CIV Classification and Static Analyses

This document describes the current attack classification and our approach on detecting the attacks.

TOC

Threat Model

- Assume driver isolation (memory isolation).
- Data synchronization is selective: Only shared states are selectively synchronized across the boundary, based on the access attributes computed by static analysis.
- Assume driver contains bugs allowing the attacker to perform arbitrary R/W in the driver compartment. But don't assume the driver can perform arbitrary calls (less powerful then arbitrary code execution).
- Boundary flow integrity is assumed.

Question

Should we assume pointer swizzling (KSplit/RLBox) or pointer authentication (PAC)?

Seems some attacks on pointer can be prevented by this, e.g., attacker corrupts a pointer's value, and this corrupted value will not have corresponding pointer in the lookup table, and causing attack failure.

Attack Summary Table

Attacks	Static Analysis
Memory safety	(Taint analysis)
– *MemV1: Dereference corrupted pointers*	– Taint sink is pointer dereference in the kernel.
– *MemV2: Calling memory API with corrupted parameters*	– Taint sink is a list of kernel memory helper APIs, such as memcpy.
– *MemV3: Accessing buffer with corrupted index*	– Taint sink is a buffer access
Control data corruption	(Taint analysis)
– *ControlV1: Corrupt loop condition*	Taint sink is at the loop condition
– *ControlV2: Corrupt other branch condition*	Taint sink is at branch condition (can customize finer filtering, e.g., locating branches contain sensitive operations)
Concurrency primitive corruption	

Attacks	Static Analysis
- *ConcurrentV1: Lock under conditions*	- Compute unpaired lock/unlock calls under conditions - Trace the location of such calls
- *ConcurrentV2: Loop in critical section*	1. Identify shared critical section locations 2. Identify loop within shared critical sections. 3. Trace the location of loop in CS
Type confusion	1. Compute shared union and multicast pointer passed from driver to kernel 2. Trace the location uses multicast data
Protocol Violation (PV)	
- *PV1: Breaking kernel interface call orders*	- Analyze kernel interface functions called by driver under conditions - Analyze driver exported functions, obtain the call sites of the driver functions
- *PV2: Violating lifetime (reference counting)*	- Analyze direct/indirect called reference counting API from the driver
- *PV3: return value corruption*	- Identify the the call sites of driver call back functions. - Generate a warning for the return value location, allowing manual checks for potential risky usage.
Semantic violation (SV)	

Attacks	Static Analysis
- *SV1: Panic condition*	- Detect BugOn location that use shared data
- *SV2: Data structure invariants*	- No analysis yet. Potentially requires some inferences. Such as checks voting etc.
- *SV3: Divided by zero*	- Taint analysis, the taint source is shared states - Taint sink is divide operation within the kernel
TOCTOU	No analysis.

Shared Data Corruption - Memory Safety

Description

Kernel code contains various kinds of memory operations.

- Memory access through pointer.
  - Pointer dereference
- Memory API calls, such as memcpy.
- Array access.

These kinds of memory operations may use corrupted data supplied from the driver.

- For memory access through pointer, the pointer may be corrupted by the driver. Yet isolation framework maybe able to prevent such corruption through pointer swizzling.
- Parameters to memory API calls
  - src/dst pointers
  - size parameters
- Array access: buffer size.

Resulting Attacks

- Pointer Dereference
  - Dereference a nullptr causes kernel crash (DoS).
  - Dereference an illegal pointer, readi (DoS).
- Memory API Calls
  - Allocate large object, occupy memory and cause DoS.
- Buffer Access
  - Out of bound access with OOB buffer index

Static analysis

Analysis: taint analysis.

Taint analysis setup:

- **Taint source:**
  - kernel isolation interface boundary parameters
  - kernel read locations of shared states
- **Taint sink:**
  - Memory access through pointers (pointer dereference)
  - Memory API calls
  - Array access
- **Taint propagation**: data flow

Attack Examples

Pointer Dereference

```
// Driver caller
nr_pages = radix_tree_gang_lookup(&nullb->dev->cache,(void **)c_pages,
nullb->cache_flush_pos, FREE_BATCH);
```

```
// Kernel risky location (interface function)
radix_tree_gang_lookup(const struct radix_tree_root *root, void **results,
                       unsigned long first_index, unsigned int max_items) {
```

```
        radix_tree_for_each_slot(slot, root, &iter, first_index) {
                results[ret] = rcu_dereference_raw(*slot);
                if (!results[ret])
                ...
        }
        ...
}
```

All the parameters are directly passed from driver to kernel. While the max_items is a constant, the results can be corrupted. The result can be illegal memory access or nullptr dereference.

Memory API Calls

```
// Driver caller
static inline int ixgbe_init_rss_key(struct ixgbe_adapter *adapter)
{
        netdev_rss_key_fill(rss_key, IXGBE_RSS_KEY_SIZE);
        adapter->rss_key = rss_key;
        ...
}
```

```
// Kernel
void netdev_rss_key_fill(void *buffer, size_t len)
{
```

```
        BUG_ON(len > sizeof(netdev_rss_key));
        net_get_random_once(netdev_rss_key, sizeof(netdev_rss_key));
        memcpy(buffer, netdev_rss_key, len);
}
```

Here, we assume that all the data from driver can be corrupted, even for constant. So both buffer and len are corrupted, can result in buffer overflow attack.

Buffer Access

```
// Driver caller
static int ixgbe_fwd_ring_up(struct ixgbe_adapter *adapter,
                             struct ixgbe_fwd_adapter *accel) {
        ...
        netdev_unbind_sb_channel(adapter->netdev, vdev);
        ...
}
```

```
// Kernel
void netdev_unbind_sb_channel(struct net_device *dev,
                              struct net_device *sb_dev)
{
        struct netdev_queue *txq = &dev->_tx[dev->num_tx_queues];
```

```
        ...
}
```

Here, the driver calls the **netdev_unbind_sb_channel** function, and it can corrupt the vdev→num_tx_queue field, resulting a buffer overflow, causing illegal memory access, can crashing the kernel.

Attack Cases

Driver Sok Memory Corruption Attack Cases

Shared Data Corruption – Control Data Corruption

Description

Shared control data refers to the data used in kernel branch conditions. By corrupting such data, the attacker can control or divert the control flow in the core kernel. A basic example is corrupting conditions of if/else, and causing the kernel to diver on a different path.

Static Analysis

Analysis: taint analysis.
Taint analysis setup:

- **Taint source:**
  - kernel isolation interface boundary parameters
  - kernel read locations of shared states
- **Taint sink:**
  - Branch conditions
    - loop
    - if/else
    - switch
- **Taint propagation**: data flow

Note: In many cases, the branch are actually safe checks. These branches simply return earlier, preventing possible faulty or corrupted values. Thus, it's important to prioritize the checks for branches that are not safe checks.

Attack Example

```
// Driver
static struct pci_dev *ixgbe_get_first_secondary_devfn(unsigned int devfn)
{
        struct pci_dev *rp_pdev;
        rp_pdev = pci_get_domain_bus_and_slot(0, 0, devfn);
```

```
        if (rp_pdev && rp_pdev->subordinate) {
                bus = rp_pdev->subordinate->number;
                pci_dev_put(rp_pdev);
                return pci_get_domain_bus_and_slot(0, bus, 0);
        }

        pci_dev_put(rp_pdev);
        return NULL;
}
```

```
// Kernel
void pci_dev_put(struct pci_dev *dev)
{
        if (dev)
                put_device(&dev->dev);
}
```

This is a reference count, but the dev here can be corrupted. The dev can be a corrupted object, with the dev→dev field points to an unexpected object, causing the kernel to decrease the wrong reference counter, and result in early release.

Attack Cases

Driver Sok Control Data Corruption Attack Cases

ConcurrentV1: Lock Under Conditions

Description

Some locks can be called under certain conditions. Suppose the condition uses attacker corrupted data, the lock call can be skipped. In that case, some atomic operations may be running in non-atomic context.

Static Analysis

1. Find the call sites of lock instructions.
2. Check if the call site is control dependent on any conditions.
3. For lock call sites that are dependent on some conditions, check if there are corresponding unlock call site that are control dependent on the condition.
    1. If no, then the lock call is considered an unpaired lock call site guarded by a condition.
4. generate warnings for all the conditional unpaired lock call site.

Attack Example

Attack Cases

Driver Sok DeadLock Attack Cases

ConcurrentV2: DeadLock with infinite loop

Description

If a critical section includes a loop, such as a for/while loop or a linked list traversal, it can potentially lead to a deadlock if the loop fails to terminate. This deadlock situation can be exploited by an attacker who has control over the control variable or can manipulate the linked list structure, for example, by creating a cycle within the linked list.

Note: I do think a corrupted linked list itself also posts threat. But the difference is that the infinite loop would be within driver, and the driver will hang. But with the deadlock, the driver can cause kernel to have a deadlock situation.

Static Analysis

1. Find all the critical section using shared locks.
2. For each critical section CS, compute all the instructions in the CS>

3. For each instruction in the CS, check if there is a loop, e.g., a while loop or some macro like iterating through linked list.
4. For critical sections that contain loop struct, generate warnings for the critical sections.

Attack Example

```
// Driver
static void alx_netif_stop(struct alx_priv *alx)
{
        int i;
        netif_trans_update(alx->dev);
        if (netif_carrier_ok(alx->dev)) {
                ...
                netif_tx_disable(alx->dev);
        }
}
// header function
static inline void netif_tx_disable(struct net_device *dev)
{
        ...
        spin_lock(&dev->tx_global_lock);
        for (i = 0; i < dev->num_tx_queues; i++) {
                struct netdev_queue *txq = netdev_get_tx_queue(dev, i);
```

```
                __netif_tx_lock(txq, cpu);
                netif_tx_stop_queue(txq);
                __netif_tx_unlock(txq);
        }
        spin_unlock(&dev->tx_global_lock);
        ...
}
```

In this example, the driver can invoke the function netif_tx_disable, which obtain a **tx_global_lock**, which is a shared global lock.
Suppose the driver can corrupt the num_tx_queues, e.g., setting it to a large number, it can cause deadlock (DoS) in the kernel as kernel will be waiting for this lock to be released by the driver.

Attack Cases

[Driver Sok DeadLock Attack Cases]

Protocol Violation

Description

Driver follows implicit protocols in its lifetime. The protocol we define here is a state machine of function transition between the driver and the kernel during the driver's lifetime. The following is the canonical driver protocol.

We define driver in various states:

1. Loaded
2. Unloaded
3. Initialized
4. Uninitialized
5. Active
6. Idle
7. Inactive

Events:

1. insmod (unloaded → loaded)
2. rmmod (loaded → unloaded)
3. modle_init+probe (loaded → initialized)
4. uninit (initialized →)
5. execute call back (idle → active)
6. interrupt (no transit)
7. module_exit (idle → unloaded)

PV1: Protocol Violation – Breaking Kernel Interface Call Ordering

Description

We do not assume the attacker has arbitrary code execution capability within the driver itself. However, we do assume the adversary can achieve arbitrary read/write access to the driver's memory space. Via this memory corruption, the attacker can tamper with key control variables governing execution flow and data values in the driver code.

Consequently, when a kernel thread enters the driver to invoke some intended functionality, the corrupted variables may redirect it to follow an unexpected code path chosen by the attacker instead of the legitimate path. This malicious control flow could then induce the driver to invoke kernel functions outside its normal protocol

Static Analysis

1. Compute all the kernel interface function called by the driver as **KI**.
2. For each function **f** in KI, if f is under a condition in the driver, add f to a set KCI.
3. Output KCI and the corresponding call sites.

Attack Example

```
// Driver
static void ixgbe_shutdown(struct pci_dev *pdev)
{
        bool wake;

        __ixgbe_shutdown(pdev, &wake);

        if (system_state == SYSTEM_POWER_OFF) {
                pci_wake_from_d3(pdev, wake);
                pci_set_power_state(pdev, PCI_D3hot);
        }
}
```

Here, the kernel interface function **pci_wake_from_d3** and **pci_set_power_state** are called under a condition in the driver. Driver may corrupt the system_state, and causing the driver to transit to a different power state. (need verification)

Attack Cases

Driver Sok Protocol Violation Attack Cases

PV2: Protocol Violation – Breaking Object Lifetime Protocol

Description

The adversary's objective is to corrupt a reference counter associated with a kernel object. This can be achieved by improperly increasing or decreasing its value. As explained in this article , while excessive increments can be detected, excessive decrements are more difficult to catch.

The attacker's ultimate goal is to identify a path that reaches a ref_inc or ref_dec call operating on the target reference counter, and manipulate control flow to decrease its value below zero. This has the effect of prematurely freeing the object even if references remain.

There are two primary data types used for reference counting in the Linux kernel – atomic_t and refcount_t. The former is a legacy type still actively used across current kernel versions. Hence, when examining reference counting APIs, both need to be considered as potential targets.

Static Analysis

Assumptions

1. we assume a reference counter is a field within the counted object.

2. we assume a reference counter API can be directly/indirectly accessed by a malicious driver.
    - for the path reaches in driver, we call this directly manipulatable reference counter
    - for the path reaches transitively from the kernel, we call this indirectly manipulatable reference counter.
3. we first analyze the directly manipulatable reference counter API, and then the indirect ones.
4. indirect Ref API should require an extra step, which is to analyze whether the checks for the atomic variables can be controlled.

Analysis Steps
For direct reference counting:

1. Compute all driver functions and stored in set **DrvFuncs**.
2. For each function **f** in DrvFuncs, compute all the calls made by f in set **Call**.
3. For each call in Call, check if the call is an invocation of reference counting API on atomic_t or refcount_t. If true, store the call to a set **DirectRefAPI**.
4. Output all the calls in DirectRefAPI.

Attack Example

```
// Driver function
netdev_tx_t ixgbe_xmit_frame_ring(struct sk_buff *skb,
                                  struct ixgbe_adapter *adapter,
                                  struct ixgbe_ring *tx_ring)
{
        ...
        adapter->ptp_tx_skb = skb_get(skb);
        ...
}
```

```
// Kernel function
static inline struct sk_buff *skb_get(struct sk_buff *skb)
{
        refcount_inc(&skb->users);
        return skb;
}
```

The driver can corrupt the skb→users field, causing a different object to have wrong reference counting.
In wrost case, the skb pointer points to a completely corrupted object, allowing the driver to corrupt the reference counting on any kernel objects.

Attack Cases

PV3: Protocol Violation – Breaking Driver Callback Return Value

Description

By corrupting key variables and logic governing error handling flows, the adversary can manipulate the driver's behavior to trick the kernel. We examine four prevalent error handling attack vectors:

1. Removing error checks allows fabricated success signals to pass back to the kernel even when devices fail. The kernel perceives correctness while issuing flawed I/O and state mutations.
2. Overwriting true error return codes disguises device failures as successes to downstream kernel logic. The kernel interprets completion and continues improper state transitions.
3. Preventing de-allocation of error recovery resources exhausts kernel memory pools. Repeated starved allocations degrade system stability over time.

Attack Example

× **Missing return value attack example**

Example

Attack Cases

Driver Sok Protocol Violation Attack Cases

Semantic Violation

General Description

Kernel and driver obeys certain semantic invariants on the shared states. The semantics can be explicit or implicit. The explicit ones are expressed in kernel with some mechanisms such as the BugOn macro. The BUG_ON() macro in Linux kernel is used to intentionally crash the kernel and generate a kernel panic when a condition is true at runtime. This acts as a way to assert certain conditions that should never happen.

There are also some implicit semantics that are not checked rigorously. These implicit invariants rely on assumptions that the kernel developer makes about valid state values or transitions. Violations of these assumptions can cause unexpected behavior rather than deterministic crashes.

For example, the kernel may expect a “status” flag to transition monotonically from initial value 0 through 1, 2 and finally to state 3. However, if a buggy driver sets the status to 4, the kernel may not explicitly check for and handle this case. The lack of validation on the implicit invariant that status must progress in a certain defined order can result in subtler and more difficult to detect bugs.

SV1: Semantic Violation – Panic conditions

Description

Kernel uses the BugOn macro to ensure data follows certain semantics. Attacker may cause kernel crashes by abusing such conditions.

Static Analysis

Taint analysis.

Taint sink: Calls on BugOn macro.

1. Identify all BugOn location that may use corrupted shared states.
2. Generate warnings for these locations.

Attack Example

```
static void ixgbe_remove(struct pci_dev *pdev)
{
        ...
        free_netdev(netdev);
        ...
}
```

```
// Kernel
void free_netdev(struct net_device *dev)
{
        ...
        BUG_ON(dev->reg_state != NETREG_UNREGISTERED);
        dev->reg_state = NETREG_RELEASED;
        ...
}
```

The driver can corrupt the dev→reg_state field, allowing the driver to trigger the Bugon location, and causing kernel panic.

Attack Cases

Driver Sok Semantic Violation Attack Cases

SV2: Semantic Violation – Data Structure Invariant

Description

Need more discussion here. It's hard to infer DS invariants. Many of these are captured by BugOn before the DS is used. But maybe there are some missing pieces.

Static Analysis

No analysis yet, this require some level of inference.

Attack Example

× **Need DS structure invariant attack example**

Example

Attack Cases

Driver Sok Semantic Violation Attack Cases

Time of Check Time of Use

Description

Time-of-check to time-of-use (TOCTOU) vulnerabilities arise from race conditions between validating/checking data and subsequently using that data. Specifically, there is a gap between when data is checked or validated, and when that data is actually used.

In an isolated driver and kernel scenario, TOCTOU bugs are possible because the driver and kernel share access to certain state data structures or variables. The kernel may check or validate elements of a shared data structure, then perform additional operations before actually using or accessing the now validated data. An attacker in control of the isolated driver could potentially modify the shared state in that gap between validation and use by the kernel.

However, realistically exploiting this race condition requires precise synchronization between the attacker actions and victim kernel operations. With only basic read/write capabilities, an attacker would likely find it difficult to forcibly invoke kernel synchronization at an exact moment required to introduce malicious data between check and use.

Therefore, while data isolation gaps between kernel and driver introduce potential TOCTOU vulnerabilities, practical exploitation depends on amplification of the race window. We should focus our analysis on scenarios

that substantially widen that gap through additional synchronization or timing control beyond basic access capabilities.

Static Analysis

We don't deploy static analysis for directly finding TOCTOU bugs. Instead, we provide a static analysis to reveal checks on shared states passed in non-atomic context. Ideally, all the use of shared states should be accessed under atomic context, e.g., within a critical section. Otherwise, a corrupted driver may manage to find a way to launch TOCTOU attacks.

1. Compute a set of shared states S.
2. Compute the use locations of states in S in the kernel domain as S_Use, using data dependency graph.
3. Compute the checks S_Check on S_Use using control dependency graph.
4. Use atomic region analysis to check if the S_Use and the S_Check are in the same atomic region.

Attack Example

× **Need TOCTOU attack example**

Example

Attack Cases