syzbot

and the tale of thousand kernel bugs

Linux Security Summit 2018
Dmitry Vyukov, dvyukov@
Agenda

- Kernel bug disaster
- What we are doing
- Where we need help
"Civilization runs on Linux" [1]

- Android (2e9 users)
- Cloud, servers
- Desktops, notebooks, chromebooks
- Cars
- Air/Car Traffic Control, Nuclear Submarines, Power Plants
- Large Hadron Collider, International Space Station
- 
- Our coffee machines!

[1] from SLTS project which aims at maintaining kernel releases for 20+ years for industrial use
Security is Critical

- Protects privacy of 2 billion people
- Protects corp, government information
- Protects safety-critical systems
- The first line of defence for:
  - all incoming network packets
  - untrusted apps
  - VM guests
  - USB/NFC/Bluetooth (inserting a USB clicker into your notebook)
- Cars/phones/plants: stability and safety are also critical

Linux kernel is one of the most security-critical components in the world today.
Tip of The Iceberg

Bugs with logos and bold headlines

Leading Linux distros dawdle as kernel flaw persists

Code Execution Flaw Affected Linux Kernel Since 2005
Kernel has lots of bugs

453 CVEs in 2017 including:

- 169 code execution
- 125 gain privileges/information

But lots are unaccounted!

4100 "official" bug fixes in 2017 (again lots are unaccounted).
syzbot: continuous kernel fuzzing

For 12 months ~200 bugs/month:

- **1000 bugs** in upstream kernel
- 1200 bugs in Android/ChromeOS/internal kernels

+ **1000** bugs reported manually before syzbot (~40 bugs/mo for 2 years)

= 3200 bugs
USB Stack State

Barely scratching the surface yielded 80+ externally triggerable bugs (18 CVEs).

Did not even get past handshake (WIP)

USB is not special. Flow of bugs is representative for any subsystem (kvm, tcp, udp, rdma, sound, 9p, bpf, you name it)
<table>
<thead>
<tr>
<th>Title</th>
<th>Repro</th>
<th>Count</th>
<th>Last</th>
<th>Reported</th>
</tr>
</thead>
<tbody>
<tr>
<td>KASAN: slab-out-of-bounds Read in nfts_attr_find</td>
<td>C</td>
<td>1</td>
<td>23d</td>
<td>22d</td>
</tr>
<tr>
<td>KASAN: slab-out-of-bounds Read in pfkey_add</td>
<td>C</td>
<td>769</td>
<td>1d06h</td>
<td>130d</td>
</tr>
<tr>
<td>KASAN: slab-out-of-bounds Write in process_preds</td>
<td>C</td>
<td>456</td>
<td>1h35m</td>
<td>13d</td>
</tr>
<tr>
<td>KASAN: stack-out-of-bounds Read in rdma_resolve_addr</td>
<td>C</td>
<td>3</td>
<td>26d</td>
<td>46d</td>
</tr>
<tr>
<td>KASAN: stack-out-of-bounds Read in update_stack_state</td>
<td>C</td>
<td>312</td>
<td>3h53m</td>
<td>62d</td>
</tr>
<tr>
<td>KASAN: stack-out-of-bounds Read in xfrm_state_find (5)</td>
<td>C</td>
<td>4</td>
<td>23d</td>
<td>23d</td>
</tr>
<tr>
<td>KASAN: stack-out-of-bounds Write in compat_copy_entries</td>
<td>syz</td>
<td>4</td>
<td>3h49m</td>
<td>3h57m</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in __dev_queue_xmit</td>
<td>C</td>
<td>9</td>
<td>101d</td>
<td>111d</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in __put (2)</td>
<td></td>
<td>1</td>
<td>14d</td>
<td>5d17h</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in __list_add_valid (5)</td>
<td>C</td>
<td>16</td>
<td>23d</td>
<td>30d</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in __list_del_entry_valid (4)</td>
<td>C</td>
<td>16</td>
<td>23d</td>
<td>30d</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in decode_session4</td>
<td>C</td>
<td>3</td>
<td>25d</td>
<td>25d</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in build_segment_manager</td>
<td>C</td>
<td>5</td>
<td>4d00h</td>
<td>4d16h</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in ccid2_hc_tx_packet_recv</td>
<td>C</td>
<td>3</td>
<td>13d</td>
<td>22d</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in cma_cancel_operation</td>
<td>C</td>
<td>8</td>
<td>5d02h</td>
<td>22d</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in debugfs_remove (2)</td>
<td>C</td>
<td>1</td>
<td>4d02h</td>
<td>1d22h</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in ip6_xmit</td>
<td>C</td>
<td>5174</td>
<td>33d</td>
<td>110d</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in ip_defrag</td>
<td>C</td>
<td>1</td>
<td>115d</td>
<td>110d</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in iput</td>
<td>C</td>
<td>2</td>
<td>7d13h</td>
<td>7d08h</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in irq_bypass_register_consumer</td>
<td>C</td>
<td>292</td>
<td>7d01h</td>
<td>174d</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in l2tp_session_create</td>
<td>C</td>
<td>119</td>
<td>31d</td>
<td>98d</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in l2tp_session_register</td>
<td>C</td>
<td>4</td>
<td>21d</td>
<td>67d</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in memcmp</td>
<td>C</td>
<td>1</td>
<td>88d</td>
<td>87d</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in nfts_read_locked_inode</td>
<td>C</td>
<td>1</td>
<td>20d</td>
<td>20d</td>
</tr>
<tr>
<td>KASAN: use-after-free Read in radix_tree_next_chunk</td>
<td>C</td>
<td>1637</td>
<td>5h11m</td>
<td>24d</td>
</tr>
</tbody>
</table>
Bug split

Use-after-free 18.5%
Heap-out-of-bounds 5.2%
Stack-out-of-bound 2.4%
Double-free 0.8%
Wild-access 4.8%
Uninit-memory 4.0%
GPF 20.2%
BUG/panic/div0 10.3%
deadlock/hang/stall 12.5%
WARNING 21.1%

Modest estimation: 500 security bugs (not counting DoS; very few have CVEs).
Exploit != use-after-free

- "unresponsive" machine -> full guest->host escape
  - page ref leak
  - [CVE-2017-2596](https://security advisories) / kvm: fix page struct leak in handle_vmon

- **WARNING** -> inter-VM/process info leaks
  - failure to restore registers
  - [WARNING in __switch_to](https://security advisories) / WARNING in fpu_copy

- **stall** -> remote network DoS
  - `lockup in udp[v6]_recvmsg`
  - anything remotely triggerable is a concern
Every "looks good and stable" release we produce contains >20'000 bugs. No, not getting better over time. No, this is not normal.

+ not backported fixes (700+)
+ not fixed upstream bugs (200+)
+ not found bugs (???)
+ not detectable yet bugs (???) (info leaks, races)
Distros State

End distros is what matters security-wise in the end.

It isn't always possible for distributions to track the linux-stable tree or fully monitor the commits that flow into it.

CVE-2017-18344 discussion on linux-distros@

Stable process is not fully working, CVE process is not working.

Why?
"Stable" releases

Backports/month

<table>
<thead>
<tr>
<th>Version</th>
<th>1st year</th>
<th>2nd year</th>
</tr>
</thead>
<tbody>
<tr>
<td>v3.4</td>
<td>200</td>
<td>150</td>
</tr>
<tr>
<td>v3.18</td>
<td>150</td>
<td>100</td>
</tr>
<tr>
<td>v4.4</td>
<td>250</td>
<td>200</td>
</tr>
<tr>
<td>v4.9</td>
<td>350</td>
<td>300</td>
</tr>
<tr>
<td>v4.14</td>
<td>700</td>
<td>600</td>
</tr>
</tbody>
</table>
Each bug fork is effectively a new bug for most practical purposes. Hundreds of thousands of bugs for Google. **Millions of bugs industry-wide.**
Goal

Reduce bugs/release $100x$: 20,000 -> 200
Existing Defences Are Not Enough

● Attack surface reduction
  ○ large surface is still open
  ○ most subsystems are still relevant (USB for clients, namespaces for servers)

● Mitigations [1]
  ○ can't mitigate hundreds of arbitrary memory corruptions (assume there are few bugs)
  ○ don't mitigate lots of bug types (races, uninit memory, write what/where)
  ○ some are not backported/enabled (performance!)

[1] KASLR, REFCOUNT_FULL, STACKPROTECTOR, VMAP_STACK, SLAB_FREELISTRANDOM, STRUCTLEAK, RANDSTRUCT, etc
Existing Defences Are Not Enough (2)

- Selinux/namespaces/fs-verity
  - logical protection: directly assume that kernel is not buggy ([1])
  - namespaces open even larger attack surface ([1], [2], [3], [4])

- Hiding buggy code "under root"
  - SELinux/AppArmor/IMA/module signing restrict root
  - root is not trusted on some systems (Android)
  - user still needs to do the thing, so they just issue sudo left and right
What we are doing
What we have

● bug detection:
  ○ KASAN
  ○ KMSAN
  ○ KTSAN

● bug discovery:
  ○ syzkaller

● systematic testing:
  ○ syzbot
KASAN (KernelAddressSANitizer)

- security "workhorse"
- Detects:
  - use-after-free
  - out-of-bounds on heap/stack/globals
- detects bugs at the point of occurrence
- outputs informative reports
- easy to use (CONFIG_KASAN=y)
- based on compiler instrumentation (gcc4.9+ or clang)
- fast: ~2x slowdown, ~2x memory overhead
- upstream in 4.3 kernel
KMSAN (KernelMemorySanitizer)

Detects uses of uninitialized values.

In the context of security:

- **information leaks** (local and remote) [easy to exploit: 1, 2]
- **control-flow subversion** [1]
- **data attacks** (uninit uid) [1, 2]

Not upstreamed yet (on github), work-in-progress.

Already found 50+ bugs.
KTSAN (KernelThreadSanitizer)

Detects data races.

Kernel data races represent security threat:

- TOCTOU (time-of-check-time-of-use) ([1])
- uninit/wrong credentials ([1])
- racy use-after-frees/double-frees ([1], [2], [3], [4])

Prototype on github, frozen due to lack of resources, found 20+ bugs.

Main obstacle: kernel is full of "benign" races (undefined behavior in C).
syzkaller

System call fuzzer:
- grammar-based
- coverage-guided
- unsupervised
- multi-OS/arch/machine

As compared to other kernel fuzzers:
- finds deeper bugs
- provides reproducers
- does regression testing
- scalable to large number of bugs
Syscall Descriptions

Declarative description of system calls:

open(file filename, flags flags[open_flags],
     mode flags[open_mode]) fd
read(fd fd, buf buffer[out], size len[buf])
close(fd fd)

Tests only what's described.
Programs

Descriptions allow to generate and mutate "programs" in the following form:

```
mmmap(& (0x7f0000000000), (0x1000), 0x3, 0x32, -1, 0)
r0 = open(& (0x7f0000000000)="./file0", 0x3, 0x9)
read(r0, & (0x7f0000000000), 42)
close(r0)
```
syzbot: fuzzing automation

- continuous kernel/syzkaller build/update
- test machine management (qemu, GCE VMs, Android phones, ODROID, ...)
- bug deduplication and localization
- bug reporting/status tracking

syzkaller.appspot.com
We need YOU!
More Coverage

More syscall descriptions* -> more bugs. Coverage is not complete.

Poor environment setup: network devices, SELinux policies, etc.

CVE-2017-18017 (remote code exec): didn't test, didn't know netfilter exists
Android use-after-free (severity: high): don't test NSFS

Adding syzkaller descriptions is not hard.

* automatic interface extraction is not feasible (netlink, netfilter, images, string parsing, etc)
Injecting external inputs finds the most critical bugs. Need to test:

- Network packets (currently basic coverage via tun)
- USB
- NFC
- CAN
- Bluetooth
- Guest->host (emulation, vring, vsocks, hypercalls)
- Keyboard, mouse, touchscreen, mic, camera
- ...

Some may need better stubbing support, a-la tun.
Lots of bugs are unfixed

Hundreds of bugs are unfixed:

- Some are bad vulnerabilities
- Others affect stability or are DoS
- Rest harm syzkaller’s ability to uncover new vulnerabilities

Need help:

- Fixing bugs
- Triaging, routing, duping, closing fixed/obsolete
KASAN: manual checks

KASAN checks C accesses wrt `kmalloc()` size.

Does **not** check:

- asm accesses
- hardware accesses
- use-after-free with custom caches
- out-of-bounds with amortized growth

But can be checked with manual memory/access annotations:

```
  kasan_check_write(p, size);
```
KASAN: manual checks: SKB

SKB: core networking data structure, holds packet data.

Uses proactive/amortized growth:

```c
if (pskb_may_pull(skb, 2) {
    // can access skb->data[0-1], but not [2]
    if (pskb_may_pull(skb, 3) {
        // now can access bytes [0-2], but previous skb->data is invalidated
    }
}
```

Very easy to get wrong, bug nest: dozens of remotely-triggerable bugs.

Can make sense to do strict/exact growth under KASAN.
Do not want KASAN annotations sprinkled everywhere.

But some "biggest bang for the buck" can be worthy:

- dma/i2c/spi/virtio?
- USB: something in URB?
- something in filesystems?
- ???
Other Tools

- **KMEMLEAK**: memory leak detector
  - in server context leaks are one of the worst bugs, remote leaks are remote DoS
  - has false positives -> no systematic testing -> bugs are not found/fixed

- **KUBSAN**: Undefined Behavior SANitizer
  - finds some intra-object overflows
  - invalid bools/Enums (control flow hijacking)
  - overflows/invalid shifts (out-of-bound accesses)
  - needs cleanup, fixes face opposition

- **KTSAN**: data race detector
  - will find thousands of hard-to-localize bugs with actionable reports, but...
  - need to say NO to "benign data races" (undefined behavior in C)
  - all concurrent accesses need to be marked
Kernel Testing

Most bugs can be prevented with proper testing. We do need better testing:

- 20'000 bugs/release
- New bugs are introduced at high rate
- New bugs are backported to stable (1, 2, 3, 4, 5, 6, 7)
- Bugs are re-introduced (1, 2)
- Distros don't keep up

Development is slowed down:

- high reliance on manual labor
- delayed releases
- broken builds (bisection :())
- long fix latency (testing :())
- late feedback, reverts
Testing MUST be part of dev process

- Tests need to be easier to write, discover and run
  - userspace tests
  - in-kernel tests with hardware mocking ([kunit](https://kunit.org))
- Tests for new functionalities, regression tests
- Automated continuous testing
- Integration into dev process, presubmit testing
- Use of all available tools (trivial bugs [1], [2], [3])
Thank you!

Q&A

Dmitry Vyukov, dvyukov@
syzkaller coverage-guided algorithm

start with empty corpus of programs
while (true) {
    choose a random program from corpus and mutate it (or generate)
    execute and collect code coverage
    if (gives new coverage)
        add the program to corpus
}

Advantages:
● turns exponential problem into linear (more or less)
● inputs are reproducers
● corpus is perfect for regression testing
KMSAN: uses of uninit values

```c
int x;
put_user(&x, user_ptr);  // reported

int y;
int x = y;              // not reported
put_user(&x, user_ptr);  // reported

(just assigning something to a variable does not make its value initialized)

int x = 0, y, z = 0;
if (foo) x = y + z;      // not reported
...
if (!foo) put_user(&x, user_ptr);  // not reported

(using uninit value in computations is not a use, merely propagation)
```
HWASAN (HardWare assisted AddressSANitizer)

~KASAN, but with substantially smaller memory overhead (~10%).

Intended to be used on real devices (testing, canarying, maybe end users/prod).

Work-in-progress (patches mailed), only arm64 for now (requires TBI).

Will shine more with proper hardware implementation.
Hardware-assisted memory safety

1. We can't fix all bugs.

2. Some installations don't get timely updates (or at all).

Need better mitigations! [SPARC ADI](#) (or similar):

- Detect & mitigate most of use-after-free and out-of-bounds
- 1-5% CPU, 4-5% RAM overhead
- can actually make things faster:
  - don't need stack cookies, slab randomization, fortification, usercopy hardening, CFI, etc
BUG: KASan: **use-after-free** in remove_wait_queue
Write of size 8 by task syzkaller_execu/10568

**Call Trace:**
- list_del include/linux/list.h:107
- __remove_wait_queue include/linux/wait.h:145
- remove_wait_queue+0xfb/0x120 kernel/sched/wait.c:50
  ...
- SYSC_exit_group kernel/exit.c:885

**Allocated:**
- kmem_cache_alloc+0x10d/0x140 mm/slub.c:2517
- sk_prot_alloc+0x69/0x340 net/core/sock.c:1329
- sk_alloc+0x33/0x280 net/core/sock.c:1404
  ...
- SYSC_socketpair net/socket.c:1281

**Freed:**
- kmem_cache_free+0x161/0x180 mm/slub.c:2745
- sk_prot_free net/core/sock.c:1374
- sk_destruct+0x2e9/0x400 net/core/sock.c:1452
  ...
- SYSC_write fs/read_write.c:585
KMSAN report

BUG: KMSAN: uninit-value in ___nf_conntrack_find

Call Trace:
    ___nf_conntrack_find net/netfilter/nf_conntrack_core.c:539
    __nf_conntrack_find_get+0xc15/0x2190 net/netfilter/nf_conntrack_core.c:573
    ...
    __x64_sys_sendto+0x1a1/0x210 net/socket.c:1805

Uninit was stored to memory at:
    __nf_conntrack_confirm+0x2700/0x3f70 net/netfilter/nf_conntrack_core.c:793
    nf_conntrack_confirm include/net/netfilter/nf_conntrack_core.h:71
    ...
    __x64_sys_sendto+0x1a1/0x210 net/socket.c:1805

Uninit was created at:
    kmem_cache_alloc+0xad2/0xbb0 mm/slub.c:2739
    __nf_conntrack_alloc+0x166/0x670 net/netfilter/nf_conntrack_core.c:1137
    init_conntrack+0x635/0x2840 net/netfilter/nf_conntrack_core.c:1219
    ...
    __x64_sys_sendto+0x1a1/0x210 net/socket.c:1805
KTSAN Report (CVE-2015-7613)

ThreadSanitizer: **data-race** in ipc_obtain_object_check

**Read** at 0x123 of size 8 by thread 234 on CPU 5:
- ipc_obtain_object_check+0x7d/0xd0 ipc/util.c:621
- msq_obtain_object_check ipc/msg.c:90
- msgctl_nolock.constprop.9+0x208/0x430 ipc/msg.c:480
- SYSC_msgctl ipc/msg.c:538

**Previous write** at 0x123 of size 8 by thread 567 on CPU 4:
- ipc_addid+0x217/0x260 ipc/util.c:257
- newque+0xac/0x240 ipc/msg.c:141
- ipcget_public ipc/util.c:355
- ipcget+0x202/0x280 ipc/util.c:646
- SYSC_msgget ipc/msg.c:255

**Also:** locked mutexes, thread creation stacks, allocation stack, etc.
Say **No** to "Benign" Data Races

- Proving benignness is time consuming and impossible
- Allows automatic data race bug detection
- Makes code better documented
Proving Benignness

*p = (*p & 0xffffffff) | v;

Option 1:

0: mov (%rdi),%rax
3: and $0xffffffff,%eax
8: or %rax,%rsi
B: mov %rsi,(%rdi)

Option 2:

0: andq $0xffffffff,(%rdi)
7: or %rsi,(%rdi)
This should be atomic, right?

```c
void foo(int *p, int v)
{
    // some irrelevant code
    *p = v;
    // some irrelevant code
}
```
This should be atomic, right?

void foo(int *p, int v)
{
    // some irrelevant code
    *p = v;
    // some irrelevant code
}

void bar(int *p, int f)
{
    int tmp = *p & MASK;
    tmp |= f;
    foo(p, tmp);
}
This should be atomic, right?

```c
void foo(int *p, int v)
{
    // some irrelevant code
    *p = v;
    // some irrelevant code
}

void bar(int *p, int f)
{
    int tmp = *p & MASK;
    tmp |= f;
    foo(p, tmp);
}

    after inlining:
    *p = (*p & MASK) | f;
```
This should be atomic, right? Maybe

```c
void foo(int *p, int v) {
    // some irrelevant code
    *p = v;
    // some irrelevant code
}

void bar(int *p, int f) {
    int tmp = *p & MASK;
    tmp |= f;
    foo(p, tmp);
}

after inlining:
*p = (*p & MASK) | f;
```

```
0: andq $0xffffffff,%rdi
7: or %rsi,%rdi
```
Based on Real Bug

--- a/fs/namespace.c
+++ b/fs/namespace.c
@@ -2212,7 +2212,7 @@ static int do_remount(struct path *path, int flags,
 int mnt_flags,
     lock_mount_hash();
     mnt_flags |= mnt->mnt.mnt_flags &
-             ~MNT_USER_SETTABLE_MASK;
+             WRITE_ONCE(mnt->mnt.mnt_flags, mnt_flags);
     - mnt->mnt.mnt_flags = mnt_flags;
     + WRITE_ONCE(mnt->mnt.mnt_flags, mnt_flags);
     touch_mnt_namespace(mnt->mnt_ns);
     unlock_mount_hash();

Temporary exposes mount without MNT_NOSUID, MNT_NOEXEC, MNT_READONLY flags.
Fragile

- Changing local computations can break such code
- Changing MASK from 0xfe to 0xff can break such code
- New compiler can break such code
- LTO can break such code