Fuzzing for Race bug detection

Alma Oracevic

alma.oracevic@bristol.ac.uk

- Fuzzing has been very successful in detecting memory corruption bugs. Applying fuzzing for data race bugs is interesting!
- We will be discussing a recent article:
  - D. R. Jeong, K. Kim, B. Shivakumar, B. Lee and I. Shin, "*Razzer: Finding Kernel Race Bugs through Fuzzing,*" 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2019, pp. 754-768, doi: 10.1109/SP.2019.00017.
  - https://lifeasageek.github.io/papers/jeong-razzer.pdf
  - Read the paper up to Section III. You can skip low-level implementation details in the subsection III.B **Per-Core Scheduler in Hypervisor**

Concurrency Bugs Detection via Fuzzing

Concurrency Bugs Detection via Fuzzing

- Concurrency bugs are caused by non-deterministic interleavings between shared memory accesses.

Concurrency Bugs Detection via Fuzzing

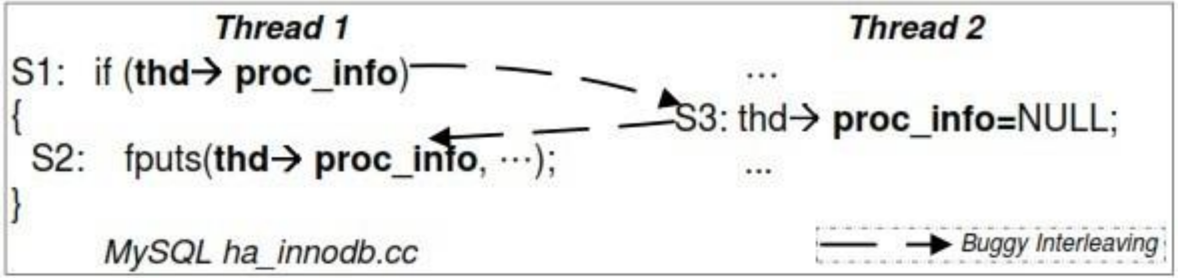
- Concurrency bugs are caused by non-deterministic interleavings between shared memory accesses.
- Their effects propagate through data and control dependencies until they cause software to crash, hang, produce incorrect output, etc

Concurrency Bugs Detection via Fuzzing

- Concurrency bugs are caused by non-deterministic interleavings between shared memory accesses.
- Their effects propagate through data and control dependencies until they cause software to crash, hang, produce incorrect output, etc
- Interleavings are not only complicated to reason about, but they also dramatically increase the state space of software.

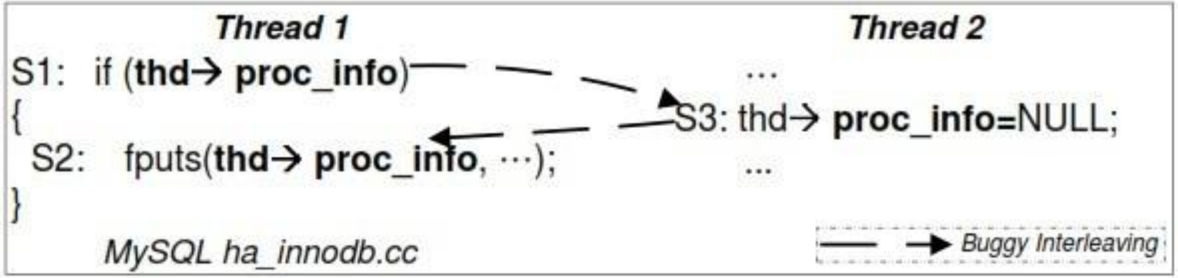
Common Concurrency bugs

Common Concurrency bugs

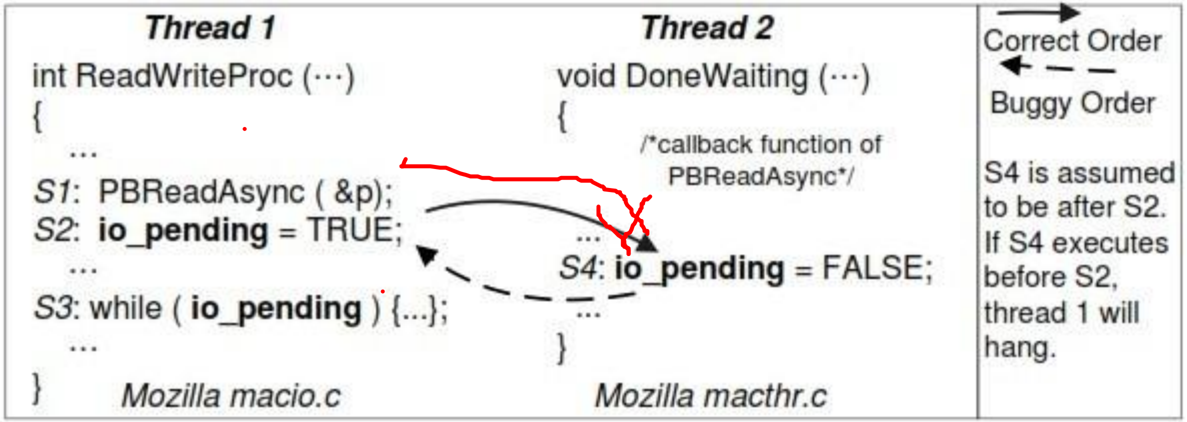


Atomicity violation

Common Concurrency bugs

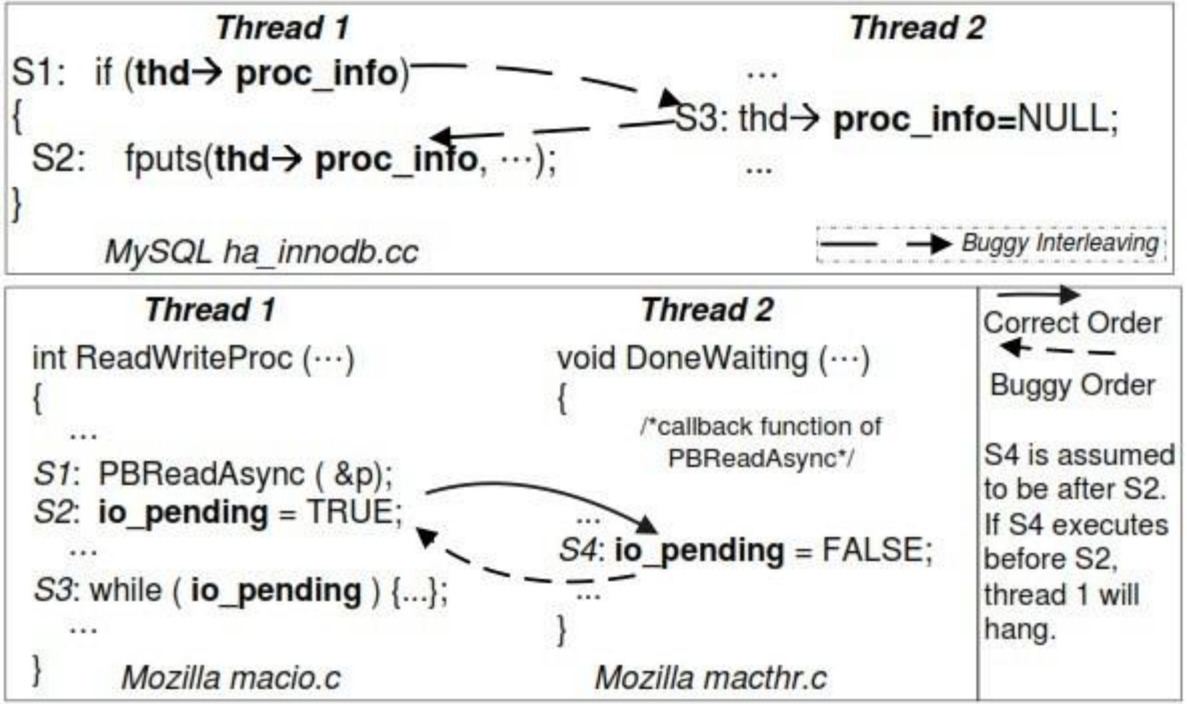


Atomicity violation



Write-write order violation

Common Concurrency bugs



Atomicity violation

Write-write order violation

Use-after free etc.

Fuzzing the scheduler: existing approaches

- Identify shared objects
- An input that executes instructions involving shared objects
- Thread scheduler
  - Rather than letting OS decides, introducing a scheduler that can control the thread scheduling
  - Schedule threads w.r.t. different ordering

Razzer: Kernel race bug detection

- In: 2019 IEEE Symposium on Security and Privacy (SP)
- Involves static and dynamic analysis
- Found 30 new race bugs in the latest kernel

Razzer: Kernel race bug detection

- In: 2019 IEEE Symposium on Security and Privacy (SP)
- Involves static and dynamic analysis
- Found 30 new race bugs in the latest kernel
- Main Idea:
  - Find shared interleaved objects (static analysis)
  - Find an input (by fuzzing) that hits a race (in single thread)
  - Use the input for fuzzing the interleaving of thread in kernel

Data race condition

- a data race occurs when two memory access instructions in a target program meet the following three conditions:
  1) they access the same memory location.
  2) at least one is a write instruction. And
  3) they are executed concurrently.

1. Static analysis component

Identifying Race candidates (`RacePair`$_{cand}$)

- Instructions that access (points) to the same memory location (satisfy the three conditions stated earlier)
- Point-to analysis
- Difficult to get it right!
  - ➢ Interprocedural analysis
  - ➢ Conservative
  - ➢ Partition based analysis (scalability)
  - ➢ Rather than analysing the entire kernel code, it partition the space w.r.t. directory structure, e.g. `Kernel, mm, fs, drivers`

- we use RacePair$_{true}$ for those that are confirmed to meet the three conditions.

2. Scheduler in Hypervisor

- Running the Razzer on a tailored VM
  - Fuzzing mulit-threaded program in guest user-land
  - Triggering races in guest OS
- Uses Virtual Machine Control Structure to:
  - Set hardware breakpoints
  - To catch when the interrupt occurs
- Resume per-Core Execution
  - At each breakpoint, ability to decide which thread to resume.

3. Two-phase fuzzing

- Single-Thread Fuzzing
  - User program generation (Single thread)
    - ➢ Random sequences of syscalls with random values of parameter
    - ➢ It uses Syzkaller (a kernel fuzzer)
  - User program execution (single thread)
    - ➢ Execute the above program and monitors (kcov)
      - Checks if two syscalls execute addresses related to a single $\texttt{RacePair}_{cand}$
    - ➢ It annotates such syscalls with the corresponding addresses from $\texttt{RacePair}_{cand}$

3. Two-phase fuzzing conti…

- Multi-Tread generator
  - Creates a multi-thread version of the single-thread user program
    - If the annotated syscalls are i, j
    - Corresponding instructions are RP_i and RP_ j

3. Two-phase fuzzing conti…

- Multi-Tread generator
  - Creates a multi-thread version of the single-thread user program
    - If the annotated syscalls are i, j
    - Corresponding instructions are RP_i and RP_ j

```
# Get pinned threads, thr0 and thr1
thr0 = get_pinned_thread(vCPU0)
thr1 = get_pinned_thread(vCPU1)

# Assign syscalls to thr0 and thr1
syscalls = get_syscalls(Pst)
thr0.add_syscalls(syscalls[:i])
thr1.add_syscalls(syscalls[i+1:j])

# Determine the execution order
r = random([vCPU0, vCPU1])
thr0.add_hypercall(hcall_order(r))

# Trigger and check races
thr0.add_hypercall(hcall_set_bp(vCPU0, RP_i))
thr0.add_syscalls(syscalls[i])
thr0.add_hypercall(hcall_check-race())

thr1.add_hypercall(hcall_set_bp(vCPU1, RP_j))
thr1.add_syscalls(syscalls[j])
thr1.add_hypercall(hcall_check_race())
```

3. Two-phase fuzzing conti…

- Multi-Thread Executor
  - Sets breakpoints at addresses in $\texttt{RacePair}_{\texttt{cand}}$
  - Checks if the breakpoints are hit
  - Concrete addresses pointed to by the respective instructions
- Resume threads (one by one) to check if race vulnerability is triggered
- Several address sanitizers are enabled during the kernel compilation.