Task 5: Fault Injection into Glibc
Software Fault-Tolerance Lab

Systems Engineering Group
Dresden University of Technology
Dresden, Germany

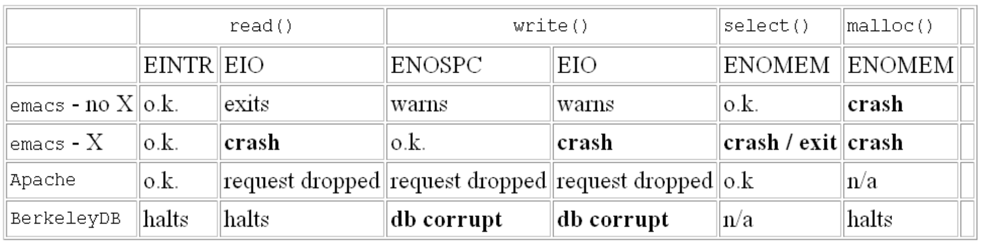
In this task we test how well applications handle errors returned by library calls. We use a similar approach as in Task 4.

Contents

1 Motivation

In a previous task we saw that applications have to use libraries which are not as robust as we need them to be. If a library cannot cope with special inputs, the application might crash.

This week we are going to deal with a related problem. Assuming valid inputs, libraries can still return errors, e.g., `malloc()` can return `NULL` and set `errno` to `ENOMEM` if not enough memory is available to fulfill the current request, `read()` can set `errno` to `EIO` if an error happen while reading the disk, etc. To behave robustly, applications have to handle such errors. Well known applications, however, do not do so and might not only



	read()		write()		select()	malloc()	
	EINTR	EIO	ENOSPC	EIO	ENOMEM	ENOMEM	
emacs - no X	o.k.	exits	warns	warns	o.k.	**crash**	
emacs - X	o.k.	**crash**	o.k.	**crash**	**crash / exit**	**crash**	
Apache	o.k.	request dropped	request dropped	request dropped	o.k	n/a	
BerkeleyDB	halts	halts	**db corrupt**	**db corrupt**	n/a	halts	

Figure 1: Applications and their error handling

crash, but also corrupt persistent state (e.g. files). Figure 1 shows some examples (source: `http://www.cs.berkeley.edu/~nks/fig/paper/fig.html`).

Again we will implement a wrapper, but instead of filtering bad inputs, we will inject faults to test application's error handling. These faults simulate errors in the library layer. We are going to focus on a single application, namely `tar`, and additionally implement a test automation framework. The approach can easily be extended for other applications.

2 Your Task

You shall build fault injector wrappers for glibc functions. Additionally, build a test framework to analyze and summarize the results. A single application will be tested: `tar`. It is used to create and extract archives.

- `tar cf temp.tar directory` creates an archive of a directory and its files.
- `tar xf temp.tar` extracts the directory and files.

Read the man page for more details.

2.1 Fault Injectors

Here follows an example of how you could implement a fault injector for `read` returning an I/O error.

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <unistd.h>
#include <assert.h>
#include <errno.h>
typedef ssize_t (*read_type) (int, void*, size_t);
ssize_t read (int fd, void *buf, size_t count) {
    static read_type orig = 0;
    if (drand48 () < 0.1) {
        errno = EIO;
        return -1;
```

```c
    }
    if (0 == orig) {
        orig = (read_type)dlsym (RTLD_NEXT, "read");
        assert (orig && "original read function not found");
    }
    return orig (fd, buf, count);
}
```

Note that the error is only injected with a certain probability, 10% in this case. This allows different parts of the code to be tested.

You shall implement six wrappers in the `faultinjectors` directory of your checkout:

- `fi_read_EINTR.c` – `read` was terminated with a signal and returned `EINTR`
- `fi_read_EIO.c` – `read` returns I/O error (`EIO`)
- `fi_write_ENOSPC.c` – `write` returns no space left error (`ENOSPC`)
- `fi_write_EIO.c` – `write` returns I/O error (`EIO`)
- `fi_select_ENOMEM.c` – `select` returns no memory left error (`ENOMEM`)
- `fi_malloc_ENOMEM.c` – `malloc` returns no memory left error (`ENOMEM`)

Follow the name convention and use the `Makefile` in the `faultinjectors` directory to compile your fault injectors. Interfaces and additional information can be found in the man pages of the respective functions. Note that if there are multiple functions in the system that have the same name, you might need to call `man` with an additional parameter, i.e. `man 3 read`. See `man man` for more details on `man`.

2.2 Testing framework

It is the goal of the test framework to perform tests with the application given a test case. The framework should also interpret the results. Outputs are written to standard output in a predefined format. The framework should read a single argument from the command line as an input parameter:

```
$ ./your-fault-injection-tool read_EIO
```

The test framework can be implemented in any of the following languages: Python, Ruby, Perl, Tcl, or C. We recommend implementing it in Python or Ruby though. We will not accept solutions written in Bash or Java.

In a nutshell, your framework has to

1. remove old test files

2. create a directory structure and files to archive. We provide you with an archive containing directories and files in your checkout. You can call `tar xf content.tar` to extract them from the archive.
3. create a new archive (e.g. `temp.tar`) by calling tar **with** fault injection.
4. analyze the behavior and output a summary in a predefined format.

2.2.1 Using the fault injectors

To use the fault injectors you have to call `tar` with the environment variable `LD_PRELOAD` set to the desired wrapper library, e.g. `LD_PRELOAD = faultinjection/fi_read_EINTR.so`

To set the environment variable you can use the following methods/modules:

- Ruby:
  `ENV['LD_PRELOAD'] = 'faultinjection/fi_read_EINTR.so'`
- Perl:
  `$ENV{'LD_PRELOAD'} = 'faultinjection/fi_read_EINTR.so';`
- Python:

```python
import os
...
os.environ['LD_PRELOAD'] = 'faultinjection/fi_read_EINTR.so'
```

- C/C++:

```c
#include<stdlib.h>
...
setenv("LD_PRELOAD", "faultinjection/fi_read_EINTR.so", 1);
```

2.2.2 Calling `tar` within your framework

To run tar for all functions/errors you should call `tar cf temp.tar content/`. Inside the framework this can be done as follows:

- Ruby: `` `tar ...` ``, `system("tar ...")`
- Perl: `` `tar ...`; ``, `system("tar ...");`
- Python: `os.spawn*(P_WAIT, "/bin/tar", "cf", "temp.tar", ...)`
- C: `fork` + `exec`*.

2.2.3 Analyzing the behavior

There are two states that have to be analyzed: the process state when it terminated; and the file state. The output of each test should look like the following:

```
Injected: read_EIO
ProcessState: exited
TarState: corrupted
```

It is very important to keep this convention because our testing system uses it for parsing the results. `Injected` is the test case (the fault injection). The `ProcessState` is the state when the application exited. It can be:

- `signaled` – when the application crashed
- `timeout` – after a 5 seconds timeout (to be implemented by you)
- `exited` – if it exited with some error (return code $\neq 0$)
- `success` – exited with 0, meaning no error

`TarState` represents the state of the tar file after the test. There are 4 possible states:

- `no_tar` – there is no file
- `empty` – the file is empty
- `corrupted` – the file is corrupted (file exists but its content does not match `content.tar`)
- `okay` – the file is the same as the original

Your testing framework should use file system functions to compare the content of `temp.tar` to `content.tar`. Table 2.2.3 summarizes the constructs you can use in the different languages to find the state of the process and the tar file as discussed above.

Language	Process State	Tar State
Ruby	`$?` - ProcessStatus	module Timeout
Python	`spawn*`	class Timer
Perl	`$?`	alarm or fork and sleep
C	`waitpid`	alarm or fork and sleep

Table 1: Overview of how to query the status of another process and how to detect timeouts in different programming languages.

Note: when implementing the timeout, make sure you kill the process if the timeout expires.

Language	**Online Documentation**	**Command Line Example**
Ruby	`http://www.ruby-doc.org/`	`ri Kernel#system`
Perl	`http://www.perl.org/docs.html`	`perldoc -f system`
Python	`http://docs.python.org/library/`	`pydoc os.system`
C/C++	`http://www.gnu.org/software/libc/manual/html_node/index.html`	`man 3 system`

Table 2: Online and offline resources for Ruby, Perl, Python, and C/C++.

2.3 Putting it all together

After you implemented the fault injector wrappers and the testing framework, you should modify the file `run.sh` in your checkout. This file is a bash script which executes your framework testing `tar` for all possible tests. You should substitute the string `<your-script>` by the command line that calls your framework.

2.4 Further Documentation

If you do not know Perl, Ruby or Python, you should stick to C/C++ for writing the testing framework. However, if you know any of these three scripting languages, then we recommend that you use it. In our experience scripting languages are a particular good fit for such a framework. And, you might get results faster than with C/C++.

For all three languages you will find documentation in the Internet. Table 2 lists some online resources as well as examples for using the offline documentation on command line for all four languages.

3 Organizational Remarks – Deadline

Solve this week's task at home or in the lab. Please be aware that there is a deadline for this task. You can find the exact date published on our website. If you do not hand in your solution until the specified date, we will regard your solution as missing and you will not receive your certificate.