Finding Multiple Bug Effects for More Precise Exploitability Estimation

Zhenpeng Lin & Yueqi Chen
The Pennsylvania State University
Kernel bugs found by Syzkaller

- **syzbot**: continuous kernel fuzzing
  - Public
  - ~4400 bugs for 4 years
  - ~3000 fixed bugs, ~1000 open bugs
  - Exploitability of these bugs are unknown
    - zero day in upstream
    - zero day in vendors’ kernel
Exploitability

• Estimate the consequence of bugs
• Promote bug fixes and fixes adoption
• Guide the design of hardening
Challenges of knowing exploitability

• Proving exploitability is hard
  – write the exploit!

• Proving unexploitability?
  – is even harder
  – no path leading to exploitation
  – talked by some academic research
  – not realistic for kernel
Exploitability approximation

- Approximate the likelihood of exploitation
- Based on the read/write ability of UAF/OOB bugs?
  - No
  - Exploitability of UAF
    - Transfer UAF object to others
    - From UAF bugs to information leaking
  - Exploitability of OOB
    - Exploit kernel with 4 zero-bytes overflow
Exploitability approximation (cont.)

• Based on the type of bug
  – Likely to exploit
    • UAF, double/invalid free
    • OOB
  – Less likely to exploit
    • WARNING
    • INFO
    • GPF
    • Null-ptr-deference
    • ...
The reliability of approximation

• How bugs are underestimated
  – severe bug doesn’t show memory corruption
  – severe bug shows limited memory corruption ability

• How to improve the reliability
  – Find the true effect of bugs
Exploitability being underestimated

• Syzkaller generates incomplete errors
  – Misses KASAN errors when “panic_on_warn” is set
Exploitability being underestimated

WARNING in xfrm_state_fini

KASAN error is not shown

---

WARNING: CPU 0 PID: 5920 at net/xfrm/xfrm_state.c:2389 xfrm_state_fini+0x1f1+0x260
MODIFIED:\n
CPU: 0 PID: 5920 Comm: worker/sa4 Not tainted 5.1.0-rc6 #83
Hardware name: Google Compute Engine/Google Compute Engine
Workqueue: netns cleanup_net
Call Trace:
  _dump_stack lib/dump_stack.c:177 [inline]
  dump_stack+0x172/0x1f0 lib/dump_stack.c:113
  panic+0x2e0/0x65c kernel/panic.c:214
  _warn.cold+0x2e0/0x4e5 kernel/panic.c:571
  report_bug+0x263/0x2b0 lib/bug.c:186
  fixup_bug arch/x86/kernel/traps.c:179 [inline]
  fixup_bug arch/x86/kernel/traps.c:174 [inline]
  do_error_trap+0x13b/0x200 arch/x86/kernel/traps.c:272
  do_invalid_op+0x37/0x50 arch/x86/kernel/traps.c:291
  _invalid_op+0x14/0x20 arch/x86/kernel/traps.c:291
  __x86_entry+0x64.S:973
RIP: 0x0100x80:state_fini+0x218/0x280 net/xfrm/xfrm_state.c:2389
Code: 41 5e 5d c3 e8 28 b0 66
ESP: 001e:ffff888a09987bd0 ET
RAR: ffff888a09970000 RXB: ff
RDX: 0000000000000000 RXI: ff
RSP: ffff888a0997e0f8 R08: ff
R10: 0000000000000000 R11: 0000000000000000 R12: ffff8880a7c0a80
R13: ffff888a09987bc8 R14: ffff888930658 R15: ffffcc000000000
xfrm_net_exit+0x25/0x270 net/xfrm/xfrm_policy.c:3934
ops_exit_list_base+0xb0/0x160 net/core/net_namespace.c:153
cleanup_net+0x3f8/0x960 net/core/net_namespace.c:155
process_one_work+0x9f6e/0x1790 kernel/workqueue.c:1269
worker_thread+0x4e0/0x6e0 kernel/workqueue.c:2415
kthread+0x3cf/0x430 kernel/kthread.c:253
ret_from_fork+0x3a/0x50 arch/x86/kernel/traps.c:1352
Kernel Offset: disabled
Rebooting in 84400 seconds...

---

WARNING: CPU 1 PID: 5920 at net/xfrm/xfrm_state.c:2389 xfrm_state_fini+0x1f1+0x260
MODIFIED:\n
CPU: 1 PID: 5920 Comm: worker/sa4 Not tainted 5.1.0-rc6 #83
Hardware name: Google Compute Engine/Google Compute Engine
Workqueue: netns cleanup_net
Call Trace:
  _dump_stack lib/dump_stack.c:177 [inline]
  dump_stack+0x172/0x1f0 lib/dump_stack.c:113
  panic+0x2e0/0x65c kernel/panic.c:214
  _warn.cold+0x2e0/0x4e5 kernel/panic.c:571
  report_bug+0x263/0x2b0 lib/bug.c:186
  fixup_bug arch/x86/kernel/traps.c:179 [inline]
  fixup_bug arch/x86/kernel/traps.c:174 [inline]
  do_error_trap+0x13b/0x200 arch/x86/kernel/traps.c:272
  do_invalid_op+0x37/0x50 arch/x86/kernel/traps.c:291
  _invalid_op+0x14/0x20 arch/x86/kernel/traps.c:291
  __x86_entry+0x64.S:973
RIP: 0x0100x80:state_fini+0x218/0x280 net/xfrm/xfrm_state.c:2389
Code: 41 5e 5d c3 e8 29 b0 66
ESP: 001e:ffff888a09987bd0 ET
RAR: ffff888a09970000 RXB: ff
RDX: 0000000000000000 RXI: ff
RSP: ffff888a0997e0f8 R08: ff
R10: 0000000000000000 R11: 0000000000000000 R12: ffff8880a7c0a80
R13: ffff888a09987bc8 R14: ffff888930658 R15: ffffcc000000000
xfrm_net_exit+0x25/0x270 net/xfrm/xfrm_policy.c:3934
ops_exit_list_base+0xb0/0x160 net/core/net_namespace.c:153
cleanup_net+0x3f8/0x960 net/core/net_namespace.c:155
process_one_work+0x9f6e/0x1790 kernel/workqueue.c:1269
worker_thread+0x4e0/0x6e0 kernel/workqueue.c:2415
kthread+0x3cf/0x430 kernel/kthread.c:253
ret_from_fork+0x3a/0x50 arch/x86/kernel/traps.c:1352
Kernel Offset: disabled
Rebooting in 84400 seconds...

---

BUG: KASAN: use-after-free in __lock_acquire+0x3b96/0x3d10
Read of size 8 at addr ffff8886105e018 by task swapper/0/0

---

WARNING in xfrm_state_fini

KASAN: use-after-free Read in __lock_acquire
Exploitability being underestimated

• Syzkaller generates incomplete errors
  – Misses KASAN errors when “panic_on_warn” is set
  – Only reports first error kernel triggers
Exploitability being underestimated

KASAN error is ignored

WARNING: held lock freed!
Exploitability being underestimated

- Incomplete error reported by Syzkaller
  - Misses KASAN errors when “panic_on_warn” is set
  - Only reports first error kernel triggers

- **Multiple Error Behaviors (MEB)**
Multiple Error Behaviors

• With the *same* root cause, but *different* errors at *different* sites

Input 1
Input 2
Input 3

root cause

WARNING: xxx :(  
GPF: xxx :(  
KASAN: UAF in xxx 😊

Exposing MEB to avoid underestimation
“Unexploitable” path v.s. “Exploitable” path

```c
static void tun_attach(struct tun_struct *tun, ...) {
    if (tun->flags & IFF_NAPI) {
        // initialize a timer
        hrtimer_init(&napi->timer, CLOCK_MONOTONIC,
                     HRTIMER_MODE_REL_PINNED);
        // link current napi to the device's napi list
        list_add(&napi->dev_list, &dev->napi_list);
    }
}

static void tun_detach(struct tun_file *tfile, ...) {
    struct tun_struct *tun = rtnl_dereference(tfile->tun);
    if (tun->flags & IFF_NAPI) {
        // GPF happens if timer is uninitialized
        hrtimer_cancel(&tfile->napi->timer);
        // remove the current napi from the list
        netif_napi_del(&tfile->napi);
    }
    destroy(tfile); // free napi
}

void free_netdev(struct net_device *dev) {
    list_for_each_entry_safe(p, n,
                             &dev->napi_list, dev_list)
        netif_napi_del(p); // use-after-free
}
```
"Unexploitable" path v.s. "Exploitable" path

```c
static void tun_attach(struct tun_struct *tun, ...)
{
    if (tun->flags & IFF_NAPI) {
        // initialize a timer
        hrtimer_init(&napi->timer, CLOCK_MONOTONIC,
                     HRTIMER_MODE_REL_PINNED);
        // link current napi to the device's napi list
        list_add(&napi->dev_list, &dev->napi_list);
    }
}

static void tun_detach(struct tun_file *tfile, ...)
{
    struct tun_struct *tun = rtnl_dereference(tfile->tun);
    if (tun->flags & IFF_NAPI) {
        // GPF happens if timer is uninitialized
        hrtimer_cancel(&tfile->napi->timer);
        // remove the current napi from the list
        netif_napi_del(&tfile->napi);
    }
    destroy(tfile); // free napi
}

void free_netdev(struct net_device *dev)
{
    list_for_each_entry_safe(p, n,
                              &dev->napi_list, dev_list)
        netif_napi_del(p); // use-after-free
}
```
“Unexploitable” path v.s. “Exploitable” path

void free_netdev(struct net_device *dev) {
    list_for_each_entry_safe(p, n, &dev->napi_list, dev_list)
    netif_napi_del(p); // use-after-free
}

void free_netdev(struct net_device *dev) {
    list_for_each_entry_safe(p, n, &dev->napi_list, dev_list)
    netif_napi_del(p); // use-after-free
}

static void tun_attach(struct tun_struct *tun, ...) {
    if (tun->flags & IFF_NAPI) {
        // initialize a timer
        hrtimer_init(&napi->timer, CLOCK_MONOTONIC,
        HRTIMER_MODE_REL_PINNED);
        // link current napi to the device's napi list
        list_add(&napi->dev_list, &dev->napi_list);
    }
}

static void tun_detach(struct tun_file *tfile, ...) {
    struct tun_struct *tun = rtnl_dereference(tfile->tun);
    if (tun->flags & IFF_NAPI) {
        // GPF happens if timer is uninitialized
        hrtimer_cancel(&tfile->napi->timer);
        // remove the current napi from the list
        netif_napi_del(&tfile->napi);
    }
    destroy(tfile); // free napi
}
“Unexploitable” path v.s. “Exploitable” path

1. `tun_attach` with `IFF_NAPI` disabled
   - no timer
   - current napi not in the list

2. `tun_detach` with `IFF_NAPI` enabled
   - cancel the timer

Null-ptr-def happens
“Unexploitable” path v.s. “Exploitable” path

1. tun_attach with IFF_NAPI enabled
   - initialize the timer
   - current napi linked in the list
2. tun_detach with IFF_NAPI disabled
   - napi still in the list
   - napi freed by destroy(tfile)
3. free_netdev
   - dereference the dangling pointer

UAF happens
Exploitability of two behaviors

• Exploit the Null-ptr-dereference
  – mapping at 0 is not allowed

• Exploit the UAF
  ▪ netif_napi_del(napi)
    ▪ kfree_skb(napi->skb)
      ▪ napi->skb->destructor(napi->skb) (Hijack control flow)

Precise exploitability estimation needs to expose Multiple Error Behaviors of bug.
Finding Multiple Error Behaviors

• Static analysis
  – A lot of false positives
  – No input

• Fuzzing
  – Code-coverage feedback will detour the path
  – How to restrict the fuzzing scope
  – What is the proper fuzzing scope
Finding Multiple Error Behaviors (cont.)

• Some observations
  – Linux kernel’s design is object-oriented
  – Bugs result from incorrect usage of kernel object
  – Incorrectness propagates to different places

• Object-driven kernel fuzzing
  – Static analysis to find critical objects
  – Under-scope fuzzing based on the reachability of identified objects
Object-driven kernel fuzzing

- Static analysis to find critical objects
Object-driven kernel fuzzing (cont.)

- Static analysis to find critical objects

\[
\text{base} = \text{READ\_ONCE}(&\text{timer} -> \text{base}) \text{ in } \text{hrtimer\_active}
\]

\[
\text{hrtimer\_try\_to\_cancel}(&\text{timer}) \text{ in } \text{hrtimer\_cancel}
\]

\[
\text{hrtimer\_cancel}(&\text{tfile} -> \text{napi} -> \text{timer}) \text{ in } \text{tun\_detach}
\]
Object-driven kernel fuzzing (cont.)

- Under-scope fuzzing based on Syzkaller
  - Instrument basic blocks involved with critical objects
  - Only inputs reaching these objects are interesting
Experiment setup

• 60 kernel bugs (2017-2021)
• Each cases comes with a patch
• 7 days for Syzkaller and our tool
• Manually categorize reports tied to the same bug
Experiment results

• Exploitability escalation
  – less likely to exploit bug (44/60)
    • 4 escalation found by Syzkaller
    • 26 escalation found by our tool, 3 error behaviors on avg.

• More exploit potential
  – likely to exploit bug (16/60)
    • Syzkaller found 1 bug has other exploitable behaviors
    • Our tool found 8 bugs have other exploitable behaviors
Takeaway

- A kernel bug could have *Multiple Error Behaviors*
- MEB contribute to more precise exploitability estimation
- Finding MEB automatically is possible
- Utilizing kernel objects to find MEB is effective and efficient.

Zhenpeng Lin (@Markak_)
https://zplin.me
Looking for summer internship!
LINUX SECURITY SUMMIT