Fighting Uninitialized Memory in the Kernel

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

```
Uninitialized memory
--------------------

- is memory that wasn't initialized after creation:
```

```
1) int i;
   if (i) { ... }

2) int *p = kmalloc(size, ...);
   copy_to_user(uptr, p, size);

3) kfree(p);
   array[*p] = q;
```

```
4) struct pair {
     char a;
     int b;
   }

   pair c = {1, 2};
   if (((char *)&c)[2]) {
     ...
   }
```

Uninitialized memory (contd.)

* C89 considers using uninitialized memory undefined behavior
  - see 6.5.7 and 7.10.3.

* The compiler may optimize the code as it wants
  - Clang does!
  - even if they don't, the result is still indeterminable.

* Attackers may still control this memory:
  - crashes;
  - information leaks;
  - privilege escalations and RCE.

MemorySanitizer (MSan)

Userspace LLVM tool that detects uses of uninitialized memory

* around since 2013
* 1:1 shadow memory to track every bit of app memory
* compiler instrumentation to update shadow
* optional origin tracking (extra 1.5x memory)

Tool needs to know about every memory access in the code,
including non-instrumented parts:

* standard libraries
* syscalls
* JIT and inline assembly

```
Shadow memory
-------------

 * huge memory mapping at 0x200000000000
 * every bit corresponds to a bit of application memory
 * 0 means initialized, 1 - uninitialized

  0??11010b | 00000011b = 0??11011b
  0??11010b & 00000011b = 00000010b

 * compile-time constants are initialized
 * malloc()ed memory is uninitialized
 * local variables are uninitialized
```

MemorySanitizer instrumentation

```
char a = *pa, b = 3;

char c = a | b;


if (c) { ... }
```

MemorySanitizer instrumentation

```
char a = *pa, b = 3;
char a_shadow = *(pa - 0x400000000000);
char b_shadow = 0x0;
char c = a | b;
char c_shadow = (a_shadow & b_shadow) |
                (a & b_shadow) | (b & a_shadow);
if (c_shadow)
    __msan_warning();
if (c) { ... }
```

MemorySanitizer instrumentation (contd.)

* copying uninits is not an error
* using them is an error:
  - conditions
  - pointer dereferencing and indexing

* TLS variables to track function parameters
  - no ABI changes

* instrumentation actually done at SSA level
  - a lot of redundant checks are deleted

KernelMemorySanitizer (KMSAN)

Kernel tool that detects uses of uninitialized memory.

* available as kernel fork since 2017, review in progress

* (almost the) same Clang instrumentation
* runtime library to create and track uninit values:
  - each struct page has two metadata pages:
    @ shadow (bit-to-bit uninitialized value tracking)
    @ origin (4-byte stack ID for every 4 uninit bytes)
  - SLUB, pagealloc and vmap hooks
  - additional checks for information leaks
    @ values copied to the userspace, network, DMA memory

```
KMSAN instrumentation
---------------------

  int a = *pa, b = 3;




  char c = a | b;



  if (c) { ... }
```

KMSAN instrumentation

```
int a = *pa, b = 3;
struct shadow_origin_ptr a_so =
    __msan_metadata_ptr_for_load_4(pa);
int a_shadow = *a_so.s;
int a_origin = *a_so.o;
int b_shadow = 0x0;
int b_origin = 0x0;
char c = a | b;
char c_shadow = (a_shadow & b_shadow) |
                (a & b_shadow) | (b & a_shadow);
char c_origin = a_shadow ? a_origin : b_origin;
if (c_shadow)
    __msan_warning(c_origin);
if (c) { ... }
```

```
Differences from MSan instrumentation
-------------------------------------

 * kernel shadow scattered across the address space
   - metadata addresses cannot be calculated inline
     @ __msan_metadata_ptr_for_load_X(ptr)
     @ __msan_metadata_ptr_for_store_X(ptr)

 * origin tracking is mandatory

 * per-task struct to track function parameters
   - calling __msan_get_context_state() in prologue

 * we can instrument (almost) everything!
   - no precompiled libraries or JIT code
```

<PLUG> Taint checking with KMSAN </PLUG>

```
 * copy_from_user() memory is now poisoned
   - treated as uninitialized
 * stack and heap allocations are unpoisoned
 * need to annotate sinks with kmsan_check_memory()
```

Any security people around?

Current KMSAN status

---

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

* code at http://github.com/google/kmsan
  - rebased on current -rc at least monthly
  - RFC v4: 42 patches, ~3KLOC
  - need more eyes!
    @ http://bit.ly/review-kmsan

Current KMSAN status (contd.)

---

* fully integrated with syzkaller
  - reports are premoderated
    @ only true positives are sent upstream
    @ unless fixed by Eric Dumazet :)
  - ~200 bugs reported so far, ~150 of them fixed

Fun fact: NetBSD has a working KMSAN implementation

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

* 58 open bugs
* 147 fixed bugs:
  - 22 infoleaks (18 to userspace, 4 to USB)
  - 93 network bugs
  - 13 bugs in USB drivers, 6 in Bluetooth, 4 in ALSA
  - 5 KVM bugs

+ 22 bugs reported manually

```
KMSAN bugs lifetime
-------------------
 (based on 79 Fixes: tags)
                                        #  #  #
                                        #  #  #
                                        #  #  #
                                        # ##  #
                            #           # ##  #
                         #  #     #     # ##  #
       #  #        #  #  #  #     #     # ##  #
       #  #        #  #  #  #     #     # ##  #
       #  #        #  #  #  #  #  #  #  # ## ##
    #  #  #  #  #  #  #  #  #  #  #  # ## ## ##
 2005 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20

 # - found since my talk at LPC in September 2019
```

```
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
     && more than 0 bytes were read
```

Most bugs are still there

```
 ─┐      ┌──────┐
..###   │..@..(│
.│   #  │.x..$.│
     ## │......│
      ##.a.....│
      #  └──────┘
      ####
```

syzbot coverage:

```
  drivers/ -  3% of  728155
  net/     - 22% of  307985
  fs/      -  7% of  220438
  total    - 10% of 1510606 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

Fun fact: Google Chrome is at 48% fuzzing coverage

Uninits are unlikely to disappear

"... the problem of leaking uninitialized kernel memory to user space is not caused merely by simple programming errors. Instead, it is deeply rooted in the nature of the C programming language, and has been around since the very early days of privilege separation in operating systems."

- Mateusz Jurczyk, Project Zero.

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.
* People have also noticed things in Apple-shipped code.

```
Initialize all stack!
---------------------

Three configs for GCC under the sky:

 * GCC_PLUGIN_STRUCTLEAK_USER
 * GCC_PLUGIN_STRUCTLEAK_BYREF
 * GCC_PLUGIN_STRUCTLEAK_BYREF_ALL

One config to rule them all:

 * INIT_STACK_ALL
   - 0xAA-init everything on the stack
   - uses Clang's -ftrivial-auto-var-init
```

Why not zero everything?

```
-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 introducing a new C++ dialect.

O brave new world!

"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.

Possible solutions

* go with a language dialect that zeroes out locals
  - perhaps only for the kernel
  - shall we have -std=linux-c on top of the base standard?

* or consistently break code that uses uninitialized locals
  - keep using the non-zero pattern
  - replace some uninits with trap values
  - improve -Wuninitialized

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
   - ~0% slowdown for hackbench, netperf, Linux build
   - 0-3% for Android benchmarks
   - 4% for af_inet_loopback

On ARM64 some benchmarks actually became faster!

```
Code size impact
-----------------

 * x86_64 defconfig:
   +0.05% image
   +0.03% .text

 * Android kernel:
   +0.6% image
   +1.3% .text



Overall size impact is low, but certain
hot functions need an extra cacheline now.
```

Can we do better?

* zero-initialization is a must
  - more compact immediates, XZR register on ARM

* Clang is bad at dead store elimination:
  - cross-basic-block DSE (MemSSA to the rescue)
  - removing redundant stores at machine instruction level
  - moving instrumentation from AST to SSA may help

* FDO and LTO.

* maybe GCC can do better?

* __attribute__((uninitialized)) for opt-out

```
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 (QEMU/x86)
   - Android hwuimacro: 0.5-1.5% (ARM64)
   - hackbench: 2.9% (ARM64), ~7% (QEMU/x86)

 * init_on_free=1
   - Android hwuimacro: 0.5-3% (ARM64)
   - hackbench: ~7% (QEMU/x86)
   - 8% for parallel Linux build
```

Can we do better?

Yes, by explicitly asking for uninitialized memory:

* __GFP_NO_AUTOINIT for kmalloc() and friends:
  - only works for init_on_alloc
  - hackbench improvement: 6.84% -> 3.45%
  - easy to abuse (like GFP_TEMPORARY and GFP_REPEAT were)
  - for small allocations compiler can emit
    kmalloc(__GFP_NO_AUTOINIT)+memset(), then apply DSE

* SLAB_NO_SANITIZE for certain slab caches:
  - will work for both init_on_alloc/init_on_free
  - easier to set up and control (e.g. at boot time)
  - done by PAX_MEMORY_SANITIZE

Opt-outs are inevitable

"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.

Bonus: ARM Memory Tagging Extension (MTE)

* Doesn't exist in hardware yet :(

* Memory tags:
  - every aligned 16 bytes have a 4-bit tag stored separately
  - every pointer has a 4-bit tag stored in the top byte
  - load/store instructions check that tags match

* "Hardware-ASAN on steroids":
  - RAM overhead: 3%-5%
  - CPU overhead: (hopefully) low-single-digit %
  - should be possible to use in production

Bonus: ARM MTE (contd.)

* need to set tags for every stack and heap allocation
  - in the very same places we're initializing them!

* one instruction to perform both initialization and tagging.

halt
