Razzer: Finding Kernel Race Bugs through Fuzzing

Dae R. Jeong†  Kyungtae Kim∗  Basavesh Shivakumar∗  Byoungyoung Lee‡  Insik Shin†

†Korea Advanced Institute of Science and Technology
‡Seoul National University
∗Purdue University
Kernel Vulnerability

Attacker can control the entire system
Fuzzing: Focused to Extend Coverage

• Fuzzing
  • One of the most practical approaches in finding vulnerabilities

• Coverage-guided fuzzing
  • It gathers **interesting** inputs that extend code coverage.
  • The more coverage, the more vulnerabilities
Race Bugs

• Assumption: Race condition between two threads

• Race condition occurs if following three conditions meet
  • Two instructions access the same memory location
  • At least one of two is a write instruction
  • These two are executed concurrently

• If a race occurs, the computational results may vary depending on the execution order
  • A race vulnerability is caused by the execution order unintended by developers.
Inefficient Fuzzing for Race Bugs

• Traditional fuzzers are inefficient to find race bugs
  • Instructions should be executed within a specific time window
    • Called as race window
  • Execution orders are not determined by the fuzzer
    • Execution orders are determined by the kernel scheduler
Inefficient Fuzzing for Race Bugs: Example

Thread 1

Race window

Syscall: open()

len = strlen(file_name);
buf = kmalloc(len);

strcpy(buf, file_name);

Buffer overflow!

Thread 2

Syscall: rename()

strcpy(file_name, longer_name);

file_name is longer than the allocated buffer
Inefficient Fuzzing for Race Bugs: Syzkaller

- Syzkaller
  - A kernel syscall fuzzer developed by Google

- Run Syzkaller to find three race bugs with limited set of syscalls
  - CVE-2016-8655
  - CVE-2017-17712
  - CVE-2017-2636

- None of CVEs was found within 10 hours
  - Traditional fuzzing is inefficient to find race bugs
  - Razzer can find all of them within 7~30 minutes
Our approach: Razzer

Code coverage + Thread interleaving

len = strlen(file_name);
buf = kmalloc(len);
strcpy(file_name, longer_name);
strcpy(buf, file_name);
Our approach: Razzer

Thread 1

- Syscall: open()
  - `len = strlen(file_name);`
  - `buf = kmalloc(len);`
  - `strcpy(buf, file_name);`
  - Buffer overflow!

Thread 2

- Syscall: rename()
  - `strcpy(file_name, longer_name);`
Design Overview

Source code → Static analysis → Offline analysis → Over-approximated data races → Online testing → Single-thread fuzzing → Multi-thread fuzzing
Design Overview

Offline analysis

Source code → Static analysis → Over-approximated data races

Online testing

Multi-thread input → Single-thread fuzzing → Multi-thread fuzzing
Static Analysis

• Identifying instructions that may race
  • Teaching Razzer where to install breakpoints to trigger race

• Inclusion-based points-to analysis
  • Also known as Andersen-style points-to analysis

• This static analysis certainly has false positives
  • Next phases (fuzzing) takes care of this issue because it is “fuzzing”
Razzer identified \textbf{3.4M} race candidates over the entire Linux kernel.

```c
len = strlen(file_name);
buf = kmalloc(len);
strcpy(buf, file_name);
```

```c
strcpy(file_name, longer_name);
```
Design Overview

Offline analysis

Source code → Static analysis → Over-approximated data races

Online testing

Single-thread fuzzing → Multi-thread input → Multi-thread fuzzing
Single-thread Fuzzing

Thread 1

Syscall: open()

```c
len = strlen(file_name);
buf = kmalloc(len);
strcpy(buf, file_name);
```

Syscall: rename()

```c
strcpy(file_name, longer_name);
```
Transformation to Multi-thread Input

Thread 1

```c
strcpy(buf, file_name);

len = strlen(file_name);

buf = kmalloc(len);
```

Thread 2

```c
strcpy(file_name, longer_name);
```
Design Overview

Offline analysis
- Source code
- Static analysis
- Over-approximated data races

Online testing
- Multi-thread input
- Single-thread fuzzing
- Multi-thread fuzzing
Multi-thread Fuzzing

Two threads access the same memory ➔ A race condition is occurred

CPU 1

Thread 1

Hypercalls

Syscall $n$

len = strlen(file_name);
buf = kmalloc(len);
strcpy(buf, file_name);

CPU 2

Thread 2

Hypercalls

Syscall $m$

strcpy(file_name, longer_name);

Hypervisor

Guest VM

strcpy(buf, file_name);
Implementation

• **Static analysis**
  • Implemented using SVF which is based on LLVM compiler suite

• **Single-thread/Multi-thread fuzzing**
  • Implemented based on Syzkaller
  • Deterministic scheduler
    • Implemented using QEMU/KVM
    • Exposing hypercall interfaces to support per-core breakpoint
Evaluation

- 30 new races in the Linux kernel
- 15 were fixed

### Kernel crash summary

**Use-after-free**

- KASAN: slab-out-of-bounds write in tty_insert_flip_string_flag
- KASAN: null-ptr-deref Read in rds_ib_get_mr
- KASAN: null-ptr-deref Read in list_tru_del
- BUG: unable to handle kernel NULL ptr dereference in corrupted

**Heap overflow**

- KASAN: slab-out-of-bounds Write in __register_sysctl_table
- KASAN: use-after-free Write in skb_release_data
- KASAN: invalid-free in plock_free

**Double free**

- Kernel BUG at lib/list_debug.c:L!NE!
- INFO: trying to register non-static key in __handle_rmm_fault
- KASAN: use-after-free Read in vhost-chr_write_iter
- BUG: soft lockup in vmemdup_user
- KASAN: use-after-free Read in rds_tcp_accept_one
- WARNING in sg_rq_end_io
- BUG: soft lockup in snd_virmidi_input_trigger
- KASAN: null-ptr-deref Read in smc_ioctl
- KASAN: null-ptr-deref Read in binder_update_page_range
- WARNING in port_delete
- KASAN: null-ptr-deref in inode_permission

**Reachable Warning**

- WARNING in __static_key_slow_dec
- WARNING at net/packet/af_packet.c:L!NE!
- WARNING in refcount_dec
- unable to handle kernel paging request in snd_seq_oss_readq_puts

**Reachable Assertion**

- WARNING in ip_recv_error
- WARNING in remove_proc_entry
- KASAN: null-ptr-deref Read in ip6gre_exit_batch_net

**Null ptr deref**

- KASAN: use-after-free Read in nd_jump_root
- KASAN: use-after-free Read in link_path_walk
- BUG: unable to handle kernel paging request in __inet_check_established

**Page Fault**

- KASAN: null-ptr-deref Read in ata_pio_sector
- WARNING in init_module
Evaluation: Comparison with Syzkaller

- Run Razzer and Syzkaller with limited set of syscalls

- Razzer found race bugs 23~85 faster than Syzkaller
  - Razzer found 3 race bugs within short time
  - Syzkaller didn’t find 3 race bugs within 10 hours

<table>
<thead>
<tr>
<th>Race bugs</th>
<th>Syzkaller</th>
<th>Razzer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of exec</td>
<td>Time</td>
</tr>
<tr>
<td>CVE-2016-8655</td>
<td>29 M</td>
<td>10 hrs</td>
</tr>
<tr>
<td>CVE-2017-17712</td>
<td>37 M</td>
<td>10 hrs</td>
</tr>
<tr>
<td>CVE-2017-2636</td>
<td>5 M</td>
<td>10 hrs</td>
</tr>
</tbody>
</table>
Conclusion

• Razzer, a new fuzzer focusing on race bugs

• Taming non-deterministic behavior of races

• Combining static analysis and fuzzing

• Source code (by May 25, 2019)
  • https://github.com/compsec-snu/razzer
Thank you

Dae R. Jeong
threeearcat@gmail.com