Static Detection of Unsafe DMA Accesses in Device Drivers

In USENIX Security 2021

Jia-Ju Bai\textsuperscript{1}, Tuo Li\textsuperscript{1}, Kangjie Lu\textsuperscript{2}, Shi-Min Hu\textsuperscript{1}

\textsuperscript{1}Tsinghua University, \textsuperscript{2}University of Minnesota

https://baijjiaju.github.io/
Background

- DMA is widely used in modern device drivers
  - Direct data transfer between hardware registers and system memory
  - Perform data transfer without CPU involvement
DMA access

Basic steps

- S1: Create a DMA buffer
- S2: Perform a DMA access like a regular variable access
  - Read a DMA buffer: data = dma_buf->data;
  - Write a DMA buffer: dma_buf->data = data;
- S3: Delete a DMA buffer
DMA type

- Streaming DMA buffer
  - It is asynchronously available to both the CPU and hardware device
  - The driver needs to explicitly synchronize the data between hardware registers and CPU cache
  - Each DMA access is relatively cheap

- Coherent DMA buffer
  - It is simultaneously available to both the CPU and hardware device
  - The driver does not need to explicitly synchronize the data between hardware registers and CPU cache
  - Each DMA access is relatively expensive
Security risks of DMA access

Streaming DMA access
- After a streaming DMA buffer is created, the driver should not access the content of this buffer, until this buffer is unmapped
- The driver is allowed to access buffer content during synchronization with hardware registers and CPU cache

Security risks of violations
- Inconsistent DMA access
- Data inconsistency between hardware registers and CPU cache
Inconsistent DMA access in the Linux rtl8192ce driver

- Introduced in Linux 4.4 (released in Jan. 2016)
- Fixed in Oct. 2020 by us

```c
FILE: linux-5.6/drivers/net/wireless/realtek/rtlwifi/rtl8192ce/trx.c
522. void rtl92ce_tx_fill_cmddesc(...) {
      ......  
      // Streaming DMA mapping
531.    dma_addr_t mapping = pci_map_single(..., skb->data, ...);
      ......  
535.    struct iee80211_hdr *hdr = (struct iee80211_hdr *)(skb->data);
536     fc = hdr->frame_control;  // Inconsistent DMA access!
      ......  
584. }
```
Security risks of DMA access

- Coherent DMA access
  - The hardware device can be untrusted, and thus can write bad data into coherent DMA buffers, which are used by the driver
  - The driver should perform correct validation of the data from DMA buffers before using the data

- Security risks of violations
  - Unchecked DMA access
  - Security bugs, such as buffer overflow and invalid-pointer access
Unchecked DMA access in the Linux vmxnet3 driver

- Introduced in Linux 3.16 (released in Aug. 2014)
- Fixed in Jun. 2020 by us
Unsafe DMA access

- Basic rules

  - **dma_addr = dma_map_single(buf)**
    - Accessing the content of buf is forbidden!
  
  - **dma_sync_single_for_cpu(dma_addr)**
    - Accessing the content of buf is allowed!
  
  - **dma_sync_single_for_device(dma_addr)**
    - Accessing the content of buf is forbidden!
  
  - **dma_unmap_single(dma_addr)**

  Streaming DMA access

  - **dma_buf = dma_alloc_coherent(...)**
    - Data in dma_buf should be correctly validated!
  
  - **Use data in dma_buf**

  Coherent DMA access
Challenges of detecting unsafe DMA access

C1: Identifying DMA access
- Each DMA access is implemented as a regular variable access, without calling specific interface functions
- DMA creation and DMA access often have no explicit execution order from static code observation, namely in a *broken control flow*

C2: Checking the safety of DMA access
- Accuracy and efficiency of analyzing large OS code

C3: Dropping false positives
- Validating code-path feasibility is difficult and expensive
Key techniques

- **C1: Identifying DMA access**
  - *Field-based alias analysis* to effectively identify DMA access

- **C2: Checking the safety of DMA accesses**
  - *Flow-sensitive and pattern-based analysis* to accurately and efficiently check the safety of DMA access

- **C3: Dropping false positives**
  - *Efficient code-path validation method* to drop false positives and reduce the overhead of using a SMT solver
DMA-access identification

- **S1: Handling DMA-buffer creation**
  - Identify DMA-creation function calls
  - Collect the information about their return variables, including variable names, data structure types and fields

- **S2: Identifying DMA access**
  - Check each variable access in the driver
  - If variable name or data structure information matches the collected information, the access is identified to be a DMA access

- Alias analysis is useful to handling variable assignments
  - Intra-procedural, flow-insensitive and Andersen-style alias analysis
## DMA-access safety checking

### Checking streaming DMA access

- Four patterns about DMA operations
- Forward and backward flow-sensitive analysis

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Code Snippet</th>
<th>Analysis Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern 1</td>
<td><code>dma_addr = dma_map_single(buf) // Start</code>&lt;br&gt;<code>Read or write the content of buf // Report!</code></td>
<td>Forward flow-sensitive analysis</td>
</tr>
<tr>
<td>Pattern 2</td>
<td><code>dma_sync_single_for_device(dma_addr) // Start</code>&lt;br&gt;<code>Read or write the content of buf // Report!</code></td>
<td>Forward flow-sensitive analysis</td>
</tr>
<tr>
<td>Pattern 3</td>
<td><code>dma_unmap_single(dma_addr) // Start</code>&lt;br&gt;<code>Read or write the content of buf // Report!</code></td>
<td>Backward flow-sensitive analysis</td>
</tr>
<tr>
<td>Pattern 4</td>
<td><code>dma_sync_single_for_cpu(dma_addr) // Start</code>&lt;br&gt;<code>Read or write the content of buf // Report!</code></td>
<td>Backward flow-sensitive analysis</td>
</tr>
</tbody>
</table>
DMA-access safety checking

- Checking coherent DMA access
  - Flow-sensitive taint analysis to identify DMA-affected operations
  - Three patterns about security problems

FILE: `linux-5.6/drivers/net/wireless/intel/iwlwifi/pcie/rx.c`

1693. static u32 iwl_pcie_int_cause_ict(...) {
    1714.     ... do {
    1722.         read = trans_pcie->ict_tbl[...];
    1725.     } while (read); // Possible bug
    1743. }

2054. int iwl_pcie_alloc_ict(...) {
    ....... // Coherent DMA allocation
    2058.     trans_pcie->ict_tbl = dma_alloc_coherent(...);
    2071. }

Pattern 1: Infinite loop polling

FILE: `linux-5.6/drivers/net/wireless/intel/iwlwifi/pcie/pcie/rx.c`

2661. static void __ipw2100_rx_process(...) {
    2701.     // MASK is 0x0f
    2710.     frame_type = sq->drv[i].status_fields & MASK;
    2765.     // Possible bug
    4318.     static int status_queue_allocate(...) {
    4325.         // Coherent DMA allocation
    4334. }

Pattern 2: Buffer overflow

FILE: `linux-5.6/drivers/net/ethernet/socionext/netsec.c`

931. static int netsec_process_rx(...) {
    948. struct netsec_de *de = dring->vaddr + ...;
    971.     pkt_len = de->buf_len_info >> 16;
    1059. }

Pattern 3: Invalid pointer access
Code-Path Validation

○ S1: Getting path constraints
  • Translate each instruction in the code path to a Z3 constraint
  • *Example:* “a = b + c” -> “a == b + c”

○ S2: Adding additional constraints
  • Identify and add constraints that can trigger security bugs
  • *Example:* For buffer overflow, add “frame > MAX_SIZE” when frame
    is an index to access an array whose bound is MAX_SIZE

○ S3: Solving all constraints
  • If the constraints cannot be satisfied, the possible unsafe DMA
    access is identified as a false positive and is dropped
Approach

- **SADA** *(S)tatic (A)nalysis of (D)MA (A)ccess)*
  - Integrate the three key techniques
  - Statically detect unsafe DMA access in device drivers
  - LLVM-based static analysis
Evaluation

Driver code in Linux 5.6

- Use a regular PC with eight CPUs and 16GB memory
- Use Clang-9.0
- Make `allyesconfig` of x86-64
- Check the kernel directories `drivers/` and `sound/`
# Evaluation

## Detection of unsafe DMA accesses

<table>
<thead>
<tr>
<th>Description</th>
<th>Linux 5.6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code handling</strong></td>
<td></td>
</tr>
<tr>
<td>Source files (.c)</td>
<td>14.6K</td>
</tr>
<tr>
<td>Source code lines</td>
<td>8.8M</td>
</tr>
<tr>
<td><strong>DMA-access identification</strong></td>
<td></td>
</tr>
<tr>
<td>Encountered DMA-buffer creation</td>
<td>2,781</td>
</tr>
<tr>
<td>DMA buffers in data structure fields</td>
<td>2,074</td>
</tr>
<tr>
<td>Identified DMA accesses</td>
<td>28,732</td>
</tr>
<tr>
<td><strong>DMA-access checking</strong></td>
<td></td>
</tr>
<tr>
<td>Unsafe DMA accesses (real / all)</td>
<td>284 / 321</td>
</tr>
<tr>
<td>Inconsistent DMA accesses (real / all)</td>
<td>123 / 131</td>
</tr>
<tr>
<td>Unchecked DMA accesses (real / all)</td>
<td>161 / 190</td>
</tr>
<tr>
<td><strong>Time usage</strong></td>
<td></td>
</tr>
<tr>
<td>DMA-access identification</td>
<td>62m</td>
</tr>
<tr>
<td>DMA-access checking</td>
<td>208m</td>
</tr>
<tr>
<td>Total time</td>
<td>270m</td>
</tr>
</tbody>
</table>
Evaluation

- 123 inconsistent DMA accesses
  - Direct access after DMA creation: 108
  - Incorrect DMA synchronization: 15

- 161 unchecked DMA accesses
  - Buffer overflow: 121
  - Invalid-pointer access: 36
  - Infinite loop polling: 4

- 105 of the 284 real unsafe DMA accesses have been confirmed by driver developers
Limitations

- **False positives**
  - The current alias analyses is simple and not accurate enough
  - The path validation can make mistakes in complex cases
  - .......

- **False negatives**
  - Lack the analysis of function-pointer calls
  - Neglect other patterns of unsafe DMA accesses
  - .......
Conclusion

- DMA is popular in modern device drivers but can introduce security risks in practice

- SADA: static detection of unsafe DMA accesses
  - *Field-based alias analysis* to effectively identify DMA accesses
  - *Flow-sensitive and pattern-based analysis* to accurately and efficiently check the safety of DMA accesses
  - *Efficient code-path validation method* to drop false positives and reduce the overhead of using SMT solvers

- Find 284 real unsafe DMA accesses in Linux 5.6
Thanks for listening!

Jia-Ju Bai
E-mail: baijiaju@tsinghua.edu.cn
https://baijiaju.github.io/