Using Valgrind for file descriptor tracking

Alexandra Hájková

February 01, 2025

What are we going to talk about?

- Valgrind error manager
- Actions to take on errors - integration with GDB
- File descriptors as resources
- Detecting and reporting bad file descriptor usage

Quick recap - What is valgrind?

- an instrumentation framework for building dynamic analysis tools
- detects various memory management and threading bugs
- decompiles, instruments and recompiles your code
- intercepts syscalls, signals, threading, auxv, /proc access

Tools

- Memcheck
  - most used tool, default
  - detects unaccessible or undefined memory usage
- Cachegrind - cache profiler
- Massif - heap profiler
- Helgrind - thread debugger
- None - did nothing, now does fds tracking

Valgrind Error Manager

Tools use it to

- create events (add backtraces)
- report events (that are errors)
  - human readable
  - machine readable (xml)
- suppress specific issues
- integration with other tools like GDB

GDB Trap on errors

```
 gdb ./bad
(gdb) set remote exec-file ./bad
(gdb) set sysroot /~
(gdb) target extended-remote | vgdb --multi --vargs -q
Remote debugging using | vgdb --multi --vargs -q
(gdb) start
Temporary breakpoint 1 at 0x4011d1: file bad.c, line 27.
Starting program: bad
relaying data between gdb and process 3457526

(gdb) c
Continuing.
==3457526== Conditional jump or move depends on uninitialised value(s)
==3457526==    at 0x4011F1: main (bad.c:30)
==3457526==

Program received signal SIGTRAP, Trace/breakpoint trap.
0x00000000004011f1 in main () at bad.c:30
30          if (s.flag1 || s.flag2)
```

An article

- Valgrind and GDB in close cooperation

  `https://www.redhat.com/en/blog/valgrind-and-gdb-close-cooperation`

file descriptors are like other resources

- can be created and destroyed
- can be used while not destroyed
- Valgrind already tracks all system calls (where these events occur)

file descriptors are like blocks of memory

- open/creat are like malloc/calloc
- read/write to/from memory block/file descriptor
- close is like free, must happen (only) once
- Valgrind needs to hide its own memory from the application
- Valgrind uses file descriptors itself, must adjust `RLIMIT_NOFILE`

`--track-fds=yes`

- file descriptor represents an (open) resource
  - open file, network connection, timer, signal, process, etc.
- file descriptors are either
  - inherited at program startup (stdin/stdout/stderr), found through `/proc/self/fds`
  - created by syscalls `creat`, `open`, `socket`, `accept`, `dup[23]`, ...
- destroyed by
  - syscall `close`
  - syscall `close_range`
- Record for all events
  - where (execution context)
  - name/file/socket description (if possible)

File descriptor double close

- record event where the file descriptor was originally created
- show error where the file descriptor was used/closed again

Example

```
==3521944== File descriptor 3: /dev/pts/0 is already closed
==3521944==    at 0x497F804: close (close.c:27)
==3521944==    by 0x401322: main (bad.c:51)
==3521944==  Previously closed
==3521944==    at 0x497F804: close (close.c:27)
==3521944==    by 0x4012CF: main (bad.c:44)
==3521944==  Originally opened
==3521944==    at 0x497FA4B: dup (syscall-template.S:120)
==3521944==    by 0x401208: main (bad.c:29)
```

File descriptor use after close

- record event where it was originally destroyed

Code

```
close(fd);
write(fd, string, 3);
```

Example

```
==3696196== File descriptor 3: /dev/pts/10 is already closed
==3696196==    at 0x498BF74: close (close.c:27)
==3696196==    by 0x401356: main (bad.c:54)
==3696196==  Previously closed
==3696196==    at 0x498BF74: close (close.c:27)
==3696196==    by 0x4012D7: main (bad.c:45)
==3696196==  Originally opened
==3696196==    at 0x498C1BB: dup (syscall-template.S:120)
==3696196==    by 0x401210: main (bad.c:30)
```

File descriptor bad usage

- program uses invalid file descriptor
  - too big or fd < 0

Code

```
write(12345, string, 3);
```

Command

```
./vg-in-place --track-fds=yes ./bad
```

Example: fd is insanely big

```
==3695625== Invalid file descriptor 12345
==3695625==    at 0x4991984: write (write.c:26)
==3695625==    by 0x4012ED: main (bad.c:46)
```

File descriptor was never created

- program uses a file descriptor it never created (or inherited)

Code

```
/* Never created fd 7. */
write(7, string, 4);
```

```
==714874== File descriptor 7 was never created
==714874==    at 0x497DE84: write (write.c:26)
==714874==    by 0x40114B: main (in /home/ahajkova/valgrind
```

New `--modify-fds=[no|high|strict]` option

- return highest available fd
- POSIX requires that new file descriptors are always the lowest possible ones
- will prevent bugs caused by that the POSIX behaviour
- `strict` mode: fds 0,1,2 would be exempt
  - when 0, 1 or 2 are "free" (unallocated) then they would be picked as new fd

```
int oldfd = open ("FOO.txt", O_RDWR|O_CREAT, S_IRUSR | S_IW
/*... do something with oldfd ...*/
close (oldfd);
/* Lets open another file... */
int newfd = open ("BAD.txt", O_RDWR|O_CREAT, S_IRUSR | S_IW
/* ... oops we are using the wrong fd (but same number...)
dprintf (oldfd, "some new text\n");
```

file descriptor leaks

- like memcheck memory leaks
- Do inherited stdin/out/err file descriptors count?
  - `--track-fds=all` vs `--track-fds=yes`

`--track-fds=yes` example

```
==3696499== FILE DESCRIPTORS: 4 open (3 std) at exit.
==3696499== Open file descriptor 4: /dev/pts/10
==3696499==  at 0x498C1BB: dup (syscall-template.S:120)
==3696499==  by 0x40121D: main (bad.c:31)
```

--track-fds=all example

```
==3696688== FILE DESCRIPTORS: 4 open (3 std) at exit.
==3696688== Open file descriptor 4: /dev/pts/10
==3696688==    at 0x498C1BB: dup (syscall-template.S:120)
==3696688==    by 0x40121D: main (bad.c:31)
==3696688==
==3696688== Open file descriptor 2: /dev/pts/10
==3696688==    <inherited from parent>
==3696688==
==3696688== Open file descriptor 1: /dev/pts/10
==3696688==    <inherited from parent>
==3696688==
==3696688== Open file descriptor 0: /dev/pts/10
==3696688==    <inherited from parent>
```

The curious case of `close_range()`

- a bit of a hammer
  - better than
    ```
    for (int i=3; i < 999999; i++) close (i);
    ```
- better to use `CLOEXEC` flag
  - the close-on-exec flag for the new file descriptor
  - essential in some multithreaded programs
- flag "double close" only if closing specific range

`close_range()`

- Introduced in Linux 5.9 (released 2020), glibc 2.34
- BSDs have `closefrom`, glibc implements that as a wrapper
  ```
  close_range (lowfd, ~0U, 0);
  ```

GDB inspecting file descriptors example

```
gdb -ex 'set remote exec-file ./bad' -ex 'set sysroot /' ./bad

(gdb) target extended-remote | vgdb --multi --vargs -q --track-fds=yes

(gdb) monitor v.info open_fds
==3698979== FILE DESCRIPTORS: 5 open (3 std) .
Open AF_UNIX socket 4: <unknown>
==3698979==    at 0x498C1BB: dup (syscall-template.S:120)
==3698979==    by 0x40121D: main (bad.c:31)
==3698979==
Open AF_UNIX socket 3: <unknown>
==3698979==    at 0x498C1BB: dup (syscall-template.S:120)
==3698979==    by 0x401210: main (bad.c:30)
```

Valgrind can act as a gdbserver

- `vgdb` intermediary between Valgrind and GDB
- `valgrind -q -vgdb-error=0 ./bad`
- `(gdb) target remote | vgdb -pid=3781640`

- How to debug memory errors with Valgrind and GDB
  - https://developers.redhat.com/articles/2021/11/

Work in progress

- `--track-fds=bad` work in progress
- when you are only interested about misusing fds
- do not warn about the fd leaks
- should it be on by the default?

Conclusion

- file descriptors is the resource somewhat similar to memory
- `--track-fds=yes` will warn you about misusing fds
- `--modify-fds` work in progress
  - non POSIX behaviour
- `--track-fds=bad` work in progress
- it is useful to use Valgrind together with GDB

Thank you for your attention!

- Questions?

My articles about Valgrind and GDB

- How to track file descriptors with Valgrind
  `https://developers.redhat.com/articles/2024/11/07/track-file-descriptors-valgrind`
- Valgrind and GDB in close cooperation `https://www.redhat.com/en/blog/valgrind-and-gdb-close-cooperation`
- 7 pro tips for using the GDB step command
  `https://opensource.com/article/22/12/gdb-step-command`
- How to debug memory errors with Valgrind and GDB
  `https://developers.redhat.com/articles/2021/11/01/debug-memory-errors-valgrind-and-gdb#`
- Using Valgrind's –trace-flags option
  `https://developers.redhat.com/articles/2021/06/15/debugging-valgrind-adding-fused-multiply-add-support-aa`
  `p=878147`