KernelMemorySanitizer (KMSAN)

Signed-off-by: Alexander Potapenko <glider@google.com>
Dynamic Tools Team @ Google

---------------------------

Userspace tools:
* ASan, LSan, MSan, TSan, UBSan
* libFuzzer (coverage-based userspace fuzzer)
* control flow integrity in LLVM
* tens of thousands bugs in Google and opensource code

Kernel tools:
* KASAN, KMSAN, KTSAN (prototype)
* syzkaller (coverage-based kernel fuzzer)
* hundreds of bugs in the kernel(s)
MemorySanitizer (MSan)
----------------------

* around since 2012
* detects uses of uninitialized values in the userspace
* found 2000+ bugs
* works on big programs (think Chrome or server-side apps)

See also:
"MemorySanitizer: fast detector of uninitialized memory use in C++" by E. Stepanov and K. Serebryany, CGO 2015
KernelMemorySanitizer (KMSAN)
--------------------------------

* detects uses of uninitialized values in the kernel
What one might think KMSAN does
-------------------------------------

```
int a;
int b = c + a;       // report reading of uninit a
```

or:

```
int p = a;
copy_to_user(u, &p, 4);  // don't report since p is initd
```

This is useless: both false positives and false negatives!
What KMSAN actually does
-------------------------------

int a;
if (flag)
    a = 1;        // initialized
b = c + a;            // not a "use"
if (flag)
    copy_to_user(p, &b, 4);    // use: don't report
What KMSAN actually does (contd.)
---------------------------------

int x; // uninitialized
int a = x; // still uninitialized

copy_to_user(p, &a, 4); // use: report an error
KernelMemorySanitizer (KMSAN)
---------------------------------

* detects uses of uninitialized values in the kernel:
  - conditions
  - pointer dereferencing and indexing
  - values copied to the userspace, hardware etc.
Example 1
--------

```c
struct config *update_config(struct config *conf)
{
    if (!conf)
        conf = kmalloc(CONFIG_SIZE, GFP_KERNEL);
    do_update(conf);
    return conf;
}

void do_update(struct config *conf)
{
    if (conf->is_root) allow_everything(conf);
}
```
Example 2
---------

int socket_bind(int sockfd, __user struct sockaddr *uaddr,
    int ulen)
{
    struct sockaddr kaddr;
    if (ulen > sizeof(struct sockaddr) || ulen < 0)
        return -EINVAL;
    copy_from_user(&kaddr, uaddr, ulen);
    return do_bind(sockfd, &kaddr);
}
Example 3
---------

void put_dev_name_32(struct device *dev, __user char *buf)
{
    char name[32];
    strncpy(name, dev->name, 32);
    if (buf)
        copy_to_user(buf, name, 32);
}
KernelMemorySanitizer (KMSAN)
---------------------------------

* detects uses of uninitialized values in the kernel:
  - conditions
  - pointer dereferencing and indexing
  - values copied to the userspace, hardware etc.
* almost working since April 2017
* found/fixed 13 bugs (and counting)
* based on MSan
  * therefore requires Clang
KernelMemorySanitizer (KMSAN)
--------------------------------

* detects uses of uninitialized values in the kernel
  - conditions
  - pointer dereferencing and indexing
  - values copied to the userspace, hardware etc.
* almost working since April 2017
* found/fixed 13 bugs (and counting)
* based on MSan
  * therefore requires Clang
* life is too short to hack GCC `\_("/\)_/`
BUG: KMSAN: use of uninitialized memory in strlen
__msan_warning32 mm/kmsan/kmsan_instr.c:424
strlen lib/string.c:484
strlcpy lib/string.c:144
packet_bind_spkt net/packet/af_packet.c:3132
SYSC_bind net/socket.c:1370
origin:
__msan_set_alloca_origin4 mm/kmsan/kmsan_instr.c:380
SYSC_bind net/socket.c:1356
SyS_bind net/socket.c:1356
origin description: ----address@SYSC_bind
Shadow memory
-------------

Bit to bit shadow mapping
* struct page { ... struct page *shadow; ... };
* "1" means "poisoned" (uninitialized)

Uninitialized memory:
* kmalloc()
* local stack objects

Writing a constant to memory unpoisons it

Shadow is propagated through arithmetics and memory accesses
Compiler instrumentation

```
$ clang -fsanitize=kernel-memory
```

adding code that:
* poisons local variables
* handles loads and stores
* propagates shadow through arithmetic operations
* passes shadow to/from function calls
* performs shadow checks
Poisoning locals

```c
void foo() {
    int a = 1;
    char b[8];
}
```
Poisoning locals
-----------------

void foo() {
    int a = 1;
    __msan_unpoison(&a, 4);
    char b[8];
    __msan_poison_alloca(b, 8, "b");
}
Instrumenting loads and stores
--------------------------------

void copy(char *from, char *to) {
    if (!from)
        *to = -1;
    } else {
        *to = *from;
    }
}
void copy(char *from, char *to) { 
  if (!from) 
    *to = -1; 
    __msan_store_shadow_1(to, 0); 
  } else { 
    u64 shadow = __msan_load_shadow_1(from); 
    *to = *from; 
    __msan_store_shadow_1(to, shadow); 
  } 
}
Shadow propagation

0b00??1101 & 0b000011?1 is always initialized

\[ A = B + C \implies A' = B' \mid C' \]
\[ A = B \ll C \implies A' = B' \ll C \]
\[ A = B \& C \implies A' = (B' \& C') \mid (B' \& \neg C) \mid (\neg B \& C') \]

* helps to minimize the number of false positives
* somewhat similar to Valgrind, but working with SSA registers at compile time
  - we can leverage compiler optimizations
* operations are sometimes approximated for efficiency
Instrumenting function calls
---------------------------------

int sum_n(int n) {

    if (n == 0) {

        return 0;
    }
    int sum_rec = sum_n(n - 1);

    return n + sum_rec;
}
Instrumenting function calls
--------------------------------

int sum_n(int n) {
    kmsan_context_state *s = __msan_get_context_state();
    int shadow_n = s->args[0];
    if (n == 0) {
        s->ret = 0;
        return 0;
    }
    int sum_rec = sum_n(n - 1);
    s->ret = shadow_n | s->ret;
    return n + sum_rec;
}
Adding shadow checks
---------------------

```java
if (i >= 0) {
    res = a[i];
}
```
Adding shadow checks
---------------------

```c
if (__msan_load_shadow_4(&i) & INT_MIN)
    __msan_warning();
if (i >= 0) {
    if (__msan_load_shadow_4(a) ||
        __msan_load_shadow_4(&i))
        __msan_warning();
    u64 shadow = __msan_load_shadow_4(&a[i]);
    res = a[i];
    __msan_store_shadow(&res, shadow);
}
```
Tracking origins

---

```c
a = kmalloc(...);
...
b = kmalloc(...);
...
memcpy(c, b, sizeof(*b));
...
d = *a + *c;
...
if (d) ...  // Which argument is guilty in the case of UMR?
```
Tracking origins (contd.)

* when an uninit value is allocated:
  - put the stack into the stack depot (lib/stackdepot.c)
  - for each 4 bytes of allocated memory, store the 4-byte stack ID into the secondary shadow

* when the memory is copied:
  - create a new origin from the current stack and the previous origin

* when two values are used in an expression:
  - take the origin of the first uninitialized operand
Handling non-instrumented code
-----------------------------------

* asm() in *.c:
  - check that inputs are initialized
  - outputs are unpoisoned
* can't instrument around 40 files:
  - arch/x86/...
  - mm/...
  - *.S
* KMSAN_SANITIZE_filename.o := n
  - no instrumentation
  - locals, function args, return values may be dirty
Closing the gap
--------------

* `__attribute__((no_sanitize("kernel-memory")))`
  - no shadow propagation, unpoison locals and stores
* `kmsan_poison_memory()`
  - `kmalloc()`
* `kmsan_unpoison_memory()`
  - `copy_from_user()`
  - `struct pt_regs` in interrupts
  - RNGs
* `kmsan_check_memory()`
  - `copy_to_user()`
  - hardware (send to network, write to disk)
What about kmemcheck?
--------------

* When did you last run kmemcheck?
  - 1 commit fixing a bug from kmemcheck in 2017, 4 in 2014
  - 1 false positive in 2016, 1 in 2014

* Throughput in `netperf -l 30`:
  - nodebug: 39056.37
  - kasan: 5217.185
  - kmsan: 478.96 (there's still room for improvement)
  - kmemcheck: was 2000 times slower than nodebug in 2015
Long shot: taint analysis

* use shadow to indicate that a value came from an untrusted source
* use origin to mark the place where this value was obtained
* call kmsan_check_memory() at places where we expect only trusted data

# There's also another Clang tool, DFSan, which can help.
We already have instrumentation of comparison instructions and switch statements in LLVM:
* for each comparison, insert `instrument_cmp(arg1, arg2)`
* if either `arg1` or `arg2` can be found in the input [1], try to mutate that input

But the value's presence in the input doesn't guarantee the input actually affects this value!

[1] - or some `f(argi)` can be found in the input
Long shot: fuzzing assistance (contd.)

* poison each argument of each syscall and assign a unique origin to it
* for each comparison:
  if (shadow1 | shadow2)
    instrument_cmp(arg1, sh1, orig1, arg2, sh2, orig2);
* mutate only the arguments that really affect arg1 or arg2
Food for thought

CVE-2017-1000380: data race on /dev/snd/timer allows the attacker to read uninitialized heap memory.

In fact, a user with access to the device was able to e.g. read the data another user wrote into a file or socket.

Can we do something to kill all uninit bugs? (Something smarter than s/kmalloc/kzalloc ?)
Status
-----

* code at https://github.com/google/kmsan
* currently using v4.12
* x86_64 only (but nothing really arch-specific)
* requires patched Clang (will get rid of the patches soon)
* planning to upstream by the end of 2017
"That's all folks!"

-------------------
Can we combine KASAN and KMSAN?

- No.
A couple of requests
---------------------

* please don't break Clang compilation
* please don't break our userspace tools