Dealing with 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;                        | 4) struct pair {
   if (i) { ... }                |      char a;
                                 |      int b;
2) int *p = kmalloc(size, ...);  |    }
   copy_to_user(uptr, p, size);  |
                                 |    pair c = {1, 2};
3) kfree(p);                     |    if (((char *)&c)[2]) {
   array[*p] = q;                |      ...
                                 |    }
```

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
  - some compilers actually do so;
  - even if they don't, the result is still indeterminable.

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

KernelMemorySanitizer (KMSAN)

A fast 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.

* https://github.com/google/kmsan/, upstreaming is WIP

KMSAN on syzbot
----

* found 240+ bugs in 2 years
  - that's ~20 CPU-years
  - 10x more machines are fuzzing KASAN

* 200+ bugs are real
  - there still are false positives and one-off errors
  - therefore bugs are premoderated
  - some code still needs annotations
    @ e.g. devices produce initialized memory

Found bugs

* 119 fixed:
    - 21 infoleak (19 userspace, 2 USB)
    - 5 KVM bugs
    - 93 network bugs (most were never reported upstream)

* 58 open (reported upstream)

* 61 in premoderation queue
    - 25 use-after-frees (KASAN duplicates)
    - 14 older than 3 months (not reproducible anymore)
    - 14 without C reproducers

* 3 with pending fixes

```
Report lifetimes
----------------

Of 119 bugs, 20 were already fixed at the time of reporting
(KMSAN is only rebased on release and -rc branches)

The remaining 99 were fixed within 291 day:
  * within 1 day:     24 ####
  * within 1 week:    60 ##########
  * within 1 month:   76 ############
  * within 3 months:  89 ###############
```

```
Error reporting rate in 2019
----------------------------

Jan: #######
Feb: ###
Mar: #######
Apr: ###
May: ###
Jun: #############
Jul: #########
Aug: ##########
Sep: ###
Oct: ################

- that's 7 bugs/month on average.
```

Bug lifetimes

(based on 53 Fixes: tags)

```
                              #
         #  #                 #              #
   #  #  #  #  #     #  #  #  #  #  #        #  @
   #  #  #  #  #  #  #  #  #  #  #  #        #  @  @
#  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  #  @  @
 1m 2m 3m 6m 1y 2y 3y 4y 5y 6y 7y 8y 9y 10 11 12 13 14 15
```

@ - these commits date back to Linux-2.6.12-rc2.

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/   -  5% of  733266
  net/       - 21%     305529
  fs/        -  8%     223039
  security/  - 13%      24910
  total      - 12%    1513040 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

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.

A: Initialize all the memory!

Q: What should we do to never have to deal
   with uninitialized memory again?

Why initialize?
----

* no information leaks;
* deterministic execution;
* may complicate use-after-free exploitation.

* By the way, Microsoft ships Windows 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_PLUGIN_STRUCTLEAK_BYREF
  - zero-init structs passed by reference
* GCC_PLUGIN_STRUCTLEAK_BYREF_ALL
  - zero-init anything passed by reference

* INIT_STACK_ALL
  - 0xAA-init everything on the stack (Clang)

Fun with Flags

Clang can also zero-initialize locals

* to initialize locals with 0xAA:
  - compile with -ftrivial-auto-var-init=pattern

* to initialize locals with 0x00:
  - compile with -ftrivial-auto-var-init=zero
  - don't forget -enable-trivial-auto-var-init-zero-knowing-it-will-be-removed-from-clang

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

We must converge

"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.

Make -ftrivial-auto-var-init=zero a thing
----

* need a similar option in GCC (see also BZ#87210);
* drop the guard flag in Clang.

* Random proposal from Clang community:
  - maybe push for -std=linux-c on top of
    the base C standard?

Performance costs (Clang)

- 0xAA initialization (used in the kernel now)
  - ~0%   - netperf and parallel Linux build
  - +1.5% - hackbench
  - +0-4% - Android hwuimacro benchmarks
  - +7%   - af_inet_loopback

- 0x00 initialization (hidden behind a Clang flag)
  - ~0%   - netperf and parallel Linux build
  - ~0%   - hackbench
  - +0-3% - Android hwuimacro benchmarks
  - +4%   - af_inet_loopback

Benchmarking is hard.

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
  - removing redundant stores at machine instruction level

- 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%   - parallel Linux build
  - <0.5% - most Android hwuimacro benchmarks (up to 1.5%)
  - ~7%   - hackbench

- init_on_free=1
  -   8% - parallel Linux build
  - <2% - most Android hwuimacro benchmarks (up to 3%)
  - ~7% - hackbench

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

------

* glider@google.com

* @Glider (mostly in Russian)

* https://github.com/google/kmsan/

* http://bit.ly/review-kmsan

(Backup) Sensitive data lifetime

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