Fighting Uninitialized Memory in the Kernel

Signed-off-by: Alexander Potapenko <glider@google.com>

KernelMemorySanitizer (KMSAN)

Tool that detects uses of uninitialized memory.

* runtime library maintains the metadata:
  - bit-to-bit shadow to track uninitialized values
  - creation stack for every 4 uninit bytes (origin)
* Clang instrumentation propagates uninit values
  - copying uninits is not an error
  - using them is an error:
    @ conditions
    @ pointer dereferencing and indexing
    @ values copied to the userspace, hardware etc.

Changes since 2017

* Linux kernel builds with Clang
* kmemcheck is gone!
  - and KMSAN isn't there yet :(

* stable compiler interface
  - upstream Clang supports KMSAN
* reworked shadow layout to support vmalloc()
* crashes--;
* basic asm() support, USB and network infoleak detection

Changes since 2017 (contd.)

* fully integrated with syzkaller
  - reports are premoderated
    @ only true positives are sent upstream
    @ unless fixed by Eric Dumazet :)
  - ~150 bugs reported so far, ~108 of them fixed
* code at http://github.com/google/kmsan
  - rebased on current -rc at least monthly
  - still not upstream

* fun fact: NetBSD has a working KMSAN implementation

http://bit.ly/review-kmsan

Need more eyes:

* how do I send 3KLOC for review?
* better way to organize shadow memory?
* better interaction with printk() and kmalloc() locks
* some debug configs (e.g. LOCKDEP) are broken
* more checks in subsystems
  - DMA, file I/O, virtio, you name it

Uninitialized memory bugs in the kernel

(Wanted to insert a CVE breakdown here -
 if only someone cared about requesting CVEs!)

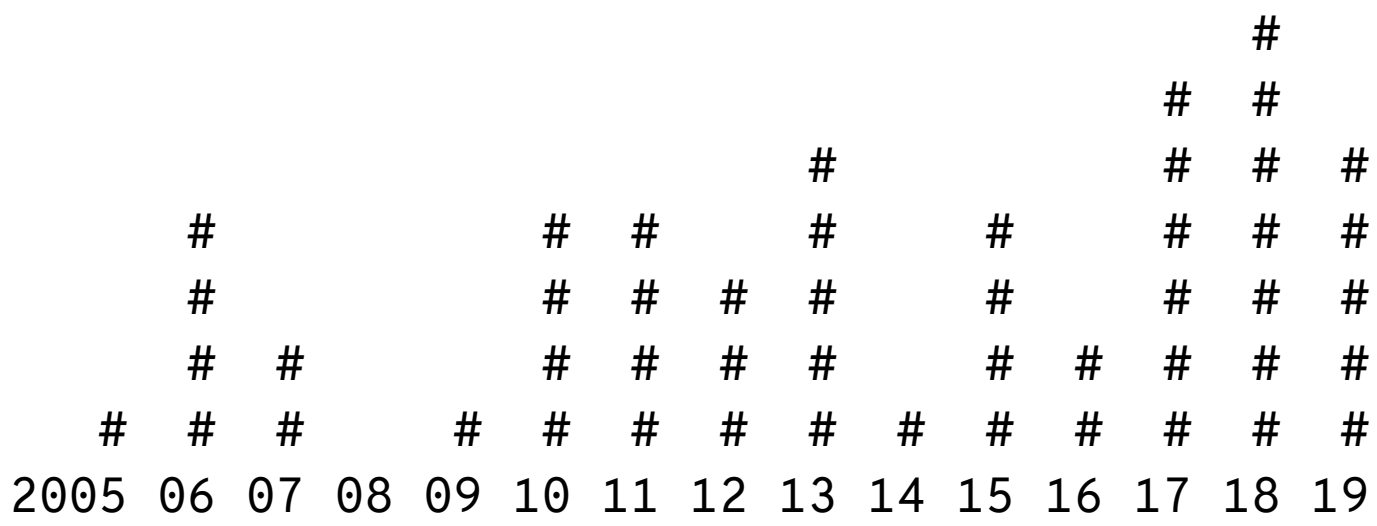
syzbot stats for ~2 years
 * 42 open bugs
 * 108 fixed bugs:
   - 21 infoleak (19 to userspace, 2 to USB)
   - 5 KVM bugs
   - 86 network bugs (16 in TIPC)

Top antipatterns

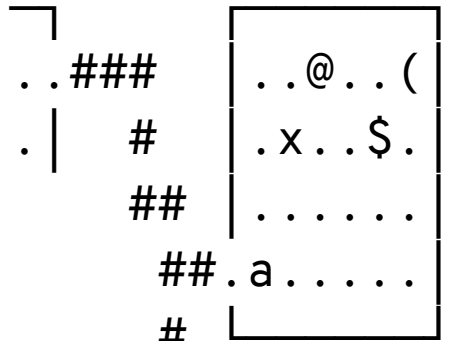
* copy part of struct sockaddr from userspace
  - treat it as a whole struct
* allocate a structure, forget to init fields/padding
  - copy it to userspace
* read registers from USB device
  - do not check that the read succeeded

Bugs lifetime

(based on 49 Fixes: tags for KMSAN bugs)



Most bugs are still there



syzbot coverage:

drivers/	–	3% of	732051
net/	–	19% of	302920
fs/	–	7% of	219699
total	–	11% of	1555946 basic blocks

attractive attack vectors are only barely scratched:

* basic IPv4/IPv6 support in syzkaller
* very limited support for USB and virtualization
* no Bluetooth, 802.11, NFC

Initialize all memory!

- What if we could always assume new memory is initialized?
- We can!

Why initialize?

* no information leaks
* deterministic execution

* By the way, Microsoft ships kernel builds with initialized local PODs since November 2018.

Initialize all stack!

Configs for stack allocations:

* GCC_PLUGIN_STRUCTLEAK_USER
  - zero-init structs marked for userspace (GCC)
* GCC_PLUGIN_STRUCTLEAK_BYREF
  - zero-init structs passed by reference (GCC)
* GCC_PLUGIN_STRUCTLEAK_BYREF_ALL
  - zero-init anything passed by reference (GCC)
* INIT_STACK_ALL
  - 0xAA-init everything on the stack (Clang)

Initialize all stack! (contd.)

"So I'd like the zeroing of local variables to be a native compiler option, so that we can (_eventually_ - these things take a long time) just start saying "ok, we simply consider stack variables to be always initialized".

Linus Torvalds.

Performance costs

* 0xAA initialization (used in the kernel now)
  - ~0% for netperf and parallel Linux build
  - 1.5% for hackbench
  - 0–4% for Android hwuimacro benchmarks
  - 7% for af_inet_loopback
* 0x00 initialization (hidden behind a Clang flag)
  - ~0% slowdown for hackbench, netperf, Linux build
  - 0–3% for end-to-end Android benchmarks
  - 4% for af_inet_loopback

Benchmarking is hard.

We can do even better

Clang is bad at dead store elimination:

* cross-basic-block DSE
* removing redundant stores at machine instruction level

Initialize all heap!

Boot parameters for heap and page_alloc (in 5.3):
  - caches with RCU and ctors are unaffected

* init_on_alloc=1 (also INIT_ON_ALLOC_DEFAULT_ON=y)
  - zero-initializes allocated memory
  - cache-friendly, noticeably faster

* init_on_free=1 (also INIT_ON_FREE_DEFAULT_ON=y)
  - zero-initializes freed memory
  - minimizes the lifetime of sensitive data
  - somewhat similar to PAX_MEMORY_SANITIZE

Performance costs

* init_on_alloc=1
  - ~0% for parallel Linux build
  - <0.5% on most Android hwuimacro benchmarks (up to 1.8%)
  - ~7% on hackbench

* init_on_free=1
  - <2% on most Android hwuimacro benchmarks (up to 5%)
  - ~7% on hackbench
  - 8% for parallel Linux build

Quotes by famous people

"Again - I don't think we want a world where everything is force-initialized. There _are_ going to be situations where that just hurts too much. But if we get to a place where we are zero-initialized by default, and have to explicitly mark the unsafe things (and we'll have comments not just about how they get initialized, but also about why that particular thing is so performance-critical), that would be a good place to be."

Linus Torvalds.

```
# halt
------
```

Backup

(Backup) Can we combine KASAN and KMSAN?

- No.

(Backup) What's wrong with shadow?

Ideally we need a 1:2 mapping from every physical and virtual address to metadata.

If a memory range can be accessed contiguously, its metadata ranges must also be contiguous.

Problem: not everything has an associated struct page

* CPU entry area
* vmalloc() memory

(Backup) What's wrong with locks?

Almost every function in the kernel may call KMSAN functions. Those, in turn, may call printk() or kmalloc().

If we already were in printk() or kmalloc(), the kernel deadlocks.

Ugly workaround: check console lock before printing.
Ugly workaround #2: don't instrument kmalloc() guts (doesn't work for transitive calls).

(Backup) The pattern controversy

```
-ftrivial-auto-var-init=pattern
 vs.
-ftrivial-auto-var-init=zero \
    -enable-trivial-auto-var-init-zero-knowing-it-will-be-
                                          removed-from-clang
```

The main concern is to avoid introducing a new C++ dialect.

(Backup) Sensitive data lifetime

```
buf1 = kmalloc(...)
write_sensitive_data(buf1);
kfree(buf1);                    # init_on_free=1 wipes buf1
buf2 = kmalloc(...)             # init_on_alloc=1 wipes buf1
```