https://richiejp.github.io/fuzzy-sync-pres-2021/

FOSDEM and SUSE Engineering summit 2021

Reliably Reproducing Kernel Data Races

From user land with
LTP Fuzzy Sync

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.

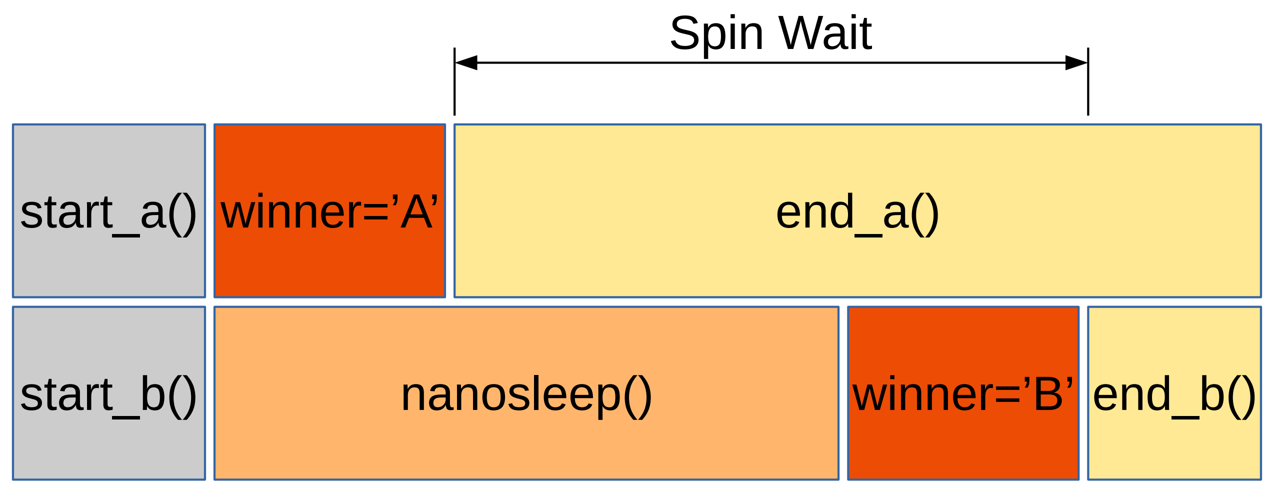
A simple race to get us started

```
// Thread A

while (fzsync_run_a(&pair)) {
    winner = 'A';

    fzsync_start_race_a(&pair);
    if (winner == 'A' && winner == 'B')
        winner = 'A';
    fzsync_end_race_a(&pair);
}
```

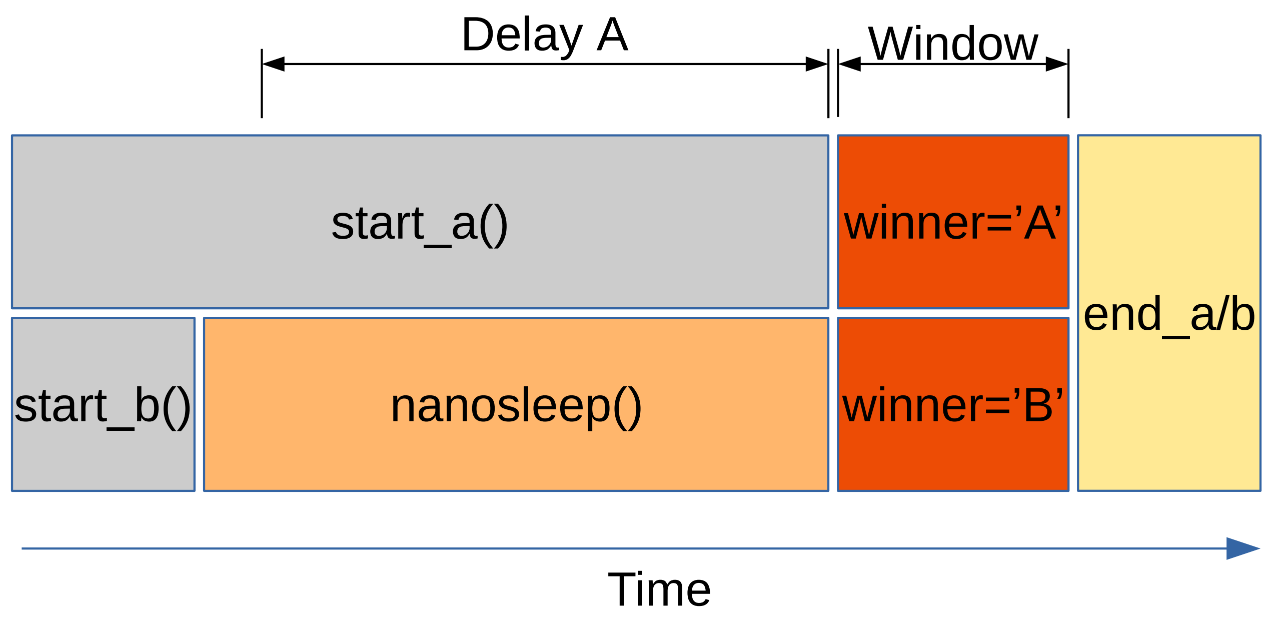
```
// Thread B

while (fzsync_run_b(&pair)) {


    fzsync_start_race_b(&pair);
    nanosleep(/* for 1ns */);
    winner = 'B';
    fzsync_end_race_b(&pair);
}
```

- 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?

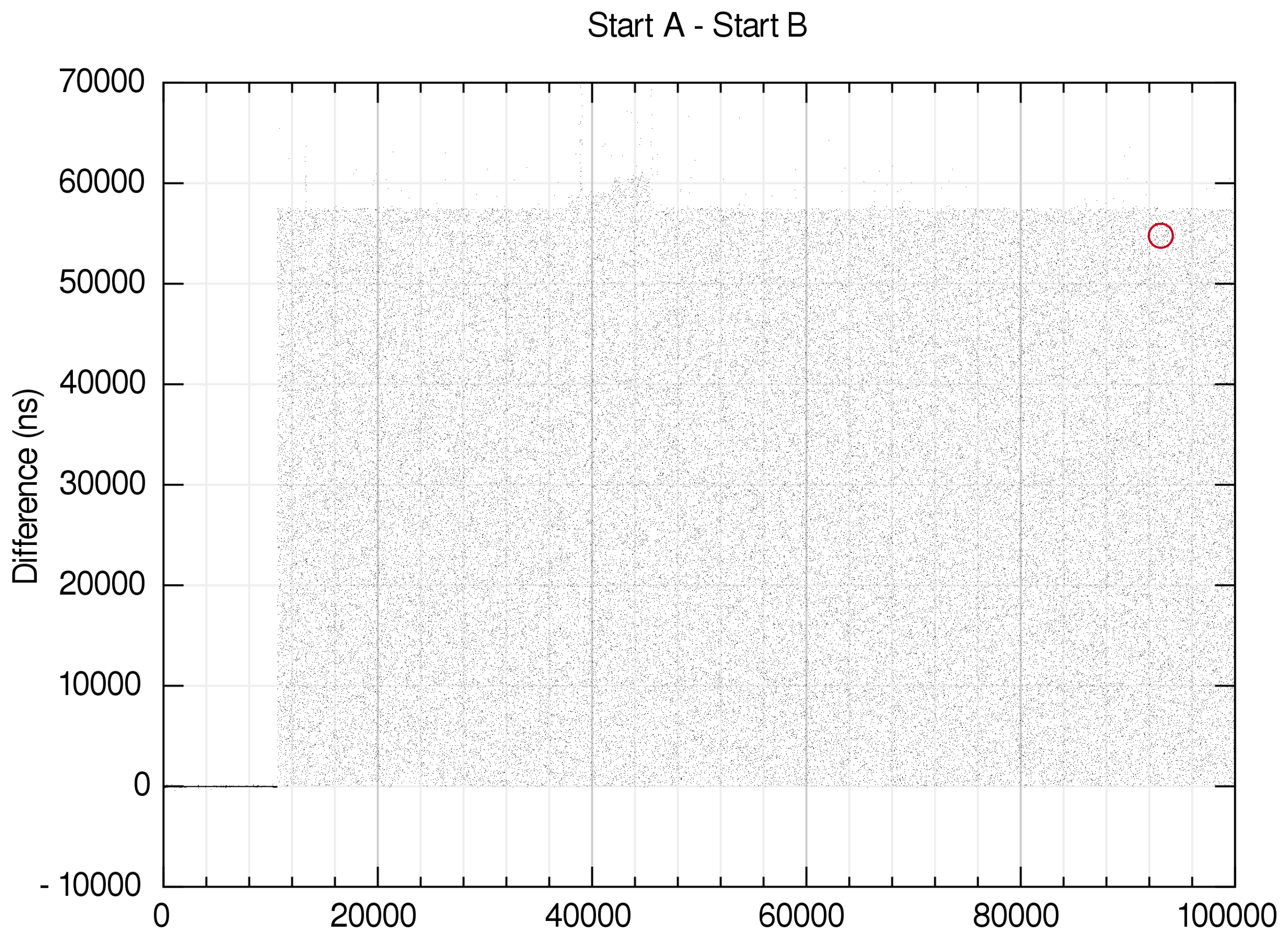
Calculating Delay

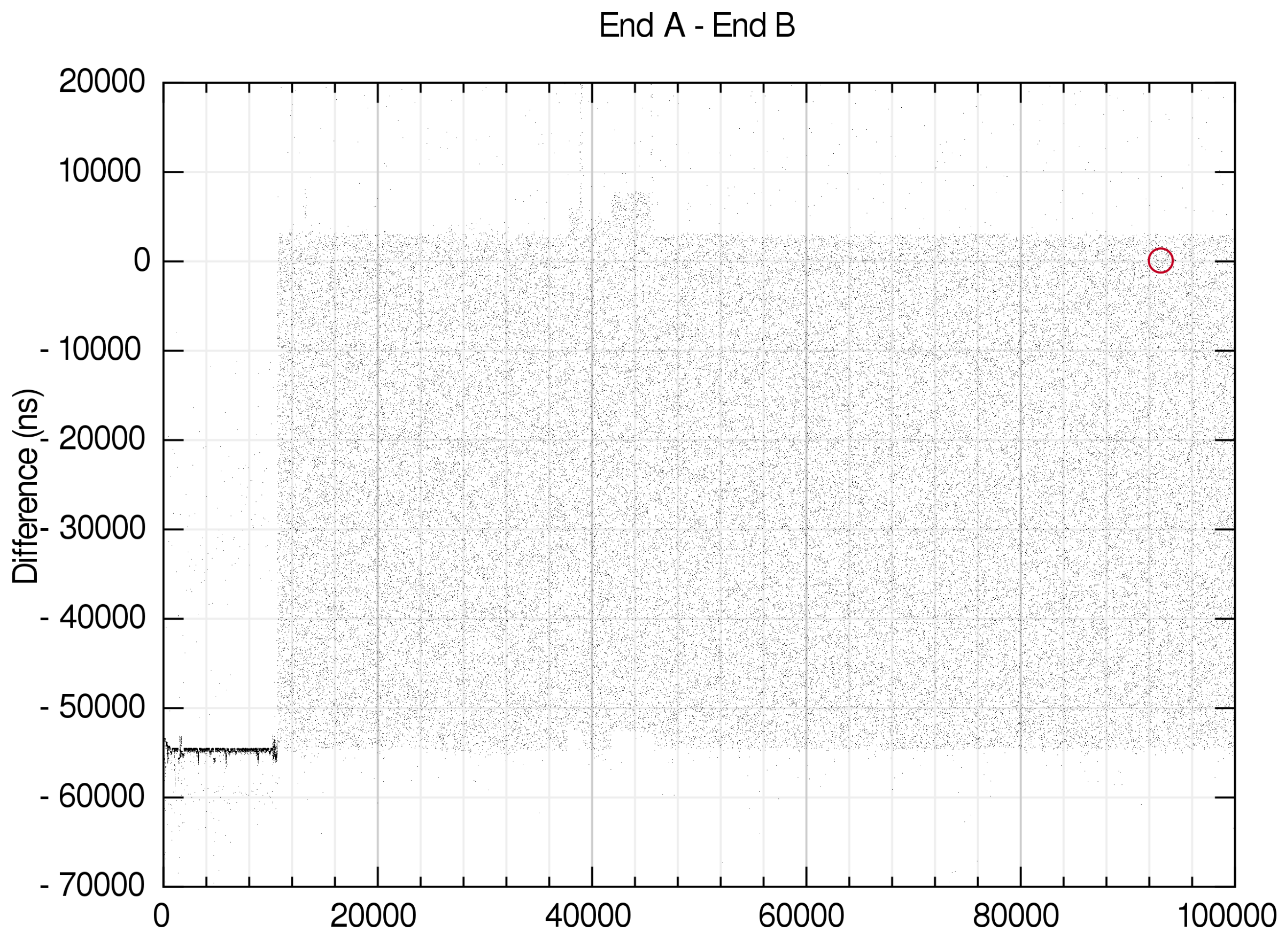


Implementing Delay



Timing Plots





- **`winner == 'A'`** only once (red circle), when **`A`** is delayed by roughly 55000ns.
- More about this at richiejp.com/a-rare-data-race.

sendmsg03 and LTP test anatomy

```
// SPDX-License-Identifier: GPL-2.0-or-later
...
#include "tst_test.h"
#include "tst_fuzzy_sync.h"
...
static struct tst_fzsync_pair fzsync_pair;

static void setup(void)
{
    ...
    fzsync_pair.exec_loops = 100000;
    tst_fzsync_pair_init(&fzsync_pair);
}

static void cleanup(void)
{
```

- 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

SENDMSG03

```c
// Thread A
int val = 1;
...
while (tst_fzsync_run_a(&fzsync_pair)) {
    SAFE_SETSOCKOPT_INT(sockfd, SOL_IP,
                        IP_HDRINCL, val);
    tst_fzsync_start_race_a(&fzsync_pair);
    sendmsg(sockfd, &msg, 0);

    tst_fzsync_end_race_a(&fzsync_pair);
    ...
}
```

```c
// Thread B
int val = 0;

while (tst_fzsync_run_b(&fzsync_pair)) {


    tst_fzsync_start_race_b(&fzsync_pair);
    setsockopt(sockfd, SOL_IP, IP_HDRINCL,
               &val, sizeof(val));
    tst_fzsync_end_race_b(&fzsync_pair);

}
```

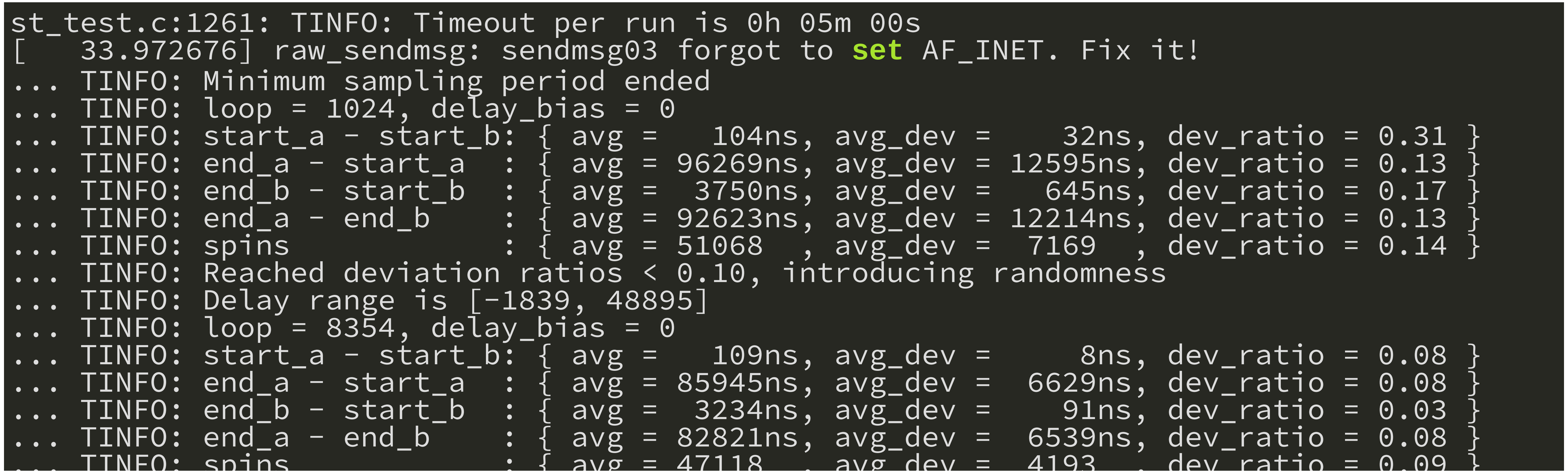
- **`sendmsg`** and **`setsockopt`** are *system calls* which act on a *socket*
- They are both acting on the same socket (**`sockfd`**)
- It is clear just from the **`fzsync`** calls that the test is racing **`sendmsg`** against **`setsockopt`**.
- For some reason setting **`IP_HDRINCL`** to zero at the same time as sending a message is bad

```
// Thread A (net/ipv4/raw.c)
static int raw_sendmsg(..) {
    ...
    if (!inet->hdrincl) { //Branch 1
        rfv.iov = msg->msg_iov;
        rfv.hlen = 0;
        err = raw_probe_proto_opt(&rfv, &fl4);
        ...

    if (!inet->hdrincl) { //Branch 2
        ...
        err = ip_append_data(..., &rfv, ...);
```

```
// Thread B (net/ipv4/ip_sockglue.c)
static int do_ip_setsockopt(...)
{
    ...
    case IP_HDRINCL:
        if (sk->sk_type != SOCK_RAW) {
            err = -ENOPROTOOPT;
            break;
        }
        inet->hdrincl = val ? 1 : 0;
        break;
    ...
```

- **`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.

SENDMSG03



```
st_test.c:1261: TINFO: Timeout per run is 0h 05m 00s
[   33.972676] raw_sendmsg: sendmsg03 forgot to set AF_INET. Fix it!
... TINFO: Minimum sampling period ended
... TINFO: loop = 1024, delay_bias = 0
... TINFO: start_a - start_b: { avg =   104ns, avg_dev =    32ns, dev_ratio = 0.31 }
... TINFO: end_a - start_a  : { avg = 96269ns, avg_dev = 12595ns, dev_ratio = 0.13 }
... TINFO: end_b - start_b  : { avg =  3750ns, avg_dev =   645ns, dev_ratio = 0.17 }
... TINFO: end_a - end_b    : { avg = 92623ns, avg_dev = 12214ns, dev_ratio = 0.13 }
... TINFO: spins            : { avg = 51068  , avg_dev =  7169  , dev_ratio = 0.14 }
... TINFO: Reached deviation ratios < 0.10, introducing randomness
... TINFO: Delay range is [-1839, 48895]
... TINFO: loop = 8354, delay_bias = 0
... TINFO: start_a - start_b: { avg =   109ns, avg_dev =     8ns, dev_ratio = 0.08 }
... TINFO: end_a - start_a  : { avg = 85945ns, avg_dev =  6629ns, dev_ratio = 0.08 }
... TINFO: end_b - start_b  : { avg =  3234ns, avg_dev =    91ns, dev_ratio = 0.03 }
... TINFO: end_a - end_b    : { avg = 82821ns, avg_dev =  6539ns, dev_ratio = 0.08 }
... TINFO: spins            : { avg = 47118  , avg_dev =  4193  , dev_ratio = 0.09 }
```

- 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
- Found, fixed and original POC by Mohamed Ghannam https://seclists.org/oss-sec/2017/q4/401
- Reproducer converted to LTP Fuzzy Sync by Martin Doucha

af_alg07 (CVE-2019-8912)

```
// Thread A
while (tst_fzsync_run_a(&fzsync_pair)) {
    sock = tst_alg_setup_reqfd(...);
    tst_fzsync_start_race_a(&fzsync_pair);
    TEST(fchownat(sock, /*this user*/));
    tst_fzsync_end_race_a(&fzsync_pair);
    ...
    if (TST_RET == -1 && TST_ERR == ENOENT) {
        tst_res(TPASS | TTERRNO, ...
```

```
// Thread B
while (tst_fzsync_run_b(&fzsync_pair)) {

    tst_fzsync_start_race_b(&fzsync_pair);
    dup2(fd, sock);
    tst_fzsync_end_race_b(&fzsync_pair);
}
```

- 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 cc

Meanwhile in **net/socket.c**

```
// Thread A, inode lock is held
static int sockfs_setattr(
    struct dentry *dentry /* has sock */,
    struct iattr *iattr) {
    ...
        if (sock->sk)
            sock->sk->sk_uid = iattr->ia_uid;
        else
            err = -ENOENT;
    ...
```

```
// Thread B
static void __sock_release(
    struct socket *sock,
    struct inode *inode) {
    ...
    if (inode) inode_lock(inode);
 // af_alg_release -> sock_put(sock->sk)
    sock->ops->release(sock);
    if (inode) inode_unlock(inode);
    ...
```

- **__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 te
- This causes **`fchownat`** to return *much quicker* with **`EBADF`**.
- If this happens consistently, our delay range for **`fchownat`** will be too short.

Delay bias

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

- 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

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