GETTING STARTED WITH KMEMCHECK

Vegard Nossum <vegardno@ifi.uio.no>

Contents

0. Introduction

kmemcheck is a debugging feature for the Linux Kernel. More specifically, it is a dynamic checker that detects and warns about some uses of uninitialized memory.

Userspace programmers might be familiar with Valgrind's memcheck. The main difference between memcheck and kmemcheck is that memcheck works for userspace programs only, and kmemcheck works for the kernel only. The implementations are of course vastly different. Because of this, kmemcheck is not as accurate as memcheck, but it turns out to be good enough in practice to discover real programmer errors that the compiler is not able to find through static analysis.

Enabling kmemcheck on a kernel will probably slow it down to the extent that the machine will not be usable for normal workloads such as e.g. an interactive desktop. kmemcheck will also cause the kernel to use about twice as much memory as normal. For this reason, kmemcheck is strictly a debugging feature.

1. Downloading

As of version 2.6.31-rc1, kmemcheck is included in the mainline kernel.

2. Configuring and compiling

kmemcheck only works for the x86 (both 32- and 64-bit) platform. A number of configuration variables must have specific settings in order for the kmemcheck menu to even appear in "menuconfig". These are:

  o CONFIG_CC_OPTIMIZE_FOR_SIZE=n

```
        This option is located under "General setup" / "Optimize for size".

        Without this, gcc will use certain optimizations that usually lead to
        false positive warnings from kmemcheck. An example of this is a 16-bit
        field in a struct, where gcc may load 32 bits, then discard the upper
        16 bits. kmemcheck sees only the 32-bit load, and may trigger a
        warning for the upper 16 bits (if they're uninitialized).

  o CONFIG_SLAB=y or CONFIG_SLUB=y

        This option is located under "General setup" / "Choose SLAB
        allocator".

  o CONFIG_FUNCTION_TRACER=n

        This option is located under "Kernel hacking" / "Tracers" / "Kernel
        Function Tracer"

        When function tracing is compiled in, gcc emits a call to another
        function at the beginning of every function. This means that when the
        page fault handler is called, the ftrace framework will be called
        before kmemcheck has had a chance to handle the fault. If ftrace then
        modifies memory that was tracked by kmemcheck, the result is an
        endless recursive page fault.

  o CONFIG_DEBUG_PAGEALLOC=n

        This option is located under "Kernel hacking" / "Debug page memory
        allocations".

In addition, I highly recommend turning on CONFIG_DEBUG_INFO=y. This is also
located under "Kernel hacking". With this, you will be able to get line number
information from the kmemcheck warnings, which is extremely valuable in
debugging a problem. This option is not mandatory, however, because it slows
down the compilation process and produces a much bigger kernel image.

Now the kmemcheck menu should be visible (under "Kernel hacking" / "kmemcheck:
trap use of uninitialized memory"). Here follows a description of the
kmemcheck configuration variables:

  o CONFIG_KMEMCHECK

        This must be enabled in order to use kmemcheck at all...

  o CONFIG_KMEMCHECK_[DISABLED | ENABLED | ONESHOT]_BY_DEFAULT

        This option controls the status of kmemcheck at boot-time. "Enabled"
        will enable kmemcheck right from the start, "disabled" will boot the
        kernel as normal (but with the kmemcheck code compiled in, so it can
        be enabled at run-time after the kernel has booted), and "one-shot" is
        a special mode which will turn kmemcheck off automatically after
        detecting the first use of uninitialized memory.

        If you are using kmemcheck to actively debug a problem, then you
        probably want to choose "enabled" here.
```

The one-shot mode is mostly useful in automated test setups because it can prevent floods of warnings and increase the chances of the machine surviving in case something is really wrong. In other cases, the one-shot mode could actually be counter-productive because it would turn itself off at the very first error -- in the case of a false positive too -- and this would come in the way of debugging the specific problem you were interested in.

If you would like to use your kernel as normal, but with a chance to enable kmemcheck in case of some problem, it might be a good idea to choose "disabled" here. When kmemcheck is disabled, most of the run-time overhead is not incurred, and the kernel will be almost as fast as normal.

o CONFIG_KMEMCHECK_QUEUE_SIZE

Select the maximum number of error reports to store in an internal (fixed-size) buffer. Since errors can occur virtually anywhere and in any context, we need a temporary storage area which is guaranteed not to generate any other page faults when accessed. The queue will be emptied as soon as a tasklet may be scheduled. If the queue is full, new error reports will be lost.

The default value of 64 is probably fine. If some code produces more than 64 errors within an irqs-off section, then the code is likely to produce many, many more, too, and these additional reports seldom give any more information (the first report is usually the most valuable anyway).

This number might have to be adjusted if you are not using serial console or similar to capture the kernel log. If you are using the "dmesg" command to save the log, then getting a lot of kmemcheck warnings might overflow the kernel log itself, and the earlier reports will get lost in that way instead. Try setting this to 10 or so on such a setup.

o CONFIG_KMEMCHECK_SHADOW_COPY_SHIFT

Select the number of shadow bytes to save along with each entry of the error-report queue. These bytes indicate what parts of an allocation are initialized, uninitialized, etc. and will be displayed when an error is detected to help the debugging of a particular problem.

The number entered here is actually the logarithm of the number of bytes that will be saved. So if you pick for example 5 here, kmemcheck will save 2^5 = 32 bytes.

The default value should be fine for debugging most problems. It also fits nicely within 80 columns.

o CONFIG_KMEMCHECK_PARTIAL_OK

This option (when enabled) works around certain GCC optimizations that produce 32-bit reads from 16-bit variables where the upper 16 bits are thrown away afterwards.

        The default value (enabled) is recommended. This may of course hide
        some real errors, but disabling it would probably produce a lot of
        false positives.

  o CONFIG_KMEMCHECK_BITOPS_OK

        This option silences warnings that would be generated for bit-field
        accesses where not all the bits are initialized at the same time. This
        may also hide some real bugs.

        This option is probably obsolete, or it should be replaced with
        the kmemcheck-/bitfield-annotations for the code in question. The
        default value is therefore fine.

Now compile the kernel as usual.

3. How to use

3.1. Booting

First some information about the command-line options. There is only one option specific to kmemcheck, and this is called "kmemcheck". It can be used to override the default mode as chosen by the CONFIG_KMEMCHECK_*_BY_DEFAULT option. Its possible settings are:

- o kmemcheck=0 (disabled)
- o kmemcheck=1 (enabled)
- o kmemcheck=2 (one-shot mode)

If SLUB debugging has been enabled in the kernel, it may take precedence over kmemcheck in such a way that the slab caches which are under SLUB debugging will not be tracked by kmemcheck. In order to ensure that this doesn't happen (even though it shouldn't by default), use SLUB's boot option "slub_debug", like this: slub_debug=-

In fact, this option may also be used for fine-grained control over SLUB vs. kmemcheck. For example, if the command line includes "kmemcheck=1 slub_debug=,dentry", then SLUB debugging will be used only for the "dentry" slab cache, and with kmemcheck tracking all the other caches. This is advanced usage, however, and is not generally recommended.

3.2. Run-time enable/disable

When the kernel has booted, it is possible to enable or disable kmemcheck at run-time. WARNING: This feature is still experimental and may cause false positive warnings to appear. Therefore, try not to use this. If you find that it doesn't work properly (e.g. you see an unreasonable amount of warnings), I will be happy to take bug reports.

Use the file /proc/sys/kernel/kmemcheck for this purpose, e.g.:

```
$ echo 0 > /proc/sys/kernel/kmemcheck # disables kmemcheck
```

The numbers are the same as for the kmemcheck= command-line option.

3.3. Debugging

A typical report will look something like this:

```
WARNING: kmemcheck: Caught 32-bit read from uninitialized memory
(ffff88003e4a2024)
80000000000000000000000000000000000000000088ffff0000000000000000
 i i i i u u u u i i i i i i i i u u u u u u u u u u u u u u u u
         ^

Pid: 1856, comm: ntpdate Not tainted 2.6.29-rc5 #264 945P-A
RIP: 0010:[<ffffffff8104ede8>]  [<ffffffff8104ede8>] __dequeue_signal+0xc8/0x190
RSP: 0018:ffff88003cdf7d98  EFLAGS: 00210002
RAX: 0000000000000030 RBX: ffff88003d4ea968 RCX: 0000000000000009
RDX: ffff88003e5d6018 RSI: ffff88003e5d6024 RDI: ffff88003cdf7e84
RBP: ffff88003cdf7db8 R08: ffff88003e5d6000 R09: 0000000000000000
R10: 0000000000000080 R11: 0000000000000000 R12: 000000000000000e
R13: ffff88003cdf7e78 R14: ffff88003d530710 R15: ffff88003d5a98c8
FS:  0000000000000000(0000) GS:ffff880001982000(0063) knlGS:00000
CS:  0010 DS: 002b ES: 002b CR0: 0000000080050033
CR2: ffff88003f806ea0 CR3: 000000003c036000 CR4: 00000000000006a0
DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
DR3: 0000000000000000 DR6: 00000000ffff4ff0 DR7: 0000000000000400
 [<ffffffff8104f04e>] dequeue_signal+0x8e/0x170
 [<ffffffff81050bd8>] get_signal_to_deliver+0x98/0x390
 [<ffffffff8100b87d>] do_notify_resume+0xad/0x7d0
 [<ffffffff8100c7b5>] int_signal+0x12/0x17
 [<ffffffffffffffff>] 0xffffffffffffffff
```

The single most valuable information in this report is the RIP (or EIP on 32-bit) value. This will help us pinpoint exactly which instruction that caused the warning.

If your kernel was compiled with CONFIG_DEBUG_INFO=y, then all we have to do is give this address to the addr2line program, like this:

```
$ addr2line -e vmlinux -i ffffffff8104ede8
arch/x86/include/asm/string_64.h:12
include/asm-generic/siginfo.h:287
kernel/signal.c:380
kernel/signal.c:410
```

The "-e vmlinux" tells addr2line which file to look in. IMPORTANT: This must be the vmlinux of the kernel that produced the warning in the first place! If not, the line number information will almost certainly be wrong.

The "-i" tells addr2line to also print the line numbers of inlined functions. In this case, the flag was very important, because otherwise, it would only have printed the first line, which is just a call to memcpy(), which could be called from a thousand places in the kernel, and is therefore not very useful. These inlined functions would not show up in the stack trace above, simply because the kernel doesn't load the extra debugging information. This technique can of course be used with ordinary kernel oopses as well.

In this case, it's the caller of memcpy() that is interesting, and it can be found in include/asm-generic/siginfo.h, line 287:

```
281 static inline void copy_siginfo(struct siginfo *to, struct siginfo *from)
282 {
283         if (from->si_code < 0)
284                 memcpy(to, from, sizeof(*to));
285         else
286                 /* _sigchld is currently the largest know union member */
287                 memcpy(to, from, __ARCH_SI_PREAMBLE_SIZE +
sizeof(from->_sifields._sigchld));
288 }
```

Since this was a read (kmemcheck usually warns about reads only, though it can warn about writes to unallocated or freed memory as well), it was probably the "from" argument which contained some uninitialized bytes. Following the chain of calls, we move upwards to see where "from" was allocated or initialized, kernel/signal.c, line 380:

```
359 static void collect_signal(int sig, struct sigpending *list, siginfo_t
*info)
360 {
...
367         list_for_each_entry(q, &list->list, list) {
368                 if (q->info.si_signo == sig) {
369                         if (first)
370                                 goto still_pending;
371                         first = q;
...
377         if (first) {
378 still_pending:
379                 list_del_init(&first->list);
380                 copy_siginfo(info, &first->info);
381                 __sigqueue_free(first);
...
392         }
393 }
```

Here, it is &first->info that is being passed on to copy_siginfo(). The variable "first" was found on a list -- passed in as the second argument to collect_signal(). We continue our journey through the stack, to figure out where the item on "list" was allocated or initialized. We move to line 410:

```
395 static int __dequeue_signal(struct sigpending *pending, sigset_t *mask,
396                         siginfo_t *info)
397 {
...
410                 collect_signal(sig, pending, info);
...
414 }
```

Now we need to follow the "pending" pointer, since that is being passed on to collect_signal() as "list". At this point, we've run out of lines from the "addr2line" output. Not to worry, we just paste the next addresses from the kmemcheck stack dump, i.e.:

```
 [<ffffffff8104f04e>] dequeue_signal+0x8e/0x170
 [<ffffffff81050bd8>] get_signal_to_deliver+0x98/0x390
 [<ffffffff8100b87d>] do_notify_resume+0xad/0x7d0
 [<ffffffff8100c7b5>] int_signal+0x12/0x17

	$ addr2line -e vmlinux -i ffffffff8104f04e ffffffff81050bd8 \
		ffffffff8100b87d ffffffff8100c7b5
	kernel/signal.c:446
	kernel/signal.c:1806
	arch/x86/kernel/signal.c:805
	arch/x86/kernel/signal.c:871
	arch/x86/kernel/entry_64.S:694
```

Remember that since these addresses were found on the stack and not as the RIP value, they actually point to the _next_ instruction (they are return addresses). This becomes obvious when we look at the code for line 446:

```
422 int dequeue_signal(struct task_struct *tsk, sigset_t *mask, siginfo_t *info)
423 {
...
431                 signr = __dequeue_signal(&tsk->signal->shared_pending,
432                                          mask, info);
433                 /*
434                  * itimer signal ?
435                  *
436                  * itimers are process shared and we restart periodic
437                  * itimers in the signal delivery path to prevent DoS
438                  * attacks in the high resolution timer case. This is
439                  * compliant with the old way of self restarting
440                  * itimers, as the SIGALRM is a legacy signal and only
441                  * queued once. Changing the restart behaviour to
442                  * restart the timer in the signal dequeue path is
443                  * reducing the timer noise on heavy loaded !highres
444                  * systems too.
445                  */
446                 if (unlikely(signr == SIGALRM)) {
...
489 }
```

So instead of looking at 446, we should be looking at 431, which is the line that executes just before 446. Here we see that what we are looking for is &tsk->signal->shared_pending.

Our next task is now to figure out which function that puts items on this "shared_pending" list. A crude, but efficient tool, is git grep:

```
	$ git grep -n 'shared_pending' kernel/
	...
	kernel/signal.c:828:	pending = group ? &t->signal->shared_pending :
&t->pending;
	kernel/signal.c:1339:	pending = group ? &t->signal->shared_pending :
```

```
&t->pending;
        ...
```

There were more results, but none of them were related to list operations, and these were the only assignments. We inspect the line numbers more closely and find that this is indeed where items are being added to the list:

```
816 static int send_signal(int sig, struct siginfo *info, struct task_struct *t,
817                         int group)
818 {
...
828         pending = group ? &t->signal->shared_pending : &t->pending;
...
851         q = __sigqueue_alloc(t, GFP_ATOMIC, (sig < SIGRTMIN &&
852                                              (is_si_special(info) ||
853                                               info->si_code >= 0)));
854         if (q) {
855                 list_add_tail(&q->list, &pending->list);
...
890 }
```

and:

```
1309 int send_sigqueue(struct sigqueue *q, struct task_struct *t, int group)
1310 {
....
1339          pending = group ? &t->signal->shared_pending : &t->pending;
1340          list_add_tail(&q->list, &pending->list);
....
1347 }
```

In the first case, the list element we are looking for, "q", is being returned from the function __sigqueue_alloc(), which looks like an allocation function. Let's take a look at it:

```
187 static struct sigqueue *__sigqueue_alloc(struct task_struct *t, gfp_t flags,
188                                          int override_rlimit)
189 {
190         struct sigqueue *q = NULL;
191         struct user_struct *user;
192
193         /*
194          * We won't get problems with the target's UID changing under us
195          * because changing it requires RCU be used, and if t != current,
the
196          * caller must be holding the RCU readlock (by way of a spinlock)
and
197          * we use RCU protection here
198          */
199         user = get_uid(__task_cred(t)->user);
200         atomic_inc(&user->sigpending);
201         if (override_rlimit ||
202             atomic_read(&user->sigpending) <=
203                         t->signal->rlim[RLIMIT_SIGPENDING].rlim_cur)
204                 q = kmem_cache_alloc(sigqueue_cachep, flags);
205         if (unlikely(q == NULL)) {
```

```
                atomic_dec(&user->sigpending);
                free_uid(user);
        } else {
                INIT_LIST_HEAD(&q->list);
                q->flags = 0;
                q->user = user;
        }

        return q;
}
```

We see that this function initializes q->list, q->flags, and q->user. It seems that now is the time to look at the definition of "struct sigqueue", e.g.:

```
struct sigqueue {
        struct list_head list;
        int flags;
        siginfo_t info;
        struct user_struct *user;
};
```

And, you might remember, it was a memcpy() on &first->info that caused the warning, so this makes perfect sense. It also seems reasonable to assume that it is the caller of __sigqueue_alloc() that has the responsibility of filling out (initializing) this member.

But just which fields of the struct were uninitialized? Let's look at kmemcheck's report again:

```
WARNING: kmemcheck: Caught 32-bit read from uninitialized memory
(ffff88003e4a2024)
80000000000000000000000000000000000000000088ffff0000000000000000
 i i i i u u u u i i i i i i i i u u u u u u u u u u u u u u u u
         ^
```

These first two lines are the memory dump of the memory object itself, and the shadow bytemap, respectively. The memory object itself is in this case &first->info. Just beware that the start of this dump is NOT the start of the object itself! The position of the caret (^) corresponds with the address of the read (ffff88003e4a2024).

The shadow bytemap dump legend is as follows:

- i - initialized
- u - uninitialized
- a - unallocated (memory has been allocated by the slab layer, but has not yet been handed off to anybody)
- f - freed (memory has been allocated by the slab layer, but has been freed by the previous owner)

In order to figure out where (relative to the start of the object) the uninitialized memory was located, we have to look at the disassembly. For that, we'll need the RIP address again:

```
RIP: 0010:[<ffffffff8104ede8>]  [<ffffffff8104ede8>] __dequeue_signal+0xc8/0x190
```

```
        $ objdump -d --no-show-raw-insn vmlinux | grep -C 8 ffffffff8104ede8:
        ffffffff8104edc8:       mov    %r8,0x8(%r8)
        ffffffff8104edcc:       test   %r10d,%r10d
        ffffffff8104edcf:       js     ffffffff8104ee88 <__dequeue_signal+0x168>
        ffffffff8104edd5:       mov    %rax,%rdx
        ffffffff8104edd8:       mov    $0xc,%ecx
        ffffffff8104eddd:       mov    %r13,%rdi
        ffffffff8104ede0:       mov    $0x30,%eax
        ffffffff8104ede5:       mov    %rdx,%rsi
        ffffffff8104ede8:       rep movsl %ds:(%rsi),%es:(%rdi)
        ffffffff8104edea:       test   $0x2,%al
        ffffffff8104edec:       je     ffffffff8104edf0 <__dequeue_signal+0xd0>
        ffffffff8104edee:       movsw  %ds:(%rsi),%es:(%rdi)
        ffffffff8104edf0:       test   $0x1,%al
        ffffffff8104edf2:       je     ffffffff8104edf5 <__dequeue_signal+0xd5>
        ffffffff8104edf4:       movsb  %ds:(%rsi),%es:(%rdi)
        ffffffff8104edf5:       mov    %r8,%rdi
        ffffffff8104edf8:       callq  ffffffff8104de60 <__sigqueue_free>
```

As expected, it's the "rep movsl" instruction from the memcpy() that causes the warning. We know about REP MOVSL that it uses the register RCX to count the number of remaining iterations. By taking a look at the register dump again (from the kmemcheck report), we can figure out how many bytes were left to copy:

```
RAX: 0000000000000030 RBX: ffff88003d4ea968 RCX: 0000000000000009
```

By looking at the disassembly, we also see that %ecx is being loaded with the value $0xc just before (ffffffff8104edd8), so we are very lucky. Keep in mind that this is the number of iterations, not bytes. And since this is a "long" operation, we need to multiply by 4 to get the number of bytes. So this means that the uninitialized value was encountered at 4 * (0xc - 0x9) = 12 bytes from the start of the object.

We can now try to figure out which field of the "struct siginfo" that was not initialized. This is the beginning of the struct:

```
typedef struct siginfo {
        int si_signo;
        int si_errno;
        int si_code;

        union {
..
        } _sifields;
} siginfo_t;
```

On 64-bit, the int is 4 bytes long, so it must the the union member that has not been initialized. We can verify this using gdb:

```
        $ gdb vmlinux
        ...
        (gdb) p &((struct siginfo *) 0)->_sifields
        $1 = (union {...} *) 0x10
```

Actually, it seems that the union member is located at offset 0x10 -- which means that gcc has inserted 4 bytes of padding between the members si_code and _sifields. We can now get a fuller picture of the memory dump:

```
         _----------------------------=> si_code
        /        _--------------------=> (padding)
       |        /        _------------=> _sifields(._kill._pid)
       |       |        /        _----=> _sifields(._kill._uid)
       |       |       |        /
-------|-------|-------|-------|
80000000000000000000000000000000000000000088ffff0000000000000000
 i i i i u u u u i i i i i i i i u u u u u u u u u u u u u u u u
```

This allows us to realize another important fact: si_code contains the value 0x80. Remember that x86 is little endian, so the first 4 bytes "80000000" are really the number 0x00000080. With a bit of research, we find that this is actually the constant SI_KERNEL defined in include/asm-generic/siginfo.h:

```
#define SI_KERNEL       0x80            /* sent by the kernel from somewhere
    */
```

This macro is used in exactly one place in the x86 kernel: In send_signal() in kernel/signal.c:

```
static int send_signal(int sig, struct siginfo *info, struct task_struct *t,
                        int group)
{
...
        pending = group ? &t->signal->shared_pending : &t->pending;
...
        q = __sigqueue_alloc(t, GFP_ATOMIC, (sig < SIGRTMIN &&
                                             (is_si_special(info) ||
                                              info->si_code >= 0)));
        if (q) {
                list_add_tail(&q->list, &pending->list);
                switch ((unsigned long) info) {
...
                case (unsigned long) SEND_SIG_PRIV:
                        q->info.si_signo = sig;
                        q->info.si_errno = 0;
                        q->info.si_code = SI_KERNEL;
                        q->info.si_pid = 0;
                        q->info.si_uid = 0;
                        break;
...
}
```

Not only does this match with the .si_code member, it also matches the place we found earlier when looking for where siginfo_t objects are enqueued on the "shared_pending" list.

So to sum up: It seems that it is the padding introduced by the compiler between two struct fields that is uninitialized, and this gets reported when we do a memcpy() on the struct. This means that we have identified a false positive warning.

Normally, kmemcheck will not report uninitialized accesses in memcpy() calls when both the source and destination addresses are tracked. (Instead, we copy the shadow bytemap as well). In this case, the destination address clearly was not tracked. We can dig a little deeper into the stack trace from above:

```
        arch/x86/kernel/signal.c:805
        arch/x86/kernel/signal.c:871
        arch/x86/kernel/entry_64.S:694
```

And we clearly see that the destination siginfo object is located on the stack:

```
782 static void do_signal(struct pt_regs *regs)
783 {
784         struct k_sigaction ka;
785         siginfo_t info;
...
804         signr = get_signal_to_deliver(&info, &ka, regs, NULL);
...
854 }
```

And this &info is what eventually gets passed to copy_siginfo() as the destination argument.

Now, even though we didn't find an actual error here, the example is still a good one, because it shows how one would go about to find out what the report was all about.

3.4. Annotating false positives

There are a few different ways to make annotations in the source code that will keep kmemcheck from checking and reporting certain allocations. Here they are:

- o __GFP_NOTRACK_FALSE_POSITIVE

  This flag can be passed to kmalloc() or kmem_cache_alloc() (therefore also to other functions that end up calling one of these) to indicate that the allocation should not be tracked because it would lead to a false positive report. This is a "big hammer" way of silencing kmemcheck; after all, even if the false positive pertains to particular field in a struct, for example, we will now lose the ability to find (real) errors in other parts of the same struct.

  Example:

  ```
      /* No warnings will ever trigger on accessing any part of x */
      x = kmalloc(sizeof *x, GFP_KERNEL | __GFP_NOTRACK_FALSE_POSITIVE);
  ```

- o kmemcheck_bitfield_begin(name)/kmemcheck_bitfield_end(name) and kmemcheck_annotate_bitfield(ptr, name)

  The first two of these three macros can be used inside struct definitions to signal, respectively, the beginning and end of a bitfield. Additionally, this will assign the bitfield a name, which is given as an argument to the macros.

Having used these markers, one can later use kmemcheck_annotate_bitfield() at the point of allocation, to indicate which parts of the allocation is part of a bitfield.

Example:

```
struct foo {
    int x;

    kmemcheck_bitfield_begin(flags);
    int flag_a:1;
    int flag_b:1;
    kmemcheck_bitfield_end(flags);

    int y;
};

struct foo *x = kmalloc(sizeof *x);

/* No warnings will trigger on accessing the bitfield of x */
kmemcheck_annotate_bitfield(x, flags);
```

Note that kmemcheck_annotate_bitfield() can be used even before the return value of kmalloc() is checked -- in other words, passing NULL as the first argument is legal (and will do nothing).

4. Reporting errors

As we have seen, kmemcheck will produce false positive reports. Therefore, it is not very wise to blindly post kmemcheck warnings to mailing lists and maintainers. Instead, I encourage maintainers and developers to find errors in their own code. If you get a warning, you can try to work around it, try to figure out if it's a real error or not, or simply ignore it. Most developers know their own code and will quickly and efficiently determine the root cause of a kmemcheck report. This is therefore also the most efficient way to work with kmemcheck.

That said, we (the kmemcheck maintainers) will always be on the lookout for false positives that we can annotate and silence. So whatever you find, please drop us a note privately! Kernel configs and steps to reproduce (if available) are of course a great help too.

Happy hacking!

5. Technical description

kmemcheck works by marking memory pages non-present. This means that whenever somebody attempts to access the page, a page fault is generated. The page fault handler notices that the page was in fact only hidden, and so it calls on the kmemcheck code to make further investigations.

When the investigations are completed, kmemcheck "shows" the page by marking it present (as it would be under normal circumstances). This way, the interrupted code can continue as usual.

But after the instruction has been executed, we should hide the page again, so that we can catch the next access too! Now kmemcheck makes use of a debugging feature of the processor, namely single-stepping. When the processor has finished the one instruction that generated the memory access, a debug exception is raised. From here, we simply hide the page again and continue execution, this time with the single-stepping feature turned off.

kmemcheck requires some assistance from the memory allocator in order to work. The memory allocator needs to

1. Tell kmemcheck about newly allocated pages and pages that are about to be freed. This allows kmemcheck to set up and tear down the shadow memory for the pages in question. The shadow memory stores the status of each byte in the allocation proper, e.g. whether it is initialized or uninitialized.

2. Tell kmemcheck which parts of memory should be marked uninitialized. There are actually a few more states, such as "not yet allocated" and "recently freed".

If a slab cache is set up using the SLAB_NOTRACK flag, it will never return memory that can take page faults because of kmemcheck.

If a slab cache is NOT set up using the SLAB_NOTRACK flag, callers can still request memory with the __GFP_NOTRACK or __GFP_NOTRACK_FALSE_POSITIVE flags. This does not prevent the page faults from occurring, however, but marks the object in question as being initialized so that no warnings will ever be produced for this object.

Currently, the SLAB and SLUB allocators are supported by kmemcheck.