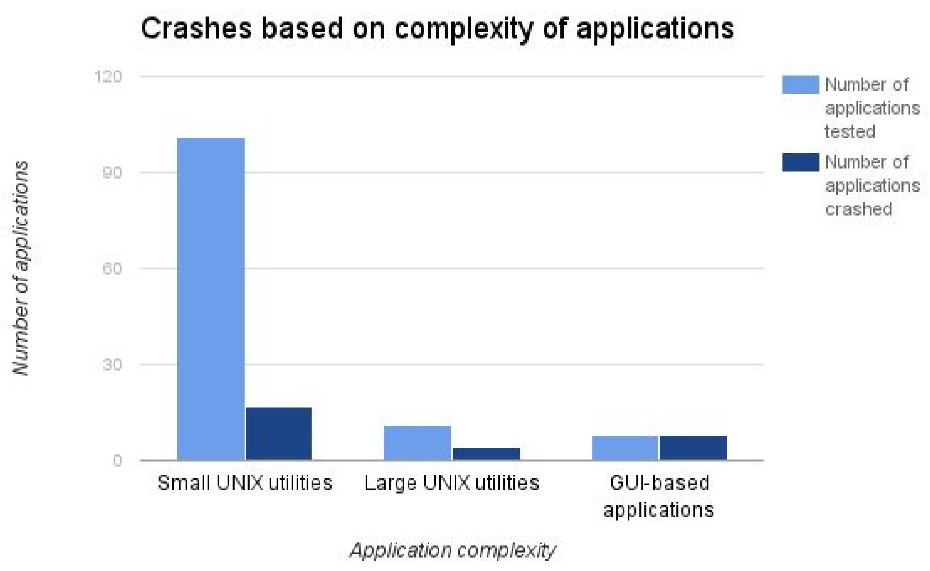

Figure 3 : Crashes based on complexity of applications

Figure 3 shows the classification of crashes based on the complexity of applications chosen. We observed that 17% (17 out of 101) of smaller utilities, 36% (4 out of 11) of larger utilities, and 100% (8 out of 8) of GUI-based applications crashed. We witnessed that most of the smaller UNIX utilities are stable and are less vulnerable to crashes whereas the GUI-based applications crashed frequently. We hypothesize that frequent crashes in larger applications is due to a code base worked upon by multiple contributors.

6. Analysis

After analysing the stack traces, we categorized the failures into four major types: Segmentation fault, Errors in the shared libraries, Deadlocks, and UI failures and trap/breakpoint trap.

We have evaluated and summarized the reason for each failure that were listed in Table 2 in Table 3.

Program name	Segmentation Fault	Errors in shared library	Deadlock	UI failures and trap/breakpoint trap
netstat	✔			
ps	✔			
lsmod	✔			
gdb	✔			
cancel	✔			

cpan	✔			
corelist	✔			
dirsplit	✔			
ifconfig	✔			
vi		✔		
vim		✔	✔	
telnet	✔			
gedit		✔	✔	✔
sort	✔			
wish		✔	✔	
hostnamectl	✔			
host	✔			
ghostscript	✔			
evince		✔	✔	✔
dm-tool				✔
eog		✔	✔	✔
dig	✔			
checkbox-gui		✔	✔	
loginctl	✔			
firefox				✔
bitmap				✔
calendar		✔	✔	
cheese		✔	✔	✔
red		✔	✔	
less		✔	✔	

Table 3 : Program crashes categorized based on failure type

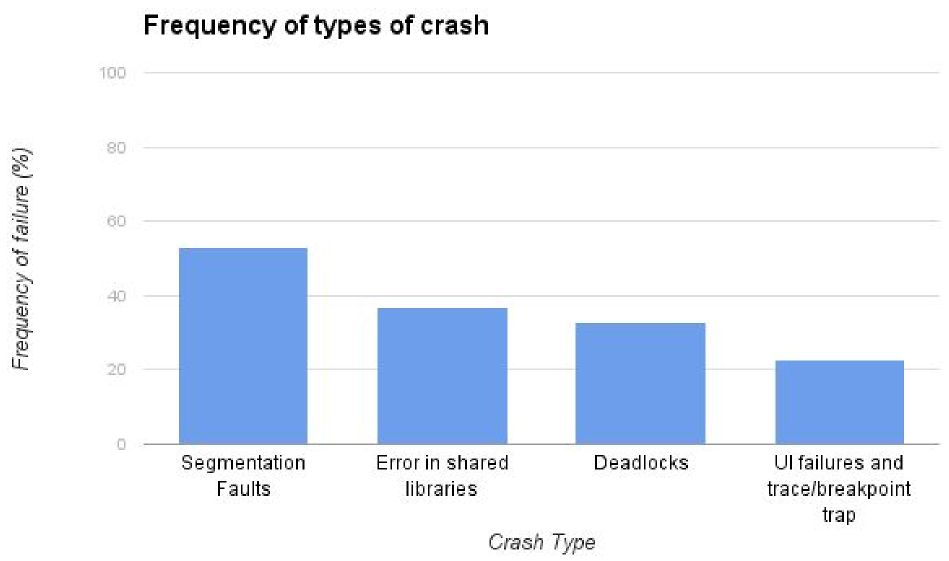

Figure 4: Frequency of types of crash

Figure 4 gives a plot of the failure percentage of each type of crash. We tested 120 programs, out of which, 30 crashed for various reasons. Among the crashed programs, 53% of programs failed due to segmentation fault, 36% failed due to bugs in shared libraries used by the program, 33% of the failures were caused by deadlocks and 23% failed due to User Interface (UI) issues and breakpoint trap errors.

We have presented the investigation of each of the categories with examples.

6.1 Segmentation faults due to errors in the program

Segmentation fault is the major reason for crashes in programs. When the memory allocation calls such as malloc, calloc, and realloc failed to allocate the promised memory, we observed crashes with segmentation fault. Dereferencing a null pointer can lead to such failures. One such instance is presented in the commonly used UNIX utility ps (sortformat.c : line 62)

```
format_node *thisnode;
thisnode = malloc(sizeof(format_node));
...
thisnode->width  =  w1 // dereferencing a null pointer without checking the return value.
```

When malloc fails, *thisnode* is assigned NULL value and dereferencing this leads to access to an illegal memory location. The stack trace and point of failure are provided in Appendix A. We observe instances of such code in many stable unix utilities. In vim, one of the most popular editors used in the UNIX world, among the twenty one times that malloc has been called explicitly, its return value has not been checked for eleven times. The filename and line number where segmentation fault occurs are given in Appendix B.

6.2 Errors in the shared libraries

Shared libraries are used extensively in all programs and often programmers assume the APIs provided to them work without failures. During the evaluation of results, we discovered that in many cases, the programs themselves abide by the good coding practices whereas the shared libraries used by them fail, leading to failures in the applications.

During our experimental run, it was observed that a low level pixel manipulation library *libpixman* is used in many GUI based applications such as firefox, cheese and across different platforms. This library contains a constructor *pixman_implementation_create* that allocates memory using malloc and returns an object of type *_pixman_implementation_t* (pixman_implementation.c : line 38). If malloc fails in our wrapper function, the constructor returns NULL without checking for the return value. This leads to failure of the library at multiple places wherever the constructor is called. (pixman_fast_path.c, pixman_combine_32.c, pixman_vmx.c). The unchecked NULL value lead to crashes in the programs that utilize the shared library *libpixman* (e.g. vim crashes at pixman_fast_path.c and pixman_combine_32.c (Appendix C)).

6.3 Deadlocks

Deadlock occurs in a system when a process or a thread requests for a system resource and stays in the waiting state indefinitely. We observed that a set of programs hanged when return values of calloc and accept were modified. To dig deeper into this issue we recompiled the program - evince, a PDF viewer. The stack trace generated confirmed that the application was waiting for a lock to be acquired and indicated that this issue was happening in glibc. We compiled a few different programs such as vim, gedit, and checkbox-gui to observe a similar stack trace. We continued the investigation by installing the debug symbols package for glibc to obtain a clear stack trace with the required details. The investigation got more interesting when we discovered that at the point where calloc fails, the return value was being checked in an attempt to report an error. However, when the error handling routine tried to acquire a lock that was previously held, it resulted in a deadlock. Below we show the code segment where the failure in calloc lead to deadlocks in many applications.

The following code path was found in glibc (gmem.c, function : g_malloc0 , point of failure : gslice.c, function : thread_memory_from_self()).

```
…
g_mutex_lock(&init_mutex)
g_slice_init_nomessage();    // enters critical section of the code
```

```
g_mutex_unlock(&init_mutex);
…
}
// inside the critical section
mem = calloc(1, n_bytes) // This fails
if(!mem)
// error handling code
g_error(...) {
  ...
   g_mutex_lock(&init_mutex);  // deadlock occurs here
   ...
};    // The error handling code calls g_mutex_lock() on the same mutex object.Thus leading to deadlock
```

The thread acquires *init_mutex lock* and enters into the critical section. Inside the critical section, the variable *mem* is allocated memory using calloc. If calloc fails here, error handling code *g_error* gets executed. In *g_error*, the thread again tries to acquire the already held *init_mutex* lock, leading to a deadlock.

This issue was discovered in a shared library, which leads the issue to easily span across multiple applications. Hence, shared libraries are frequently the weak links to many applications.

6.4 Trace/breakpoint trap and UI failures

Trace/breakpoint trap failures category is the case where failures were not handled elegantly. Some of these undesirable behaviors where the user was not being notified of the reason for the failures include incorrect loading of UI with missing buttons, wrong background colors, and missing options from the toolbars. We believe that such failures are important and must be addressed properly, though they do not completely crash the programs. These failures clearly indicate the negligence in handling errors in the code. The reason for incorrect loading of user interface is the continued execution of programs with failed return values leading to unexpected code paths. Incorrect loading of GUI applications was observed due to failures in open, read, write, socket system calls.

Another frequently observed failure was trace/breakpoint trap (core dumped). Trace/breakpoint trap leads to crashes as the executing program was explicitly sending a SIGTRAP signal to the process. The SIGTRAP signal is sent to a process when an exception (or trap) occurs: a condition that a debugger has requested to be informed of — for example, when a particular function is executed, or when a particular variable changes value. In the cases we investigated, trace/breakpoint trap was observed where the program crashed by raising a SIGTRAP signal internally because it was not able to open/read a file. Looking closely at one such failure, we found that when trying to open a schema settings file (/usr/share/glib2.0/schemas : glibc) in a shared library, the failure to find this file was handled by explicitly raising a G_BREAKPOINT that raised a SIGTRAP signal. (gmessages.c in glibc). These errors can be handled more elegantly by informing the user of the failure of the system call or inability to open/read a file rather than throwing an abrupt trap signal.

7. Future work

We have tested the basic versions of UNIX utilities without any options set. For example, sort utility has many options, some of which include -d for dictionary order, and -f to ignore cases. Our work can be extended to do fuzz testing on these utilities with various options set and also on different operating systems. It will ensure the analysis of portions of the program that were not tested by us or the code chunk that was not executed in a particular environment. The second direction to extend our work is to do return value checking on web applications source code. Most of the current web applications run JavaScript for client server communication. It will be interesting to observe how web applications behave when JavaScript function calls are failed intentionally with bad return values. As previously stated, our testing does not cover a complete set of applications. Several popular applications are open-sourced and testing these applications may be useful to identify any undesirable crash cases in them, some of which, potentially could even lead to detecting security bugs.

8. Conclusion

First, we found library interposition to be an extremely useful technique for our project. We were able to generate detailed information and logs for the various system calls without much difficulty. At the start of this project, we were unsure if we would be able to identify a good number of crash cases. However, we were successful in crashing and analyzing 25% of the programs that we tested. Most of the programs that crashed were due to not checking the return value of memory allocation libraries (malloc, calloc, realloc), ultimately leading to segmentation fault. As previously identified by Miller et. al in [9], return value checking of function calls is trivial, and hence most programmers tend to ignore it without realizing the potential issues associated with it. To avoid these issues, we recommend the use of xtmalloc, xtcalloc, and xtrealloc functions. These functions are provided by UNIX and they work exactly the same way as their respective memory allocation functions. Additionally, they also test for return values and call XtErrorMsg when the function call fails. For other issues, the programmers must exercise care while writing the program. We were not able to find an alternative solution for those issues. Based on our experience, we believe Fuzz Testing is a promising technique for automated testing and has a great potential to be used in many applications to identify bugs.

References

[1] Benjamin A. Kuperman and Eugene Spafford, “Generation of Application Level Audit Data via Library Interposition”, Technical Report CERIAS TR 99-11, COAST Laboratory, Purdue University (October 1999).

[2] J. Cargille and Barton P. Miller, “Binary Wrapping: A Technique for Instrumenting Object Code”, ACM Sigplan Notices, 27(6) pp. 17–18 (June 1992).

[3] Michael B. Jones, “Interposition Agents: Transparently Interposing User Code at the System Interface”, In Proceedings of the 14th ACM Symposium on Operating Systems Principles (December 1993)

[4] Timothy W. Curry, “Profiling and Tracing Dynamic Library Usage via Interposition”, In Proceedings of the USENIX 1994 Summer Conference (June 1994).

[5] Marc Gonzalez, Albert Serra, Xavier Martorell , Jose Oliver , Eduard Ayguade , Jesús Labarta J., Nacho Navarro , “Applying Interposition Techniques for Performance Analysis of OpenMP Parallel Applications,” In Proceedings of 14th International Parallel and Distributed Processing Symposium, IEEE Computer Society, IPDPS 2000.

[6] Thomas Ball and James R. Larus, “Optimally Profiling and Tracing Programs,” Conference Record of the 19th Anniversary ACM Symposium Principles of Programming Languages, ACM Press, New York, 1992, pp. 59-70.

[7] David Molnar, Xue Cong Li, David A. Wagner, “Dynamic test generation to find integer bugs in x86 binary linux programs”, In Proceedings of the 18th conference on USENIX security symposium (2009), SSYM’09.

[8] Barton P. Miller, Lars Fredriksen, and Bryan So, “An Empirical Study of the Reliability of UNIX Utilities”, Communications of the ACM 33(12) pp. 32-44 (December 1990).

[9] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl, “Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services”, Technical Report CSTR-95-1268, University of Wisconsin (April 1995).

Appendix A

Stack trace evaluation
Program name : ps
Recompiled with debug symbols - yes
Error type : Segmentation fault
Function : malloc
Point of failure : sortformat.c
Stack Trace :

```
#0  do_one_spec (spec=spec@entry=0x40b7f1 "time", override=override@entry=0x0) at sortformat.c:62      //
Malloc returns NULL and is dereferenced.
#1  0x000000000040b312 in generate_sysv_list () at sortformat.c:671
#2  process_sf_options (localbroken=<optimised out>) at sortformat.c:875
#3  0x0000000000408fa4 in arg_parse (argc=argc@entry=1, argv=argv@entry=0x7ffd14d2c5f8) at
parser.c:1197
#4  0x00000000004023f0 in main (argc=1, argv=0x7ffd14d2c5f8) at display.c:580
```

Appendix B

Code inspection

Program name : vim, version 7.4

Recompiled with debug symbols - yes

Error type : Segmentation fault

Function: malloc

Points of failure : gui_mac.c : line no 6878

Gvimext.cpp : line 93, line 963

if_sniff.c : line 475, 476

integration.c : line 904

os_unix.c : Line 2821, line 2827, line 2847

os_win32.c : Line 5979

workshop.c : Line 1771, line 1782

Appendix C

Program name : vim

Recompiled with debug symbols - yes, installed debug symbols for libpixman

Error type : Segmentation fault

Function : malloc

Point of failure : Shared library libpixman 1.0.32.so

```
Stack Trace : #0  _pixman_setup_combiner_functions_32 (imp=imp@entry=0x0) at
../../pixman/pixman-combine32.c:2387
#1  0x00007ffff175c12a in _pixman_implementation_create_general () at ../../pixman/pixman-general.c:219
#2  0x00007ffff175dbc6 in _pixman_choose_implementation () at ../../pixman/pixman-implementation.c:388
// Returns NULL pointer
#3  0x00007ffff1710509 in pixman_constructor () at ../../pixman/pixman.c:39
#4  0x00007ffff7dea10a in call_init (l=<optimised out>, argc=argc@entry=1, argv=argv@entry=0x7fffffffdfe8,
  env=env@entry=0x7fffffffdff8) at dl-init.c:78
```

```
#5  0x00007ffff7dea1f3 in call_init (env=<optimised out>, argv=<optimised out>, argc=<optimised out>,
l=<optimised out>)
  at dl-init.c:36
#6  _dl_init (main_map=0x7ffff7ffe1c8, argc=1, argv=0x7fffffffdfe8, env=0x7fffffffdff8) at dl-init.c:126
#7  0x00007ffff7ddb30a in _dl_start_user () from /lib64/ld-linux-x86-64.so.2
#8  0x0000000000000001 in ?? ()
#9  0x00007fffffffe342 in ?? ()
#10 0x0000000000000000 in ?? ()
```

// To our surprise, the stack trace shows '?' even after the program is compiled with debug symbols. We speculate that since the failure is in a dynamically linked library and the program crashes even before *main* gets executed, GDB is unable to find the symbols for the last three frames.