WIP: QKSAN: Towards Multiple Sanitizers for In-vehicle COTS OS Kernels

Yalong Zou
Huazhong University of Science and Technology

Ziqiu Cheng
Huazhong University of Science and Technology

Dongliang Mu
Huazhong University of Science and Technology

Abstract

With the rise of smart vehicles, increasingly intelligent in-vehicle systems are also exposing security issues, where traditional software vulnerabilities are demonstrating greater harm. Fuzzing is still an effective means to mitigate the risks of system software vulnerabilities. However, due to the fact that in-vehicle commercial off-the-shelf (COTS) system software is typically closed-source, conducting fuzzing on it presents significant challenges. The lack of information and the difficulty of modifying the system make it hard to implement effective fuzzing oracles.

To address these issues, we propose QKSAN, a sanitizer framework suitable for binary-only kernels. QKSAN innovatively combines multiple types of sanitizers and employs an efficient hypervisor-level instrumentation method to detect memory violation bugs such as out-of-bound accesses and the use of uninitialized variables. Experiments have demonstrated that QKSAN can successfully detect various vulnerabilities in binary kernels like Linux and QNX and feasibly be applied to real-world systems fuzzing.

1 Introduction

Recent advancements in smart vehicles and Internet of Vehicles (IoV) technologies have greatly enhanced connectivity among the various components within automobiles. However, this enhanced intelligence and interconnectivity have simultaneously broadened the attack surface, thereby introducing new security vulnerabilities and threats to in-vehicle COTS operating systems [12, 27, 28]. Vendors have introduced security devices (e.g. gateways) to protect sensitive components. However, it has been demonstrated that attackers can still compromise a vehicle's powertrain by exploiting vulnerabilities in various components, such as the in-vehicle infotainment (IVI) OS, instrument cluster, and Electronic Control Units (ECUs) [19, 26, 27, 40].

Due to the closed-source nature of in-vehicle COTS firmware, recent research has primarily focused on fuzzing based on firmware emulation [15, 24, 34, 48, 49]. One of the key challenges in this approach is the fuzzing oracle, i.e. determining whether a vulnerability has been successfully triggered. Even in the absence of source code, we can still detect heap-based vulnerabilities by tracking allocator functions. To achieve this, researchers have explored various methods to integrate sanitizers into binary programs [14, 16, 17, 20, 38, 41, 42]. Some of them rely on Dynamic Binary Instrumentation (DBI) techniques [20, 38], while others utilize binary rewriting techniques [17, 41, 42]. Additionally, certain approaches leverage hardware-assisted mechanisms [14] or page fault [16] to implement sanitizers efficiently.

However, the majority of these works are concentrated on the analysis of user-space programs. KRetroWrite [41] and BoKASAN [16] are the most related works. The former only focuses on rewriting the kernel module, which cannot be applied to the whole kernel image. Meanwhile, the latter requires compiling the kernel module for instrumentation, which is not feasible for COTS kernels. Moreover, it is impossible to embed multiple sanitizers simultaneously to enrich the fuzzing oracles and improve the fuzzing capability [45].

To better fuzz in-vehicle COTS OS kernels (e.g. Linux, QNX), we propose QKSAN, a multiple sanitizers framework designed for COTS kernels. QKSAN maximizes the use of the TCG (Tiny Code Generator) mode mechanism to achieve efficient memory status monitoring. Specifically, QKSAN leverages the context switch function of the kernel to achieve selective sanitization. It also innovatively uses a compound shadow memory to store metadata and employs an efficient hypervisor-level function instrumentation method to detect memory violation bugs (e.g. out-of-bound write) within the kernel. To facilitate its integration into existing emulation and fuzzing solutions, QKSAN builds on top of the QEMU translation framework. Benefiting from a function hook mechanism designed for QEMU TCG, QKSAN is equipped with KASAN functionality and can simultaneously activate a basic KMSAN, enabling memory violation detection in the kernel even when only the binary kernel file is available. Moreover, QKSAN can conduct selective sanitizing by hooking the context switch function to minimize performance overhead, which can greatly benefit the fuzzing process. Finally, we implemented a fuzzing framework that integrates QKSAN and code coverage based on QEMU to perform binary-only fuzzing on COTS kernels. To the best of our knowledge, QKSAN is the first work to implement a memory sanitizer within the QNX kernel.

In summary, the main contributions of this paper are as follows:

- We introduced a multiple sanitizers framework, which enables QKSAN to combine multiple sanitizers relying on function-level instrumentation through the hypervisor.
- We implemented QKSAN and conducted evaluations to prove QKSAN's ability to detect vulnerabilities in COTS kernels.
- We demonstrate a fuzzing framework that integrates QKSAN and code coverage based on QEMU to perform binary-only fuzzing on COTS kernels, and prove that QKSAN can be used in the fuzzing process of COTS kernels.

2 Background

In this section, we describe why we need binary-level sanitizers and explain why we chose a hypervisor-based approach to implement QKSAN.

2.1 Binary Kernel Fuzzing

In-vehicle systems, such as IVI and instrument cluster, are typically equipped with Linux or QNX kernels. For security reasons, manufacturers usually do not provide the corresponding source code, leaving us with access only to the binary files. These kernels are often customized by manufacturers, leading to the kernel not being as well-tested as an open-source kernel.

As one of the state-of-the-art kernel fuzzers, Syzkaller [22] and syzkaller-based works ([8, 13, 30, 32, 35]) leverage the Linux kernel's KCOV [3] to collect runtime code coverage and utilize various sanitizers including KASAN [4], KMSAN [6], and UBSAN [10] to catch memory violation bugs. However, these mechanisms of the Linux kernel rely on compile-time instrumentation, which is impossible to apply to a binary-only kernel. Current binary kernel fuzzers [37, 39] are not equipped with any sanitizer. BoKASAN [16] requires the user to compile a kernel module for the target kernel, which is not feasible in most cases of in-vehicle Linux kernels. Moreover, to the best of our knowledge, there are no existing sanitizer implementations for the QNX kernel. Therefore, it is necessary to fill the gap in the area of binary kernel sanitizers.

2.2 Memory Sanitization

Song et al. [45] and Emanuel et al. [47] have summarized the common types of memory errors in low-level languages (C/C++) and the existing detection methods. The primary issues we focus on are spatial and temporal memory bugs associated with heap variables, including use-after-free (UAF), double-free, out-of-bounds (OOB) reads and writes, and the use of uninitialized variables (uninit-value). These types of vulnerabilities correspond to the KASAN and KMSAN mechanisms in the Linux kernel.

Redzone inserting with shadow memory is the most commonly used technique used to detect various memory safety violations, including spatial and temporal memory safety violations. Genrally speaking, each object in memory has corresponding shadow memory. To catch the out-of-bounds access, sanitizer inserts redzone adjecent to the object in the shadow memory. If any memory accessing touches the redzone, it will be regarded as an illegal behavior. For temporal violations, sanitizer poison the shadow memory of an object as "freed" at the end of the object's lifetime. Shadow memory can also assist the detection of using uninitialized variables. KMSAN [6] poisons the shadow memory of an object until the first writing to it. Any access to the object will be checked and intercepted if the corresponding shadow memory is marked as uninitialized. Based on the rationale of sanitizers, memory access and object construction/destruction instrumentations are neccessary for the implementation of shadow memory based sanitizers.

For a bianry-only target, we cannot monitor the stack-based variables since we lack the information of the size and addresses of them. Fortunately, through the function-level instrumentation, we are able to fetch the size and addresses of heap-based variables and allocate shadow memory for them to implement sanitizers.

However, the implementation of sanitizers in the COTS kernel is not trivial. First, **the closed nature of COTS kernels makes it challenging to obtain information and modify the system. (Challenge 1).** This limits the possibilities of binary rewriting or adapting page fault hooks like BoKASAN (compiling Linux Kernel Moudle requires files that are generated during the kernel build process, e.g. `Module.symves`). Second, **system-level sanitizers may introduce unacceptable runtime overhead (Challenge 2).** Although the scale of in-vehicle kernels is relatively small, we still need to consider the runtime overhead of the sanitizers. Third, **existing sanitizer implementations cannot enable ASAN and MSAN simultaneously (Challenge 3).** For example, the Linux kernel's KASAN and KMSAN both use shadow memory strategies and greatly modify the kernel's allocator but have different implementations. Additionally, enabling both of them may introduce unacceptable runtime overhead. As a result, `CONFIG_KASAN` and `CONFIG_KMSAN` are conflicting options in the Linux kernel.

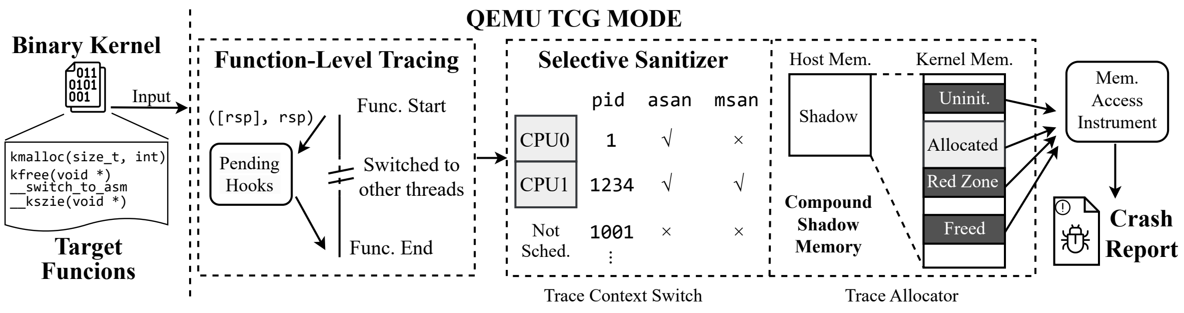

Figure 1: Workflow of QKSAN

3 Design

To address the challenges mentioned above, we proposed QKSAN, a multiple sanitizers framework designed for COTS kernels. Due to the limitation of missing source code information, details about stack variables are lost. Therefore, the objective of QKSAN is to detect various heap-based memory corruption vulnerabilities, including use-after-free (UAF), double-free, out-of-bounds (OOB) reads and writes, and the use of uninitialized variables (uninit-value). As depicted in Figure 1, given a binary-only kernel along with a list of target functions, QKSAN runs the kernel in QEMU TCG mode to facilitate the instrumentation for binary-level sanitizers. We provide a distinctive function hook mechanism (Section 3.1) to support shadow memory manipulation, avoiding making any changes to the COTS system (**Challenge 1**). To increase flexibility and dynamic performance, we design a selective sanitization mechanism (Section 3.2) by tracing the context switch function to monitor the process running on each CPU (**Challenge 2**). To solve the conflict of the current implementation of KASAN and KMSAN, we designed a compound shadow memory strategy (Section 3.3) to enable the function of multiple sanitizers simultaneously (**Challenge 3**). The detailed design of three components will be discussed in the following.

3.1 Function-Level Tracing

Instrumentation. Unlike QASan [20] and BoKASAN [16], QKSAN is restricted from modifying the target system, which results in inserting a function hook not being straightforward. They run their instrumentation functions along with the target programs, which allows them to rewrite or reuse functions easily by compiling their instrumentation functions. However, the kernel runs in the QEMU TCG mode in QKSAN, in which the code is executed in units of Translation Blocks (TB). A TB may contain multiple entry points but must end with a branch instruction or some other special instruction as the only exit point. When QEMU encounters an unknown TB, it lifts the target code to intermediate representation (IR) and translates it to host instructions. When translating a TB that starts with an interesting function address, we can insert a function hook at the beginning of the TB.

The instrumentation strategy is described in Algorithm 1. To align with the requirement of memory manager state tracing, QKSAN inserts an IR statement at the beginning of interesting functions at the lifting stage. When control flow reaches these functions, QKSAN records the return address and stack pointer of the current context. Then QKSAN invokes $prevFuncHandler$ to perform specific preprocessing before function execution, such as resizing the request size of the memory allocation. Another indispensable task of QKSAN is tracking the end of the function to get the function return value and manipulate the shadow memory. However, it is hard to track all the return instructions in the function. We choose to monitor every beginning of TB since the first instruction after a function returns is always the start of TB. If the context of the TB start matches that recorded in the $pendingHooks$, then the TB is regarded as the function's end. Then QKSAN invokes $postFuncHandler$ and deletes the corresponding entry from $pendingHooks$.

Target functions. Similar to other sanitizers, QKSAN tracks the allocator state of the target kernel by tracing functions related to memory allocation. To achieve this, QKSAN mainly focuses on the following functions (refer to Appendix A for details):

Cache creation. In Linux's slab/slub allocator, `kmem_cache` should be created before chunk allocation. For the purpose of increasing the request size of the allocator, QKSAN intercepts the `kmem_cache_create` function in the Linux kernel and modifies the request size of cache creation. After that, all chunks allocated from this function will get a larger size reserved for the redzone.

Allocation. When QKSAN catches a memory allocation, it will try to increase the requested chunk size through the $prevFuncHandler$ in Algorithm 1. After the allocation is done, QKSAN leverages the $postFuncHandler$ to get the allocated chunk address, then allocates and poisons the shadow memory. Additionally, to meet the requirements of KMSAN, QKSAN also poisons the shadow memory as uninitialized (refer to Section 3.3 for details) unless the chunk is initialized during allocation (e.g., with the `GFP_ZERO` flag or the cache is created with a constructor in Linux).

Deallocation. Similar to allocation, QKSAN checks the freed address and looks it up from the recorded chunks. Then QKSAN marks the shadow memory as freed objects. If the chunk is already freed, QKSAN will report this free operation as a double-free bug.

Symbol value concern. We cannot directly obtain symbol information from COTS kernels. However, it is feasible to retrieve this information through some manual efforts. For COTS Linux kernel, most of them include a compressed symbol table (kallsyms) used for moudle loading, from which we can reconstruct the kernel's symbol table and extract the necessary function addresses. In the case of QNX kernels, as QNX is distributed with debug information, we can analyze the stripped binary by comparing it with the binary that includes debug information (e.g., using BinDiff [23]), and subsequently locate the addresses of the target functions.

Algorithm 1 Function-level hook

```
Require: $HookTargets$
  $pendingHooks \leftarrow \varnothing$
  function $TranslateNewTB(StartAddr)$
      Invoked when TCG cpu encounter a new TB
      if $StartAddr \in HookTargets$ then
          Insert $FuncTracer$ into the IR TB
      end if
  end function
  function $FuncTracer$
      $newHook \leftarrow (retAddr, CurrentRSP + 8)$        ▷ for 64-bit machine
      $pendingHooks \leftarrow pendingHooks \cup newHook$
      Invoke $prevFuncHandler$ before the function
  end function
  function $TBStartTracer$
      Invoked at every beginning of TB
      if $isEmpty(pendingHooks)$ then
        return
      end if
      $ThisHook \leftarrow (CurrentAddr, CurrentRSP)$
      if $ThisHook \in pendingHooks$ then
          $pendingHooks \leftarrow pendingHooks \setminus \{ThisHook\}$
          Invoke $postFuncHandler$ after the function
      end if
  end function
```

3.2 Selective Sanitization by Processes

To achieve the objective of selective sanitization, we trace the context switch function in the kernel. The kernel invokes a proprietary context switch function when the CPU switches its context from one process to another in both Linux and QNX kernel. To monitor the state of processes and identify those of interest, QKSAN retrieves the Process ID (PID) and the name of each process. For the Linux kernel, this information can be read from the `task_struct`, which is the parameter of the `__switch_to_asm` function. Furthermore, we are able to fetch the field offset inside `task_struct` even though we cannot access the source code or debug information of the binary kernel. As shown in Listing 1, we can easily find a function and get the field offset from the assembly code: Line 10 in Listing 1 indicates that the offset of `pid` in the `task_struct` is `0x5c0`.

Listing 1: Example of fetching field offset of `task_struct`

```
1  int trace_save_cmdline(struct task_struct *t)
2  {
3      if (!t->pid)
4          return 1;
5      ...
6  // corresponding assembly code
7  endbr64
8  mov    eax, dword ptr [rdi + 0x5c0]
9  mov    rsi, rdi
```

QKSAN maintains a hash table that contains the PID and name of all processes, along with the current process information of all CPUs, which is updated every time the execution reaches the context switch function. QKSAN also maintains a bitmap of the sanitizers' state, which is checked before validating the shadow memory. The process information is later used to control the switch of the sanitizers and track the interesting processes by name. Therefore, QKSAN does not record the allocator process of memory chunks.

When deploying QKSAN within the fuzzing process, selective sanitization can enhance performance by disregarding non-essential processes. For instance, in the case of fuzzing Linux kernels with syzkaller [22], the underlying syscalls are executed by `syz-executor`. We can configure QKSAN to monitor solely the `syz-executor` process. However, some instances of illegal memory access might be overlooked by the sanitizer if the process is not under surveillance. Nevertheless, the proportion of such omissions remains relatively small, approximately 20% [16].

3.3 Compound Shadow Memory

As mentioned before, users cannot enable both KASAN and KMSAN in the Linux kernel due to implementation incompatibilities and the conflict of shadow memory and instrumentation. In QKSAN, we proposed compound shadow memory to integrate the effects of multiple sanitizers. To minimize memory usage, we utilize 10 bits to represent the memory state of 8 bytes in the shadow memory. The first 8 bits indicate whether each byte is poisoned by MSAN *or* ASAN, while the last 2 bits are used to represent some special states shown in Figure 2. We impose the following precondition on the shadow memory: the start and end of the chunk (object + redzone) must be aligned to 8 bytes, and we only set the right redzone when a chunk is allocated. Therefore, if the last 2 bits are `00`, `10` or `11`, we are able to determine the state of the original memory. While if they are `01`, there must be a start of the redzone within these 8 bytes according to our precondition. We store the size of the last granule of the redzone in

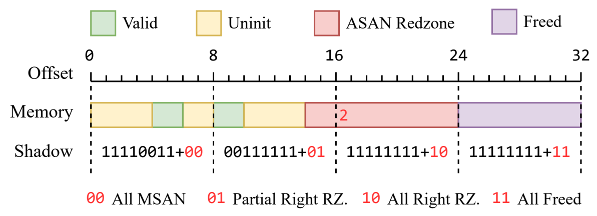

Figure 2: Compound shadow memory

the original memory so that we can read them when checking the memory access.

To monitor memory accesses, QKSAN leverages QEMU TCG to intercept memory operations. Specifically, during the translation of memory access instructions in the guest kernel, QEMU TCG invokes the `tcg_gen_qemu_ld/st_xx` functions. At this stage, QKSAN inserts customized callbacks for each memory access, which subsequently trigger memory violation checks. As a result, all sanitizers implemented in QKSAN are able to reuse this memory access instrumentation, thereby reducing runtime overhead.

When a memory chunk is allocated, QKSAN marks the legal memory area as an uninitialized valid object, then poisons the rest of the area as a redzone. QKSAN fetches the first 8 bits in the shadow memory from `shadow_base + (addr >> 3)` and the last 2 bits from `shadow_base2 + (addr >> 5)`. QKSAN first checks the KASAN state. If the shadow memory is marked as a redzone, QKSAN reports this access as a memory violation according to the shadow byte. When writing to the region that is marked as an uninitialized chunk, QKSAN unpoisons the first 8 bits in the shadow as initialized. Correspondingly, reading an uninitialized region will be reported as a memory violation. When a chunk is freed, all the shadow bytes are poisoned as a freed chunk until a new chunk is allocated at these addresses.

4 Evaluation

In this section, we set up experiments to evaluate the performance of QKSAN. We aim to answer the following research questions:

- **RQ1** Can QKSAN detect memory violation bugs in Linux and QNX kernel?
- **RQ2** What is the performance overhead of QKSAN compared to KASAN?
- **RQ3** Can QKSAN be used in binary-only COTS kernel fuzzing?

4.1 Implementation

We implemented QKSAN based on QEMU v8.1.3 and extended it with about 5,000 lines of C code. The source code of QKSAN and the evaluation dataset are all published on `https://github.com/a4rech/QKSAN-demo`. Users are able to enable or disable multiple sanitizers and customize the redzone size and the name of the interesting program. Additionally, QKSAN implements a hypervisor-level code coverage collection function for testing in fuzzing scenarios.

When using QKSAN, users must provide a configuration file that includes a list of instrumentation function addresses and key struct field offsets. These instrumentation functions may differ across various kernel versions and build configurations, which needs manual analysis of the target kernel. For Linux kernels, we initially employ the vmlinux-to-elf [36] tool to retrieve symbol information from the bzImage file, verify the symbol states of the slab/slub allocator, and document their values. For field offsets, we select specific functions and examine their assembly code as outlined in Section 3.2. For QNX kernels, we compare the target binary (using BinDiff [23]) with a binary containing debug information to identify the function addresses of specific functions. The manual effort required is typically under 20 minutes. Detailed information about allocators is available in Appendix A.

4.2 Evaluation Setup

We set up the evaluation mainly to test the functionality of QKSAN and the performance overhead. The detailed configuration is as follows:

Linux Datasets. Since QKSAN is designed for detecting heap-based vulnerabilities, to evaluate the functionality of QKSAN, we randomly selected 52 heap-based memory corruption vulnerabilities reported by KASAN which were retrieved from syzbot [7]. These bugs cover the types slab-out-of-bounds, slab-use-after-free, and double-free. For the KMSAN dataset, to demonstrate the availability of multiple sanitizers, we wrote a simple kernel module that reads from a memory chunk allocated by slub before writing any data into it. Note that we booted the kernel without KASLR for the convenience of instrumentation.

QNX Datasets. It is hard to get a rich vulnerabilities dataset of the QNX kernel, so we chose to write PoC programs to evaluate the design of QKSAN. However, the QNX kernel does not provide a kernel module mechanism like Linux. Fortunately, the `InterruptHookIdle2` kernel call [2] allows us to register interrupt handlers to execute code in ring0 within the same address space as the kernel. Therefore, it is possible to invoke the kernel's allocator functions through function pointers, by which we can simulate triggering memory violation bugs in the kernel. We implemented PoCs that cover bug types heap-out-of-bound, heap-use-after-free, double-free, and use uninit-value on QNX SDP 7.0.

Table 1: Bug detecting result of KASAN and QKSAN

Kernel	Type	# of PoC	KASAN	QKSAN
Linux	out-of-bound	26	26	25
	use-after-free	25	25	25
	double-free	1	1	1
	uninit-value	1	1	1
QNX	out-of-bound	1	-	✓
	use-after-free	1	-	✓
	double-free	1	-	✓
	uninit-value	1	-	✓

Platform and Configuration. The evaluations were conducted on an Ubuntu 24.04 LTS server with an AMD Ryzen 9 5900X CPU and 64GB RAM. For the fuzzing test, we extended syzkaller to leverage the code coverage reported by QKSAN, and then we ran syzkaller for 24 hours on both Linux and QNX kernels. Each syzkaller instance comprises 4 virtual machines, each equipped with 1 core and 2GB of memory.

4.3 Detecting Memory Violations

To answer **RQ1**, we utilized the previously described datasets to compare the detection capabilities of QKSAN and KASAN. We executed all the PoC programs on QKSAN with all sanitizers enabled. After each kernel crash, we compared the crash reports generated by QKSAN and KASAN. The evaluation results are summarized in Table 1, with detailed results for the Linux dataset provided in Appendix B. QKSAN successfully detected 25 out of 26 out-of-bounds bugs, 25 out of 25 use-after-free bugs, 1 double-free bug, and 1 uninit-value bug in the Linux dataset. The results for the QNX dataset demonstrate that QKSAN successfully detected all the bugs.

The sole out-of-bounds bug that QKSAN was unable to detect occurred because the offset of the OOB access surpassed the redzone length in both KASAN and QKSAN. In KASAN, this memory access touched the redzone of a different object rather than the intended OOB object, leading KASAN to provide incorrect information regarding the OOB object. QKSAN failed to report this crash due to the differing layouts of the objects and redzones.

These results indicate that QKSAN is capable of detecting various memory violation bugs on the heap, including out-of-bounds, use-after-free, and double-free bugs in both Linux and QNX kernels. Additionally, QKSAN can detect uninit-value bugs when KASAN is enabled. Moreover, QKSAN operates effectively *without* requiring any additional source code information beyond the kernel binary, proving its suitability for detecting memory violation bugs in black-box COTS kernels such as vehicle instrument clusters or commercial routers.

4.4 Performace Overhead

To answer **RQ2**, we evaluated the performance overhead of QKSAN compared to KASAN using sysbench [1]. We enabled the sanitizer for the entire system and compared the performance with KASAN. The performance evaluation was conducted on KASAN in KVM mode (KA. KVM in Table 2), KASAN in TCG mode (KA. TCG), TCG mode without KASAN, and QKSAN. In addition to the default test script provided by sysbench, we also measured the kernel boot time (considering the shell prompt as the boot completion).

Table 2: Performance result of KASAN and QKSAN

Item	KA. KVM	TCG	KA. TCG	QKSAN
Boot Time (s)	11.73	25.78	94.93	65.87
Memory (events/min)	103,770.10	6,685.64	5,262.35	2,675.07
Seq. File RW (events/s)	22,537.99	6,230.52	2,288.32	1904.24
Ran. File RW (events/s)	7,673.28	4,056.79	1,743.67	1,777.77
CPU (events/s)	4,045.03	274.04	264.90	110.71
Threads (events/s)	2,192.00	213.76	140.60	65.99

Table 3: Fuzzing result of QKSAN

Kernel	Bug Type	Related Func.	# of Crashes
Linux	heap-user-after-free	`__nf_conntrack_find_get`	13
	heap-user-after-free	`htc_connect_service`	1
	heap-user-after-free	`tipc_sk_backlog_rcv`	1
	heap-user-after-free	`nf_conntrack_tuple_taken`	1
	heap-user-after-free	`batadv_iv_send_outstanding_bat_ogm_packet`	1
	heap-buffer-overflow	`sctp_setsockopt`	3
	heap-buffer-overflow	`getname_kernel`	3
QNX	heap-use-after-free	`idle_exit`	14
	heap-use-after-free	`hook_resp`	1
	heap-use-after-free	`timer_next`	1
	heap-buffer-overflow	`update_system`	1
	heap-use-after-free	`event_add`	1

The results are listed in Table 2. As shown in the table, QKSAN outperforms KASAN in TCG mode in terms of kernel booting and random file read/write operations. However, QKSAN performs worse than KASAN in large memory operations, which may be due to the lack of optimization for specific functions in QKSAN.

4.5 Use QKSAN in Binary-Only Fuzzing

To answer **RQ3**, we evaluated the feasibility of using QKSAN in binary-only kernel fuzzing. We enabled all the sanitizers of QKSAN and the code coverage feedback, and ran syzkaller for 24 hours on both Linux and QNX kernels. The crashes found by QKSAN are listed in Table 3.

QKSAN found 23 crashes in the Linux kernel and 18 crashes in the QNX kernel. We confirmed that the crashes in the Linux kernel are known bugs reported by syzbot before. The reason why QKSAN found fewer bugs might be due to its lower execution efficiency. For the QNX kernel, unfortunately, the crashes found by QKSAN so far are caused by normal function calls of the kernel, which are not considered bugs. However, the crashes found by QKSAN are all related to memory violations. This result indicates that QKSAN can be used in binary-only kernel fuzzing to find memory violation bugs.

5 Discussion & Future Work

Limitations of QKSAN. QKSAN needs symbol values and struct field offsets for specific objects, which still requires manual effort. In some COTS Linux kernels, `kallsyms` may be stripped, complicating the configuration of QKSAN. Additionally, due to the paucity of information, QKSAN is limited to addressing heap-based vulnerabilities only.

Portability Between Different Kernel Versions. The details of function-level instrumentation rely greatly on the kernel's version and build configuration (e.g., affects the inline functions). QKSAN has already categorized the functions that require instrumentation, and when using QKSAN, one only needs to provide the corresponding configuration file for the specific kernel program. Furthermore, obtaining configuration files for the Linux kernel through an automated approach is possible, and we plan to implement this in the future.

Generality and Scalability. In the current commonly-used sanitizers, KASAN and KMSAN are implemented using shadow memory, which we have integrated into QKSAN. Other sanitizers, such as KCSAN and UBSAN, primarily rely on compiler instrumentation of memory access. These can also be implemented by leveraging TCG's translation stage. The primary challenge is the absence of type information in COTS kernels, complicating the application of certain sanitization strategies.

Enhance Sanitizers with Debug Information. Some COTS kernels (e.g., QNX and Windows), while not disclosing their source code, may provide debugging information along with the distributed binary programs. Debugging information may contain symbols, type information, and stack frame details, which can help us determine the structure of stack and global variables [18]. Therefore, it is possible to implement binary-level sanitizers that are able to work on stack and global variables. We can leverage TCG's translation stage to dynamically modify the function stack layout.

6 Related Work

Currently, most sanitizers rely on compile-time instrumentation to achieve better performance [4–6, 9, 10, 31, 33, 43, 46], which is suitable for developers to debug their programs. However, these methods are only valid for source-available programs. To migrate sanitizers into binary, researchers leverage dynamic binary instrumentation (DBI) to check memory violation bugs for user-space programs [11, 20, 25, 38, 44]. Among them, QASAN [20] has been integrated into AFL++ [21] and has become a common method for testing user-space binary programs. EmbSAN [31] proposes an integrated system to detect memory corruption vulnerabilities in embedded firmware with or without source code, and integrates multiple types of sanitizers. However, it does not provide effective sanitization for binary-only kernels. In addition to instrumentation through DBI, there are also some instrumentation methods on the binary level. *a) Binary rewrite.* Some works (RetroWrite [17], KRetroWrite [41], binTSAN [42]) leverage the binary rewriting technique to insert instrumentation statically, which can reach the performance of compiler-based instrumentation. *b) Page fault hook.* BoKASAN [16] hooks the page fault function by setting the page as unpresented to intercept memory access. *c) Hardware-assisted.* PACMem [29] and MTSan [14] leverage ARM's hardware features to implement sanitizers for memory violations detection with a low runtime overhead and memory footprint. Due to the lack of information, almost no work could detect memory violation bugs on stack-based variables. MTSan [14] uses a probability-based method to infer the boundary of stack variables. However, the result is not always correct.

7 Conclusion

This paper presents QKSAN, the first work to integrate multiple sanitizers for application in binary kernels. By leveraging hypervisor-level instrumentation in QEMU TCG mode, QKSAN successfully combines the functionalities of KASAN and KMSAN in both Linux and QNX kernels. Additionally, QKSAN marks the first implementation of sanitizers within the QNX kernel. Experimental results demonstrate that QKSAN can effectively detect vulnerabilities in binary kernels and feasibly be applied to real-world systems fuzzing.

8 Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62102154), State Key Lab of Processors, Institute of Computing Technology, CAS under Grant No. CLQ202301.

References

[1] GitHub - akopytov/sysbench: Scriptable database and system performance benchmark — github.com. `https://github.com/akopytov/sysbench`.

[2] Interrupthookidle2 – qnx.com. `https://www.qnx.com/developers/docs/7.0.0/#com.qnx.doc.neutrino.lib_ref/topic/i/interrupthookidle2.html`.

[3] KCOV: code coverage for fuzzing – The Linux Kernel documentation — docs.kernel.org. `https://docs.kernel.org/dev-tools/kcov.html`.

[4] Kernel Address Sanitizer (KASAN) – The Linux Kernel documentation — docs.kernel.org. `https://docs.kernel.org/dev-tools/kasan.html`.

[5] Kernel Concurrency Sanitizer (KCSAN) – The Linux Kernel documentation — docs.kernel.org. `https://docs.kernel.org/dev-tools/kcsan.html`.

[6] Kernel Memory Sanitizer (KMSAN) – The Linux Kernel documentation — docs.kernel.org. `https://docs.kernel.org/dev-tools/kmsan.html`.

[7] syzbot — syzkaller.appspot.com. `https://syzkaller.appspot.com/upstream`.

[8] syzkaller/docs/research.md at master · google/syzkaller — github.com. https://github.com/google/syzkaller/blob/master/docs/research.md. [Accessed 03-02-2025].

[9] ThreadSanitizerCppManual — github.com. https://github.com/google/sanitizers/wiki/threadsanitizercppmanual.

[10] Undefined Behavior Sanitizer - UBSAN – The Linux Kernel documentation — docs.kernel.org. https://docs.kernel.org/dev-tools/ubsan.html. [Accessed 03-02-2025].

[11] Bruening, D., and Zhao, Q. Practical memory checking with dr. memory. In *International Symposium on Code Generation and Optimization (CGO 2011)* (2011), pp. 213–223.

[12] Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., and Kohno, T. Comprehensive experimental analyses of automotive attack surfaces. In *20th USENIX Security Symposium (USENIX Security 11)* (San Francisco, CA, Aug. 2011), USENIX Association.

[13] Chen, W., Wang, Y., Zhang, Z., and Qian, Z. SyzGen: Automated generation of syscall specification of Closed-Source macos drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2021), CCS '21, Association for Computing Machinery, p. 749–763.

[14] Chen, X., Shi, Y., Jiang, Z., Li, Y., Wang, R., Duan, H., Wang, H., and Zhang, C. MTSan: A feasible and practical memory sanitizer for fuzzing COTS binaries. In *32nd USENIX Security Symposium (USENIX Security 23)* (Anaheim, CA, Aug. 2023), USENIX Association, pp. 841–858.

[15] Chen, Z., Thomas, S. L., and Garcia, F. D. MetaEmu: An architecture agnostic rehosting framework for automotive firmware. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2022), CCS '22, Association for Computing Machinery, p. 515–529.

[16] Cho, M., An, D., Jin, H., and Kwon, T. BoKASAN: Binary-only kernel address sanitizer for effective kernel fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)* (Anaheim, CA, Aug. 2023), USENIX Association, pp. 4985–5002.

[17] Dinesh, S., Burow, N., Xu, D., and Payer, M. RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), pp. 1497–1511.

[18] DWARF Debugging Information Format Committee. DWARF debugging information format version 5. https://dwarfstd.org/doc/DWARF5.pdf, 2017.

[19] Evdokimov, M. 0-click RCE on the IVI component: Pwn2Own automotive edition. *Hexacon* (2024).

[20] Fioraldi, A., D'Elia, D. C., and Querzoni, L. Fuzzing binaries for memory safety errors with QASan. In *2020 IEEE Secure Development (SecDev)* (2020), pp. 23–30.

[21] Fioraldi, A., Maier, D., Eissfeldt, H., and Heuse, M. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)* (Aug. 2020), USENIX Association.

[22] Google. GitHub - google/syzkaller: syzkaller is an unsupervised coverage-guided kernel fuzzer — github.com. https://github.com/google/syzkaller.

[23] google. google/bindiff: Quickly find differences and similarities in disassembled code. [Online; accessed 2025-04-24].

[24] Harrison, L., Vijayakumar, H., Padhye, R., Sen, K., and Grace, M. PARTEMU: Enabling dynamic analysis of Real-World TrustZone software using emulation. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 789–806.

[25] Hastings, R. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter'92 Conference* (1992), pp. 125–136.

[26] Janushkevich, D. Zero Day Initiative — Multiple Vulnerabilities in the Mazda In-Vehicle Infotainment (IVI) System — zerodayinitiative.com. https://www.zerodayinitiative.com/blog/2024/11/7/multiple-vulnerabilities-in-the-mazda-in-vehicle-infotainment-ivi-system, 2024.

[27] Jing, P., Cai, Z., Cao, Y., Yu, L., Du, Y., Zhang, W., Qian, C., Luo, X., Nie, S., and Wu, S. Revisiting automotive attack surfaces: a practitioners' perspective. In *2024 IEEE Symposium on Security and Privacy (SP)* (2024), pp. 2348–2365.

[28] Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., and Savage, S. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy* (2010), pp. 447–462.

[29] Li, Y., Tan, W., Lv, Z., Yang, S., Payer, M., Liu, Y., and Zhang, C. PACMem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022), pp. 1901–1915.

[30] Lin, Z., Chen, Y., Wu, Y., Mu, D., Yu, C., Xing, X., and Li, K. Grebe: Unveiling exploitation potential for linux kernel bugs. In *2022 IEEE Symposium on Security and Privacy (SP)* (2022), pp. 2078–2095.

[31] Liu, J., Shen, Y., Xu, Y., Sun, H., Shi, H., and Jiang, Y. Effectively Sanitizing Embedded Operating Systems. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco CA USA, June 2024), ACM, pp. 1–6.

[32] Liu, Z., Lin, Z., Chen, Y., Wu, Y., Zou, Y., Mu, D., and Xing, X. Towards unveiling exploitation potential with multiple error behaviors for kernel bugs. *IEEE Transactions on Dependable and Secure Computing 21*, 1 (2024), 93–109.

[33] LLVM Developers. Control Flow Integrity – Clang 21.0.0git documentation — clang.llvm.org. https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[34] Maier, D., Seidel, L., and Park, S. BaseSAFE: baseband sanitized fuzzing through emulation. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (New York, NY, USA, 2020), WiSec '20, Association for Computing Machinery, p. 122–132.

[35] Maier, D., and Toepfer, F. BSOD: Binary-only scalable fuzzing of device drivers. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses* (New York, NY, USA, 2021), RAID '21, Association for Computing Machinery, p. 48–61.

[36] marin m. GitHub - marin-m/vmlinux-to-elf: A tool to recover a fully analyzable .ELF from a raw kernel, through extracting the kernel symbol table (kallsyms) — github.com. https://github.com/marin-m/vmlinux-to-elf.

[37] nccgroup. GitHub - nccgroup/TriforceAFL: AFL/QEMU fuzzing with full-system emulation. — github.com. https://github.com/nccgroup/TriforceAFL.

[38] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, Association for Computing Machinery, p. 89–100.

[39] Pandey, P., Sarkar, A., and Banerjee, A. Triforce qnx syscall fuzzer. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (2019), pp. 59–60.

[40] Parnishchev, D., Ivachev, A., Evdokimov, M., Stennikov, A., Smirnova, P., and Motspan, R. Over the air: Compromise of modern volkswagen group vehicles. *Black Hat Europe* (2024).

[41] Rizzo, M. Hardening and testing privileged code through binary rewriting. Master's thesis, EPFL, 2020.

[42] Schilling, J., Wendler, A., Görz, P., Bars, N., Schloegel, M., and Holz, T. A binary-level thread sanitizer or why sanitizing on the binary level is hard. In *33rd USENIX Security Symposium (USENIX Security 24)* (Philadelphia, PA, Aug. 2024), USENIX Association, pp. 1903–1920.

[43] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, June 2012), USENIX Association, pp. 309–318.

[44] Seward, J., and Nethercote, N. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track* (2005), pp. 17–30.

[45] Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P., and Franz, M. SoK: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), pp. 1275–1295.

[46] Stepanov, E., and Serebryany, K. Memorysanitizer: Fast detector of uninitialized memory use in c++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2015), pp. 46–55.

[47] Vintila, E., Zieris, P., and Horsch, J. Evaluating the Effectiveness of Memory Safety Sanitizers. In *2025 IEEE Symposium on Security and Privacy (SP)* (Los Alamitos, CA, USA, May 2025), IEEE Computer Society, pp. 88–88. ISSN: 2375-1207.

[48] Zhou, W., Guan, L., Liu, P., and Zhang, Y. Automatic firmware emulation through invalidity-guided knowledge inference. In *30th USENIX Security Symposium (USENIX Security 21)* (2021).

[49] Zhou, W., Zhang, L., Guan, L., Liu, P., and Zhang, Y. What your firmware tells you is not how you should emulate it: A specification-guided approach for firmware emulation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022), CCS '22.

A Function Hook Details

In the Linux kernel, the core allocator functions are implemented as inline functions. A function-level tracing has to instrument a set of functions in the binary to completely understand the memory manager's state. The demo instrument functions of Linux v6.9.5 built with configuration from syzbot (refer to [7]) are listed in Table 4. QKSAN is responsible for handling various types of memory allocation and deallocation functions.

Cache creation. In the Linux kernel, a user must create a `kmem_cache` structure before allocating memory chunks using functions such as `kmem_cache_alloc`. Consequently, QKSAN should trace the cache creation and destruction functions to obtain the cache information and ensure it remains current.

`__kmem_cache_alias` is a special function that is used to create an alias for a cache. When the kernel requests a new cache, it will search for a similar cahce created before. If the cache is found, the kernel will create an alias for the cache. Somtimes the new chace and the old size may carry different chunk size, so QKSAN should also trace this function to update the chunk size and redzone size.

Allocate with cache. After a cache created, when QKSAN encounters a memory allocation operation through `kmem_cache_alloc`, QKSAN will first fetch the address of cache in the parameters and search the cache information to get the size of the chunk and the corresponding shadow memory. Note that `*bulk` functions operate on a set of chunks at one time, so QKSAN should also read the chunk number from the function's parameters and process the chunks in batches.

Table 4: Instrument function in Linux kernel

Allocate with Cache	`kmem_cache_alloc`, `kmem_cache_alloc_lru`, `kmem_cache_alloc_node`, `kmalloc_trace`, `kmalloc_node_trace`, `kmem_cache_alloc_bulk`
Allocate with Size	`__kmalloc`, `__kmalloc_node`
Deallocate	`kfree`, `kmem_cache_free`, `kmem_cache_free_bulk`
Cache Creation	`kmem_cache_create`, `kmem_cache_destroy`, `kmem_cache_create_usercopy`, `__kmem_cache_alias`
Context Switch	`__switch_to_asm`
Special	`post_alloc_hook`, `__ksize`, `memcpy`, `memmove`, `memset`

In Linux kernel, if `kmalloc` is called with a size that can be determined at compile time, it will generate `kmalloc_trace` with a predetermined `kmalloc_caches` actually. To enlarge the chunk size allocated by `kmalloc_trace`, QKSAN should also select a suitbale cache from `kmalloc_caches` for the chunk. Therefore, QKSAN should know the address of `kmalloc_caches` when sanitizing Linux kernels.

Allocate with size. QKSAN trace the `__kmalloc` function to handle the memory allocation with a size that cannot be determined at compile time. We simple reset the parameters of the function to enlarge the request size, this can easily be done by modify the register values at the very beginning of the function.

Deallocate. When encounters deallocation operations, QKSAN searches for the allocated chunk information to get the original chunk size, then unpoison the shadow memory accordingly.

Special functions. `post_alloc_hook` unpoisons pages after they are allocated by the low-level allocator; QKSAN does the same thing to keep the system stable. `__ksize` is invoked inside `kfree_sensitive`, in which KASAN needs extra unpoison operations. We also trace functions like `memcpy` to employ batch processing of shadow memory to reduce the runtime overhead.

QNX kernel. In the QNX kernel, `_srealloc` is the only core function related to memory allocation. This function takes three key parameters: pointer, old size, and new size. If the pointer and old size are not NULL, it performs a deallocation operation; if the new size is not NULL, it performs an allocation operation. The QNX kernel calls `x86_64_switch` to perform context switching.

QNX is distributed in binary form containing the kernel and a set of userspace tools. The kernel `procnto-smp-instr` is a relocatable ELF binary with debug information. We perform reverse analysis on it and fetch the address of `_srealloc` function and field offsets like Linux. There is no address space randomization during the QNX's boot stage, so we can fetch the fixed base address of the kernel by dynamic debugging. After that we can infer the right address for instrumentation.

B Linux Memory Violation Bugs Dataset

Table 5: Detailed Result of Linux Vulnerabilities Detection

KASAN BUG Type	KASAN Crash Func.	QKSAN BUG Type	QKSAN Crash Func.	QKSAN Crash Addr.	Check Pass
slab-use-after-free	`__x64_sys_io_cancel`	heap-use-after-free	`__do_sys_io_cancel`	0xffffffff8159e51c	✓
double-free	`relay_open`	heap-use-after-free	`relay_open`	0xffffffff814a3c70	✓
slab-use-after-free	`rb_first_postorder`	heap-use-after-free	`rb_first_postorder`	0xffffffff83e113b0	✓
slab-out-of-bounds	`__sctp_v6_cmp_addr`	heap-buffer-overflow	`__sctp_v6_cmp_addr`	0xffffffff83b2b91d	✓
slab-use-after-free	`snd_pcm_oss_get_formats`	heap-use-after-free	`snd_pcm_oss_get_formats`	0xffffffff834485c1	✓
slab-out-of-bounds	`pfkey_add`	heap-use-after-free	`pfkey_add`	0xffffffff83e2cc3b	✓
slab-use-after-free	`hfsplus_release_folio`	heap-use-after-free	`hfsplus_release_folio`	0xffffffff81718b0a	✓
slab-use-after-free	`__lock_acquire`	heap-use-after-free	`arch_atomic_try_cmpxchg`	0xffffffff83e41ca6	✓
slab-use-after-free	`rxrpc_lookup_local`	heap-use-after-free	`rxrpc_lookup_local`	0xffffffff83aa4258	✓
slab-out-of-bounds	`vxlan_vnifilter_dump_dev`	heap-buffer-overflow	`vxlan_vnifilter_dump_dev`	0xffffffff8276ba70	✓
slab-use-after-free	`port100_send_complete`	heap-use-after-free	`port100_send_async_complete`	0xffffffff82590d90	✓
slab-out-of-bounds	`bpf_prog_test_run_xdp`	heap-buffer-overflow	`bpf_prog_test_run_xdp`	0xffffffff8365a02d	✓
slab-out-of-bounds	`ipvlan_queue_xmit`	heap-use-after-free	`ipvlan_queue_xmit`	0xffffffff8367b8fe	✓
slab-use-after-free	`si470x_int_in_callback`	heap-use-after-free	`si470x_int_in_callback`	0xffffffff83003126	✓
slab-out-of-bounds	`watch_queue_set_filter`	heap-buffer-overflow	`watch_queue_set_filter`	0xffffffff814105e0	✓
slab-out-of-bounds	`do_garbage_collect`	heap-buffer-overflow	`do_garbage_collect`	0xffffffff81d8a84c	✓
slab-use-after-free	`nilfs_segctor_confirm`	heap-use-after-free	`nilfs_segctor_confirm`	0xffffffff81b30590	✓
slab-use-after-free	`__post_watch_notification`	heap-use-after-free	`__post_watch_notification`	0xffffffff8140fba0	✓
use-after-free	`__isofs_iget`	heap-use-after-free	`isofs_read_inode`	0xffffffff817129e6	✓
slab-use-after-free	`io_submit_one`	heap-use-after-free	`__io_submit_one`	0xffffffff8159d2dc	✓
slab-use-after-free	`nf_confirm`	heap-use-after-free	`nf_confirm`	0xffffffff8367a656	✓
slab-out-of-bounds	`add_del_if`	heap-buffer-overflow	`add_del_if`	0xffffffff83903810	✓
slab-use-after-free	`tipc_recvmsg`	heap-use-after-free	`tipc_recvmsg`	0xffffffff83c906c2	✓
slab-out-of-bounds	`snd_usbmidi_get_ms_info.isra.0`	heap-buffer-overflow	`snd_usbmidi_get_ms_info`	0xffffffff834d5e91	✓
slab-use-after-free	`ieee80211_ibss_build_presp`	heap-use-after-free	`ieee80211_ibss_build_presp`	0xffffffff83e2cc06	✓
slab-out-of-bounds	`crypto_arc4_crypt`	heap-buffer-overflow	`crypto_arc4_crypt`	0xffffffff83e2cc06	✓
slab-out-of-bounds	`strset_parse_request`	heap-buffer-overflow	`strset_parse_request`	0xffffffff83664df1	✓
slab-out-of-bounds	`nla_find`	heap-use-after-free	`nla_find`	0xffffffff8367b8fe	✓
out-of-bounds	`leaf_paste_entries`	heap-buffer-overflow	`leaf_paste_entries`	0xffffffff81fddf9e	✓
slab-out-of-bounds	`f2fs_build_segment_manager`	heap-buffer-overflow	`f2fs_build_segment_manager`	0xffffffff81db2d6d	✓
slab-out-of-bounds	`prism2sta_probe_usb`	heap-buffer-overflow	`prism2sta_probe_usb`	0xffffffff8336d87e	✓
slab-out-of-bounds	`sctp_setsockopt`	heap-buffer-overflow	`sctp_setsockopt_delayed_ack`	0xffffffff83b171d9	✓
use-after-free	`vcs_read`	heap-use-after-free	`vcs_read`	0xffffffff8367b8fe	✓
slab-out-of-bounds	`snd_usb_mixer_notify_id`	heap-buffer-overflow	`snd_usb_mixer_notify_id`	0xffffffff834bb983	✓
slab-use-after-free	`kfree_skb_reason`	heap-use-after-free	`kfree_skb_list_reason`	0xffffffff83547965	✓
slab-out-of-bounds	`bitmap_ip_ext_cleanup`	heap-use-after-free	`bitmap_ip_ext_cleanup`	0xffffffff83701cc8	✓
slab-use-after-free	`xp_put_pool`	heap-use-after-free	`xp_put_pool`	0xffffffff83dcb55d	✓
slab-use-after-free	`bitmap_port_ext_cleanup`	heap-use-after-free	`bitmap_port_ext_cleanup`	0xffffffff83704668	✓
slab-out-of-bounds	`ntfs_listxattr`	heap-buffer-overflow	`ntfs_listxattr`	0xffffffff83e2cc06	✓
slab-out-of-bounds	`setup_udp_tunnel_sock`	heap-buffer-overflow	`setup_udp_tunnel_sock`	0xffffffff83801b23	✓
slab-use-after-free	`snd_timer_resolution`	heap-use-after-free	`snd_timer_resolution`	0xffffffff834275a9	✓
slab-out-of-bounds	`inet_diag_bc_sk`	-	-	-	✗
slab-out-of-bounds	`mpol_parse_str`	heap-buffer-overflow	`mpol_parse_str`	0xffffffff814ced06	✓
slab-out-of-bounds	`hsr_debugfs_rename`	heap-buffer-overflow	`hsr_debugfs_rename`	0xffffffff83db77d0	✓
slab-out-of-bounds	`bpf_prog_create`	heap-buffer-overflow	`bpf_prog_create`	0xffffffff83e2cc3b	✓
slab-use-after-free	`kfree_skb_reason`	heap-use-after-free	`kfree_skb_reason`	0xffffffff8364ec97	✓
slab-use-after-free	`xsk_diag_dump`	heap-use-after-free	`xsk_diag_put_info`	0xffffffff83dcb950	✓
slab-use-after-free	`do_raw_spin_lock`	heap-use-after-free	`arch_atomic_try_cmpxchg`	0xffffffff83e424f2	✓
slab-out-of-bounds	`tun_net_xmit`	heap-buffer-overflow	`tun_net_xmit`	0xffffffff82747e60	✓
slab-use-after-free	`tipc_group_bc_cong`	heap-use-after-free	`tipc_sk_filter_rcv?`	0xffffffff83c93839	✓
slab-use-after-free	`get_mem_cgroup_from_mm`	heap-use-after-free	`get_mem_cgroup_from_mm`	0xffffffff814fbad5	✓
slab-out-of-bounds	`strlen`	heap-buffer-overflow	`strlen`	0xffffffff83e13aa0	✓