LINUX PLUMBERS CONFERENCE September 20-24, 2021

Adding kernel-specific test coverage to GCC's -fanalyzer option

David Malcolm <dmalcolm@redhat.com>
Carlos O'Donell <carlos@redhat.com>

Overview

- Condensed version of Monday's talk:
  - What is **-fanalyzer** ?
  - Demo of detecting kernel CVEs
- Discussion

Caveat

- I’m a compiler developer not a kernel developer!
  - I’m hoping for input from kernel experts on this

What is **-fanalyzer** ?

- A new interprocedural GCC pass (added in GCC 10)
  - Only useful for C code at the moment
- Performs a much more expensive analysis of the code than traditional GCC warnings
  - GCC 10: 15 warnings, mostly relating to malloc/free
  - GCC 11: 7 more, for 22 warnings, plus plugin support; big rewrite of internals
  - GCC 12 (in development): 1 more (uninit values), for 23 warnings, plus working on kernel-specific warnings
- Neither sound nor complete: can have **false negatives** and **false positives**
  - Various heuristics to **try** to explore all paths through the code whilst terminating in a reasonable time (merging some states, keeping others distinct)
  - Various approximations: of state, and of “shortest feasible path”

Looking at historical kernel CVEs

- What can I extend the analyzer to detect?
  - Infoleaks (information disclosure)
    - Uninitialized kernel memory being copied to user space
    - Relatively easy to detect, relatively low severity (mitigated by new **-ftrivial-auto-var-init** option in GCC 12)
  - Taint (data from untrusted source used at trusting sink)
    - e.g. user-space/network data used as array index/allocation size
    - Harder to detect, relatively higher importance (denial of service, privilege escalation, etc)

Infoleak detection (1): CVE-2017-18549

```
#define   AAC_SENSE_BUFFERSIZE 30
struct aac_srb_reply
{
    __le32 status;
    __le32 srb_status;
    __le32 scsi_status;
    __le32 data_xfer_length;
    __le32 sense_data_size;
    u8  sense_data[AAC_SENSE_BUFFERSIZE];
};
```

Infoleak detection (2): CVE-2017-18549

```
static int aac_send_raw_srb(/* [...snip...] */, void __user *user_reply)
{
    /* [...snip...] */

    struct aac_srb_reply reply;

    reply.status = ST_OK;
    /* [...snip...] */
    reply.srb_status = SRB_STATUS_SUCCESS;
    reply.scsi_status = 0;
    reply.data_xfer_length = byte_count;
    reply.sense_data_size = 0;
    memset(reply.sense_data, 0, AAC_SENSE_BUFFERSIZE);

    if (copy_to_user(user_reply, &reply, sizeof(struct aac_srb_reply))) {
       ..etc...
}
```

Infoleak detection (3): CVE-2017-18549

```
infoleak-CVE-2017-18549-1.c: In function ‘aac_send_raw_srb’:
infoleak-CVE-2017-18549-1.c:66:13: warning: potential exposure of sensitive information by copying uninitialized data from
stack across trust boundary [CWE-200] [-Wanalyzer-exposure-through-uninit-copy]
   66 |         if (copy_to_user(user_reply, &reply, sizeof(struct aac_srb_reply))) {
      |             ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ‘aac_send_raw_srb’: events 1-3
    |
    |   52 |         struct aac_srb_reply reply;
    |      |                              ^~~~~
    |      |                              |
    |      |                              (1) source region created on stack here
    |      |                              (2) capacity: 52 bytes
    |......
    |   66 |         if (copy_to_user(user_reply, &reply, sizeof(struct aac_srb_reply))) {
    |      |             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    |      |             |
    |      |             (3) uninitialized data copied from stack here
    |
```

Infoleak detection (4): CVE-2017-18549

```
infoleak-CVE-2017-18549-1.c:66:13: note: 2 bytes are uninitialized
   66 |         if (copy_to_user(user_reply, &reply, sizeof(struct aac_srb_reply))) {
      |             ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
infoleak-CVE-2017-18549-1.c:37:25: note: padding after field ‘sense_data’ is
uninitialized (2 bytes)
   37 |         u8              sense_data[AAC_SENSE_BUFFERSIZE];
      |                         ^~~~~~~~~~
infoleak-CVE-2017-18549-1.c:52:30: note: suggest forcing zero-initialization by
providing a ‘{0}’ initializer
   52 |         struct aac_srb_reply reply;
      |                              ^~~~~
      |                                    = {0}
```

Taint detection (1)
CVE 2011-0521

```
/* Example edited for brevity.  */
struct ca_slot_info_t {
    int num; /* slot number */
    ca_slot_info_t   ci_slot[2];
} sbuf;
if (copy_from_user(&sbuf, (void __user *)arg, sizeof(sbuf)) != 0)
  return -1;
ca_slot_info_t *info= &sbuf;
if (info->num > 1)
  return -EINVAL;
av7110->ci_slot[info->num].num = info->num;
/* ...etc...  */
```

Taint detection (2)
CVE 2011-0521 (cont'd)

```
taint-CVE-2011-0521.c: In function 'test_1':
taint-CVE-2011-0521.c:321:40: warning: use of attacker-controlled value '*info.num' in array lookup
 without checking for negative [CWE-129] [-Wanalyzer-tainted-array-index]
  321 |         av7110->ci_slot[info->num].num = info->num;
      |         ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~
  'test_1': events 1-5
    |
    |  310 |         if (copy_from_user(&sbuf, (void __user *)arg, sizeof(sbuf)) != 0)
    |      |            ^
    |      |            |
    |      |            (1) following 'false' branch...
    |......
    |  313 |         struct dvb_device *dvbdev = file->private_data;
    |      |                           ~~~~~~
    |      |                           |
    |      |                           (2) ...to here
```

Taint detection (3)
CVE 2011-0521 (cont'd)

```
    |......
    |  318 |         if (info->num > 1)
    |      |            ~
    |      |            |
    |      |            (3) following 'false' branch...
    |......
    |  321 |         av7110->ci_slot[info->num].num = info->num;
    |      |         ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    |      |                                 |          |
    |      |                                 |          (5) use of attacker-controlled value
'*info.num' in array lookup without checking for negative
    |      |                                 (4) ...to here
    |
```

Marking trust boundaries

```
extern long copy_to_user(void __user *to, const void *from, unsigned long n)
  __attribute__((access (untrusted_write, 1, 3),
                 access (read_only, 2, 3)));
extern long copy_from_user(void *to, const void __user *from, long n)
  __attribute__((access (write_only, 1, 3),
                 access (untrusted_read, 2, 3)));

#define __SYSCALL_DEFINEx(x, name, ...)   \
     asmlinkage __attribute__((tainted))        \
     long sys##name(__SC_DECL##x(__VA_ARGS__))

struct configfs_attribute {
     /* … */
     ssize_t (*store)(struct config_item *, const char *, size_t) __attribute__((tainted));
};
```

Integration testing

- Can we detect problems when using the system kernel headers?
- antipatterns.ko – the world's worst kernel module?
  - https://github.com/davidmalcolm/antipatterns.ko
  - Ideas/patches for other tests most welcome

-fanalyzer on the kernel

- I have an automated script to build a custom GCC, and the build the kernel using it
- Takes about **4 hours** to build a kernel with **-fanalyzer** on a fast machine
- Running it on Fedora, RHEL, and upstream kernels
  - Fixing false positives
- Found an issue in “allyesconfig” upstream kernel

Current Status

- **Infoleak detection:**
  - not yet in GCC trunk, but mostly ready to go in, but:
    - What should syntax be?
    - Where should code live?
- **Taint detection:**
  - I’m still working on this; hope to have it done by GCC 12 feature freeze
    - Similar syntax/scope considerations apply

Topics we could talk about

- Is this useful?
  - Any ideas on improvements to output format?
  - Ideas for other things to test for?
- Is this useful for dependability and assurance?
  - (given false positives and false negatives)
- How to integrate this into kernel development workflow?
- What else do people want to talk about?

More Info

- Project homepage: https://gcc.gnu.org/wiki/DavidMalcolm/StaticAnalyzer
- Thanks for listening/participating!
- Thanks to LPC for hosting us

Bonus slides

- (taken from Monday’s talk at the GNU tools track)

Internal Implementation

- Builds an “exploded graph” combining control flow and data flow
- Nodes in this graph have both:
  - Program point (CFG location and call stack)
  - State

Internal Implementation (2)

- State at a node includes:
  - Symbolic memory regions with symbolic values
    - e.g. “global variable ‘g’ has value 42”
  - Constraints on symbolic values
    - e.g. “INIT_VAL(i) < INIT_VAL(n)”
  - State machines:
    - Per-value
      - heap: e.g. “this is a freed pointer”
      - taint: “this value is unsanitized and attacker-controlled”
    - Global: “are we in a signal handler?”

Internal Implementation (3)

- Neither sound nor complete: can have false negatives and false positives
- Diagnostics are:
  - Captured at nodes
  - De-duplicated
  - Checked for feasibility (path conditions)
  - Expressed to the user using paths through the code

GCC 10: 15 new warnings

- **-Wanalyzer-double-free**
- **-Wanalyzer-use-after-free**
- **-Wanalyzer-free-of-non-heap**
- **-Wanalyzer-malloc-leak**

- **-Wanalyzer-possible-null-argument**
- **-Wanalyzer-possible-null-dereference**
- **-Wanalyzer-null-argument**
- **-Wanalyzer-null-dereference**

-Wanalyzer-double-fclose
-Wanalyzer-file-leak

-Wanalyzer-stale-setjmp-buffer
-Wanalyzer-use-of-pointer-in-stale-stack-frame

-Wanalyzer-unsafe-call-within-signal-handler

-Wanalyzer-tainted-array-index

-Wanalyzer-exposure-through-output-file

GCC 11: 5 new warnings

- **-Wanalyzer-mismatching-deallocation**
  - **__attribute__((malloc, “what_frees_this”))**
- **-Wanalyzer-shift-count-negative**
- **-Wanalyzer-shift-count-overflow**
- **-Wanalyzer-write-to-const**
- **-Wanalyzer-write-to-string-literal**

GCC 11: plugin support

- Plugins can extend the analyzer, allowing domain-specific path-sensitive warnings.
- Example (from testsuite): checking for misuses of CPython's global interpreter lock

GCC 11: plugin support (2)

```
gil-1.c: In function ‘test_2’:
gil-1.c:16:3: warning: use of PyObject ‘*obj’ without the GIL
   16 |   Py_INCREF (obj);
      |   ^~~~~~~~~
  ‘test_2’: events 1-2
    |
    |   14 |   Py_BEGIN_ALLOW_THREADS
    |      |   ^~~~~~~~~~~~~~~~~~~~~~
    |      |   |
    |      |   (1) releasing the GIL here
    |   15 |
    |   16 |   Py_INCREF (obj);
    |      |   ~~~~~~~~~
    |      |   |
    |      |   (2) PyObject ‘*obj’ used here without the GIL
    |
```

Buffer overflow detection?

- Experimented with implementing this
- **-fanalyzer** in trunk (for GCC 12) now:
  - captures the sizes of dynamic allocations as symbolic values (e.g “extents (*ptr) == (N * 8) + 64”)
  - has a consistent place for adding diagnostics about memory accesses (reads and writes)
  - But...

Buffer overflow detection (2)

- I tried verifying that all memory accesses are within bounds
- Is this access:
  - Known to be fully within bounds?
  - Known to be (at least partially) outside bounds?
  - Unknown if fully within bounds?

Buffer overflow detection (3)

- “What are the symbolic conditions that hold for this memory access to be valid?”
  - Known valid
  - Known invalid: report
    - should I implement this?
  - Unknown: what to do?
    - “**warning**: possible out-of-bounds write to ‘**arr[i]**’ when ‘**i >= n**’ or ‘**i < 0**’”
    - ...but maybe that can’t happen

Buffer overflow detection (4)

- Too many false positives: a wall of noise
- Insight: can an attacker influence this?
  - Revisit of taint detection
    - What are the “trust boundaries” in the code?
    - What is the “attack surface” of the code?