Fault Injection using PTRACE

Patrick Harvey, Do Kwon, Sunkyu Lim, Kevin McKenzie, Ian Walsh
{p1harvey, dokwon, limsk, kmckenzi, iwalsh} @ stanford.edu
CS 240 Final Project, Spring 2015

Abstract
Unchecked return values in systems code can cause bizarre errors that are almost impossible to reproduce in a lab setting - they are often caused by rare or intermittent subsystem failures and manifest far away from their sources. We have built a fault injector using the Linux `ptrace` system call that can intercept arbitrary system calls and calls to library functions by a target program, abort them, and fake their return values with values of our choosing. Using this tool, we can systematically check correct error handling in the target by faulting only certain calls per run and observing how the target reacts. We have used this tool to scan for errors in common Linux applications. Our results indicate that even commonly used applications including Firefox and Vim fail to gracefully handle uncommon errors in system calls and library functions.

1.0 Introduction
Systems programming relies heavily on system calls and common library routines in "unsafe" languages such as C and C++. Unfortunately, many of these system calls and routines have uncommon error conditions such as `ENOMEM` for "no memory available" and `EINTR` for "call interrupted by signal handler". Programs should verify such return values to gracefully handle rare but possible events, but in practice almost none actually do! In this paper, we discuss a fault injector we have built using the Linux utility `ptrace`. The tool allows us to inject uncommon failures into target programs to verify that these systems handle the errors gracefully and remain uncorrupted, or to observe exactly where and how a crash occurs if they do not. In combination with the target source code, automatic fault injection is a powerful technique for finding bugs caused by improper error handling.

2.0 Fault Injection and Methodology
`ptrace`[1] is a powerful and complex system call that allows one process, the "tracer", to monitor and modify another running process, the "tracee". `ptrace` lets the tracer read and write arbitrary words of the child's memory, as well as intercept important events in the tracee such as system calls, `fork()`s, and `execvp()`s. Entire debuggers[2] can be built using this one system call! We use it to intercept system calls and libc library routines in the tracee, abort their execution, and fake their return values. This allows us to determine how applications respond to valid but uncommon error conditions, such as `malloc`() returning `NULL` to signify out-of-memory.

On startup, our injector receives a list of one or more system call numbers to intercept (e.g. '2' for `open`) or a list of one or more libc function names (e.g. '`malloc`'), along with the desired return values and the target name to trace. It `fork()`s into a tracee which `exec()`s the target executable and a tracer which begins monitoring it using `ptrace`. An argument flag "fail_on_entry" controls whether we allow the intercepted call to actually complete or abort it prematurely; this lets us control whether the call's side effects really occur.

We do not know *where* unchecked return values may cause errors in the code, so we need a way to restrict which calls we actually intercept and fault. Another argument to the injector specifies *N*, the number of calls to skip before injecting faults. By scripting our injector to run with increasing values of *N*, we can methodically fault only the first, then the second instance of a particular call, and so on.

Several other flags control the behavior of the injector. Many real applications `fork()` several child processes to parallelize subtasks or respond to client requests, so a "`follow_clones`" flag controls

whether or not the injector traces child processes when they are created, or stays tracing the parent. The "`fail_on_entry`" controls whether a syscall should be failed at syscall entry, or if the syscall should succeed but return an error message anyway. In addition, interesting failure cases for file systems occur when writes to regular files succeed, but writes to directories fail. The "`only_dirs`" flag allows us to replicate this behavior with our injector. Finally, the "`after_main`" flag is used to indicate that function calls failures should only be injected after the `main`() function of the program has been called. This ensures that the problems we find are in the target program and not in the program loading operations.

2.1 Injecting Syscall Errors

Table 1 below details the system calls where we injected faults. We chose error values that were reasonably possible rather than error values that make little sense in context. For example, a `write` syscall wouldn't likely return `EBADF` for a file that the program has already opened successfully. Our syscall injection makes use of the `PTRACE_SYSCALL` option, as demonstrated in [3].

Syscall #	Name	Error Value	Description
0	`read`	`EIO` `EINTR`	I/O Error Call interrupted by signal handler

1	`write`	`EAGAIN`
		`EIO`
		`EINTR`
			File is locked, or too much memory is locked
			I/O Error
			Call interrupted by signal handler

2	`open`	`EACCESS`
		`EINTR`
		`ENFILE`
		`ENOMEM`
		`ENOSPC`
			Requested access not allowed
			Call interrupted by signal handler
			System limit on number of files was hit
			Insufficient kernel memory available
			No room for new file to be created

9	`mmap`	`ENOMEM`
		`ENODEV`
		`EPERM`
		`ENFILE`
		`EAGAIN`
			Insufficient kernel memory available
			File system doesn't support memory mapping
			File seal prevented operation
			System limit on number of open files is hit
			File is locked, or too much memory is locked

11	`munmap`	–	Same error modes as mmap
12	`brk`	`ENOMEM`	Insufficient kernel memory available
74	`fsync`	`EIO`	Error occurred during disk synchronization

Table 1: System call error modes tested with our injector tool

2.2 Injecting Library Call Errors

Table 2 below details the library calls where we injected faults. As before, we chose error values that were reasonably possible rather than error values that make little sense in context. Our function call injection technique relies on a custom breakpoint implementation, adapted from [4].

Function	Library	Error Value	Description
`malloc`	stdlib.h	NULL	Error occurred during memory allocation

Table 2: Library call error modes tested with our injector tool

3.0 Tested Programs

Our goal was to test our injector on applications that are commonly used. We split our target applications into three groups: enterprise applications, consumer applications, and common utilities. In each of the following applications, we faulted common syscalls and library calls specified in sections 2.1 and 2.2.

Enterprise Applications	Consumer Applications	Common Utilities
Apache HTTP Server	Firefox	Vim
Python (2.7.6)	Fuse	Emacs

Table 3: Applications tested using the injector tool

4.0 Results

Apache HTTP Server

Apache HTTP Server was difficult to run within the injector effectively – the manner in which the standard setup scripts spawned the server itself as a detached process was difficult for the injector to follow correctly. After some searching and trial-and-error, it was determined to be possible to run the server process directly within the injector, but even in this case the server's spinning off of other subprocesses made fault injection not work properly. Although a number of relatively gracefully logged errors could be found from fault injections on server startup, attempting to continue to inject faults when serving requests resulted in the server hanging indefinitely while attempting to serve any request; some testing revealed this occurred when running the server within the injector whether or not any faults were actually injected if we attempted to follow subprocesses. Thus, we were not able to obtain any particularly interesting results from attempts at fault injection on Apache HTTP Server.

Python

The Python interpreter had some strange behavior when memory management system calls (`mmap`, `brk`) were failed with out-of-memory codes (`ENOMEM`) once the interpreter had started, specifically when attempting to use the interpreter's help utilities. Entering interactive help and attempting to display the list of modules resulted in a segmentation fault and attempting to call non-interactive help with arguments did as well; entering interactive help and then exiting it resulted in a 'no mem for new parser' message and an infinite recurring sequence of `MemoryError` tracebacks. Furthermore, any operation that would result in a backtrace also appears to cause a segmentation fault if memory is unavailable after the error occurs.

Firefox

Firefox handled our fault injector very well. For the most part, it was able to fail gracefully with most of the tested syscalls and functions. We were not able to find any security concerns, but we did find an uncaught exception in the file *DownloadHistoryObserver.jsm*. When a `write` syscall fails, the `DownloadHistoryObserver` constructor returns an exception that is never caught by the calling function. After discovering the bug, we filed report number 1172292 on Bugzilla.

Fuse

Fuse is an application that allows a user to mount a filesystem in userspace. We used the injector on the code from our basic NFS filesystem running on Fuse implemented as part of the second programming assignment for CS240. Using the injector on the client and intercepting write syscalls produced an error in Fuse in which the client immediately unmounted all mounted Fuse filesystems. Furthermore, any attempts to remount the filesystem resulted in an infinite loop. We continue to investigate the cause and severity of the bug.

Vim
Vim experienced segmentation faults when memory allocation system calls were failed after the program had begun its initialization but before its initialization had completed. Moreover, it was evident that Vim attempted to catch and handle the segfault signal, but failed to clean up the terminal state before exiting: if faults began to be injected after vim started to open its TUI but before it had finished initialization, the terminal would be left in a state where no input commands were visible and newlines were printed incorrectly, or in some cases would become completely unresponsive.

Emacs
Under similar tests to Vim, Emacs also behaved somewhat erratically; its TUI would load very sluggishly and erroneously claim that the user has no home directory. However, it didn't seem to exhibit errors as fatal as those in Vim.

5.0 Future Enhancements

The fault injector we designed is now fully functional, but there are several system enhancements that can be implemented. First, the tool currently cannot handle both syscalls and library functions in arbitrary combinations. One potential use-case would be to fault syscalls 9 (`mmap`) and 12 (`munmap`) along with the library call `malloc`. In this way, we could rest assured that most memory allocators, including custom allocators, would fail to allocate memory. As of this paper's writing, the injector can fault system calls and library functions, but not both in the same invocation.

Furthermore, we would like to improve the calling interface to simplify the process of invoking our injector. Our interface currently mandates a calling convention that uses explicit flags for each option. A user must pass in a 0 for each option to *not* invoke and a 1 for each option *to* invoke. A more flexible interface using traditional Unix tool flags would make our injector easier to use.

Finally, our library call tracing requires target applications to be statically linked and compiled. Our injector uses the utility `nm` to find the function names in the symbol table, then uses `objdump` to find all call sites of the target function(s), and then issues breakpoints at those call sites. This tactic only works for statically linked target applications, where the addresses are all resolved at compile-time. As a future enhancement, we will attempt to identify the target function calls and break before the call goes to a linked library.

6.0 Conclusion

Our goal is for our fault injector to become integral to the toolchain of application development, both for coding novices and for advanced professionals. Our tool could be used to programmatically identify the syscall and library function failure modes that must be guarded against. When any application becomes widely used, every syscall or library call will eventually fail with rare error codes, and an application should be developed to handle these failures gracefully.

Our fault injector has proven useful in identifying bugs in common Linux applications. As it is further improved, we hope to see it more widely used in both academic and professional software development.

References

[1] http://linux.die.net/man/2/ptrace
[2] http://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1
[3] http://mips42.altervista.org/ptrace.php
[4] http://mainisusuallyafunction.blogspot.ca/2011/01/implementing-breakpoints-on-x86-linux.html