Reliably Reproducing Kernel Data Races

From userland with LTP Fuzzy Sync

FOSDEM and SUSE Engineering summit 2021

https://richiejp.github.io/fuzzy-sync-pres-2021/
What is a data race?

Informally and according to Richard Palethorpe.

- It is also called a race condition.
- It requires a computation which reads at least one variable from somewhere.
- The result(s) of the computation must change depending on the value of the variable.
- The value of the variable must change over time. Thus the result of the computation changes over
- Only static, purely functional code has no data races.

However...
Usually if someone talks about a "data race" or "race condition" they are talking about a bug caused by race.
What do kernel data races typically look like?

A gross and degenerate simplification.

- A block of code updates a memory pointer (Block A).
- Another block reads a memory pointer (Block B).
- The blocks may run concurrently.
- Block A should only run after/before B to ensure the pointer value is valid for B.
- The ordering of memory accesses has not been ensured in all scenarios.
- Block B blows up when it dereferences a dodgy pointer.

However...

- It is usually more complicated than that.
- A whole bunch of conditions have to be met for the value A writes to blow up B.
What is a reproducer?

And what is Fuzzy Sync for?

- A reproducer is a program which triggers a particular bug in another program.
- When a bug is fixed in the kernel, we can write an LTP test which reproduces it.
  - This validates the bug fix.
  - Ensures the bug is not reintroduced.
  - Ensures the fix is backported to older kernels.
  - Accidentally finds other bugs.
- A particular data race outcome may be difficult to reproduce.
- Fuzzy Sync helps reproduce bugs which require a particular race outcome.
A simple race to get us started

- How can `winner` be equal to 'A' and 'B'?
- Will `winner` ever be equal to 'A' when `...end_race_a` and `...end_race_b` are synchronised?
**Calculating Delay**

Spin Wait

- `start_a()`: winner='A'
- `winner='A'`
- `end_a()`

- `start_b()`: `nanosleep()`
- `winner='B'`
- `end_b()`

**Implementing Delay**

Delay A

- `start_a()`: winner='A'
- `winner='A'`
- `end_a/b`

- `start_b()`: `nanosleep()`
- `winner='B'`
- `end_a/b`

Time

- Window
Timing Plots

- **winner == 'A'** only once (red circle), when A is delayed by roughly 55000ns.
The LTP library implements `main` and many features.

We declare `struct tst_test test` and implement the test specific logic.

Has some similarities to popular testing frameworks.
sendmsg and setsockopt are system calls which act on a socket

They are both acting on the same socket (sockfd)

It is clear just from the fzsync calls that the test is racing sendmsg against setsockopt.

For some reason setting IP_HDRINCL to zero at the same time as sending a message is bad
do_ip_setsocket can set inet->hdrincl while raw_sendmsg executes.

We start with hdrincl = 1

It is possible to set hdrincl = 0 after branch 1, but before branch 2.

rfv will contain uninitialised stack data if branch 1 is not taken.

There could be other bugs as inet->hdrincl is accessed multiple times.
- Fuzzy Sync loops 8354 times until timing volatility reaches a lower threshold.
- It appears `sendmsg` takes far longer to execute than `setsocketopt`.
- Fuzzy Sync calculates a delay range which will overlap the syscalls in all possible ways.
- Shortly after we start adding random delays we quickly hit a KASAN splat.
- Stale stack data is passed to `ip_append_data` and eventually blows up `csum_and_copy_from_iter_full` which tries to dereference part of it.
sendmsg03 Wrap Up

- Most likely the initial timings are recorded with `hdrincl = 0` for all of `raw_sendmsg` because `setsockopt` is much faster. However this still results in a good delay range.
- Kernel bug assigned CVE-2017-17712
- Reproducer converted to LTP Fuzzy Sync by Martin Doucha
• Races `fchownat` against `dup2` on a crypto API socket.
• `dup2` has the side effect of closing the socket pointed to by `sock`.
• `fchownat` accesses the socket, or file, pointed to by `sock`.
• If `errno = ENOENT` is set by `fchownat`, then we hit the race window, but the kernel handled it.
Meanwhile in `net/socket.c`

- `_sock_release` (from `dup2`) frees `sock->sk`, but does not set it to `NULL`.
- While `sock->sk` is being freed `fchownat` may be waiting for the `inode` lock (or whatever).
- When `sockfs_setattr` (from `fchownat`) runs we get a `use-after-free` instead of `ENOENT`.
- Fix is to set `sock->sk = NULL` with `inode` lock held.
But there is another race

- Passes *quickly* on fixed x86 systems.
- On large ARM64 machines we occasionally get fails on fixed systems.
- `dup2` is "atomic", but...
- There is a window where `dup2` invalidates the socket's file descriptor, before re-pointing it to the target file descriptor.
- This causes `fchownat` to return *much quicker* with `EBADF`.
- If this happens consistently, our delay range for `fchownat` will be too short.
Delay bias

When we see **EBADF** we can add a constant delay to **dup2**.

This ensures **fchownat** has enough time to grab the socket from the file descriptor.

This then means **fchownat** will continue down a longer path.

Other tests with delay bias

- CVE-2016-7117
- setsockopt06
- setsockopt07

```c
if (TST_RET == -1 && TST_ERR == EBADF) {
    tst_fzsync_pair_add_bias(&fzsync_pair, 1);
    continue;
}
```
Wrapup af_alg07

• Is also a test of Fuzzy Sync’s reliability as we must hit a race window to pass.
• Discovered by Syzkaller
• LTP test written by Martin Doucha
• Delay bias added by Li Wang
• Specific fix by Mao Wenan
• General fix by Eric Biggers
• More general test(s) based on reproducer by Eric is/are possible.
• One day a kernel change will probably break the test, but sometimes we just have to live with that.
Why don't you just...

- Create many threads or processes
  - Works great for POCs, but...
  - Expensive
  - Terrible and unknown scaling properties
  - Like fishing with dynamite

- Use X
  - It works by instrumenting the code (it's invasive, requires `CAP_SYS_ADMIN` etc.)
  - We couldn't find X
  - It's usually easier to specifically rewrite something for the LTP anyway

- Add a random sleep
  - That is what Fuzzy Sync does, but we use a spin wait
  - Context switching often takes longer than the required sleep
  - Different systems require much different delay ranges.
Standalone edition

https://gitlab.com/Palethorpe/fuzzy-sync

- Just a single header file
- Only dependency is a compiler with atomic intrinsics
  - POSIX threading is used by default, but can be removed
- Can be easily copied into another project
- Contains example test using CMake/CTest
- **LTP version** is still under development, but is fairly stable now