Enforcing Murphy’s Law for Advance Identification of Runtime Failures

Zach Miller
Todd Tannenbaum
Ben Liblit
{zmiller, tannenba, liblit} @ cs.wisc.edu
USENIX ATC 2012
Murphy’s Law

“If anything can go wrong, it will”
Motivation

- Developers want to find and fix errors before they are discovered by end users
- But despite extensive in-house testing, buggy software still gets released
  - Test coverage not complete
  - Different program inputs
  - Different run-time environment
Motivation

- Developers want to find and fix errors before they are discovered by end users
- But despite extensive in-house testing, buggy software still gets released
  - Test coverage not complete
  - Different program inputs
  - Different run-time environment
- A program ultimately interacts with its environment via the kernel interface
**man page for write(2)**

**NAME**
write — write to a file descriptor

**SYNOPSIS**

```c
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

**DESCRIPTION**

`write()` writes up to `count` bytes to the file referenced by the file descriptor `fd` from the buffer starting at `buf`. POSIX requires that a `read()` which can be proved to occur after a `write()` has returned returns the new data. Note that not all file systems are POSIX conforming.

**RETURN VALUE**

On success, the number of bytes written are returned (zero indicates nothing was written). On error, -1 is returned, and `errno` is set appropriately. If `count` is zero and the file descriptor refers to a regular file, 0 may be returned, or an error could be detected. For a special file, the results are not portable.

**ERRORS**

`EAGAIN` Non-blocking I/O has been selected using `O_NONBLOCK` and the write would block.

`EBADFD` `fd` is not a valid file descriptor or is not open for writing.

`EFAULT` `buf` is outside your accessible address space.

`EFBIG` An attempt was made to write a file that exceeds the implementation-defined maximum file size or the process’ file size limit, or to write at a position past the maximum allowed offset.

`EINVAL` The call was interrupted by a signal before any data was written.

`EFAULT` `fd` is attached to an object which is unsuitable for writing; or the file was opened with the `O_DIRECT` flag, and either the address specified in `buf`, the value specified in `count`, or the current file offset is not suitably aligned.

`EIO` A low-level I/O error occurred while modifying the inode.

`ENOSPC` The device containing the file referred to by `fd` has no room for the data.

`EPipe` `fd` is connected to a pipe or socket whose reading end is closed. When this happens the writing process will also receive a `SIGPIPE` signal. (Thus, the write return value is seen only if the program catches, blocks or ignores this signal.) Other errors may occur, depending on the object connected to `fd`. 
How resilient is your program in the face of unusual conditions?
Additionally, how do you test your program handling, for example, errno ENOSPC?
  – Actually fill the disk? (causes problems for other users of the disk)
  – Loopback device? (needs root)
  – VM? (takes time to set up)
Motivation

Simulate these unusual but perfectly legitimate conditions

- Helps identify unknown problems
- Reproduce rare events for testing
Murphy: an interposition agent

- Interposition tool using ptrace debugger interface
- Trap all system calls, and optionally do some evil:
  - Simply returning a failure to the application
  - Modifying parameters on the way in (or out)
Murphy approach tradeoffs

- **Pros**
  - Language agnostic!
  - No source code required, no relinking
  - No root access required
  - Appropriate for interposition of an entire software stack

- **Cons**
  - Performance
  - Low-level view – not everything is a system call
First gremlin: readone

- When the application requests bytes from a file descriptor, rewrite the system call to ask for and return one byte at a time

```
read(0, &buf, 8)
```

```
read(0, &buf, 1)
```

I don't think so!
First gremlin: readone

- This lone gremlin was enough to expose our first bug
- We failed running /bin/true!
- When programs are dynamically linked, the first thing they actually do is load shared libraries (e.g. libc.so.6)
- glibc's dynamic loader cannot handle short reads of a shared library's ELF header
/* We successfully opened the file. Now verify it is a file we can use. */
__set_errno (0);

fbp->len = ___libc_read (fd, fbp->buf, sizeof (fbp->buf));

/* This is where the ELF header is loaded. */
assert (sizeof (fbp->buf) > sizeof (ElfW(Ehdr)));
ehdr = (ElfW(Ehdr) *) fbp->buf;

/* Now run the tests. */
if (__builtin_expect (fbp->len < (ssize_t) sizeof (ElfW(Ehdr)), 0))
{
  errval = errno;
  errstring = (errval == 0
               ? N_("file too short") : N_("cannot read file data");
  ...
  lose (errval, fd, name, NULL, NULL, errstring, NULL);
}
Working around glibc

- As a quick hack, we worked around this by adjusting gremlin activation percentage
- However, this is not really a viable solution
- We needed a better control mechanism
ClassAds and Constraints

• The **readone gremlin constraint** is now evaluated with rich context info

Constraint =
!regexp("(\.[0-9]+)*$",Name,’’);

• The above constraint leaves shared library reads unmolested

```c
[
  Gremlin = "readone";
  SyscallCount = 20;
  InvokedCount = 1;
  EvilCount = 0;
  Pid = 24068;
  ChildNum = 1;
  Meta = "";
  SyscallNum = 0;
  SyscallName = "read";
  FD = 3;
  Name = "/lib64/libc.so.6";
  Length = 832;
]
```
Gremlins get more sophisticated

• More state was needed
  – Mapping FDs to filenames
  – Tracking working directory

• A single gremlin might trap 41 different system calls! (EINTR)

• Better performance through clever tricks (e.g. writeone_s)
Evil Gremlins could simulate...

- Disk full, pipe buffer full, removable media offline, ...
  - `write(fd, buf, count) < count`
- All data not available right now
  - `read(fd, buf, count) < count`
- System is very loaded
  - System call latency, fork() fails, pids rapidly reused
- NFS server dies while a file is open read/write
  - `close()` fails, last write() may be lost
- NTP runs, corrects clock drift
  - `time()` goes backwards!
- And many more…
So much evil!

- After running experiments with several different gremlins enabled, pretty much everything broke in some way. So now what do we do?
Murphy Control Functions

• Add the ability to dynamically enable and disable gremlins at run time to narrow the scope of Murphy’s actions (instead of affecting the entire program)

• Add a mechanism to pass metadata from the application to Murphy to track locations, function entry, variables, etc. in the high-level source code
• Originally used ioctl(), but found it was not available in some managed languages

• Settled on using mkdir() instead, which is widely supported

mkdir("/Murphy/set-metada/Line 67, enabling EAGAIN gremlin");
mkdir("/Murphy/update-config/eagain:10");
So, are we done?

- We now have a tool that can expose all sorts of problems in a highly configurable manner, and produce a log file of the evil acts committed.

- Manual inspection of code can usually help you identify the problem, along with some manual instrumenting of the code with Murphy control commands.
Good Enough!
Gremlin activation log

- Murphy can log an event whenever a Gremlin does evil: “gremlin X in vpid Y did evil on opportunity Z”
- Murphy can then replay a log during a new program execution

```plaintext
#GremlinName EvilCount Vpid InvokedCount SyscallCount
cwdlongpath 0 1 0 39 Meta: readless 0 1 2 61 Meta: readless 1 1 3 62 Meta: writeone_s 0 1 0 67 Meta: writeone_s 1 1 1 68 Meta:
```

...
Delta Debugging

- Automates the scientific method:
  - Set up a hypothesis
  - Test it
  - Reject it or refine it
- Used to minimize failure-inducing input
- In this case, the input is the replay log produced by executing Murphy the first time
- Minimizing this produces the smallest set of gremlin activity that causes the same problem
Delta Debugging

Possible failure causes

Set up first hypothesis

Test first hypothesis

Second hypothesis

Third hypothesis

Fourth hypothesis
When you have a minimal log, Murphy can then

- Leave process suspended for inspection
- Jump straight into a debugger
Results

- We tested several pieces of software that are in heavy use:
  - perl
  - python
  - openssl
  - bash
  - gcc
  - binutils
Results

• We tested several pieces of software that are in heavy use:
  – perl
  – python
  – openssl
  – bash
  – gcc
  – binutils
Bugs Found!

- We actually found bugs in programs not written in C. Besides the perl interpreter itself, we also discovered bugs in the perl regression tests, which are written in perl.
- Likewise, we found problems in the python tests (which are of course written in python).
- If openssl cannot read as much as it requests from /dev/urandom, it fails.
**Bugs Found!**

- All of these bugs were discovered by first enabling gremlins for the entire run, identifying a problem, and pinpointing it using the delta debugging approach just described.
- Using fully automated delta debugging on one failed perl test took under 10 minutes to go from 140,000 system calls down to just the one that matters. Highly effective!
Performance Overhead

- Instrumenting via ptrace has overhead
- Reading and writing one byte at a time causes a significant amount of overhead
- Resulting execution times for running the openssl regression tests:

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Murphy</td>
<td>6 seconds</td>
</tr>
<tr>
<td>Murphy, no gremlins</td>
<td>34 seconds</td>
</tr>
<tr>
<td>readone and writeone</td>
<td>325 seconds</td>
</tr>
</tbody>
</table>
General observations

- Checking return values of system calls is a **must** for correct code, yet is generally lacking.
- Almost all software fails to properly handle EINTR and EAGAIN gremlins, and as such we weren't able to use them to find any interesting failures.
Contributions

- A tool to help pro-actively discover bugs arising from legitimate but unexpected kernel responses.
- Itemized several such kernel responses.
- Found Bugs! Demonstrates useful approach.

- Observation: Murphy testing uncovered problems in widely-tested software – undoubtedly need is greater for new software.
Future Work

- Better/more gremlins
- Integration w/ strace tool
- Improve mechanisms to map from system level back to source code
- Survey work
- Why propagate these errors if nobody deals with them? Squash instead?
Try it out!

- http://research.cs.wisc.edu/murphy/
- There is a README included
- Instructions for building from source
  - Currently tested only for 64-bit Linux
- Documentation and sample configuration file