SoK: Take a Deep Step into Linux Kernel Hardening Effectiveness from the Offensive-Defensive Perspective

Yinhao Hu[§†‡], Pengyu Ding[§†‡], Zhenpeng Lin[¶], Dongliang Mu[§‡✉], Yuan Li[†✉]

[§]School of Cyber Science and Engineering, Huazhong University of Science and Technology, China
[†]Zhongguancun Laboratory
[‡]Hubei Key Laboratory of Distributed System Security
[¶]Independent Researcher
{dddddd, pengyu_ding, dzm91}@hust.edu.cn, zplin@u.northwestern.edu, lydorazoe@gmail.com

***Abstract*—Despite extensive efforts to harden the Linux kernel—the foundation powering numerous widely-used distributions (e.g., Ubuntu, Debian, Fedora)—it continues to face persistent and sophisticated memory safety vulnerabilities. In this study, we introduce a novel systematic framework that decomposes kernel exploitation into three distinct phases from an attacker's perspective. Through comprehensive analysis of 121 publicly documented exploits since 2015, we identify and categorize 64 recurrent attack vectors. Leveraging this structured approach, we perform an in-depth evaluation of 51 existing kernel defense mechanisms, clearly mapping their coverage, limitations, redundancies, and interdependencies. Our results reveal significant protection gaps: 23 attack vectors remain entirely unprotected, and 31 existing defenses are bypassable or obsolete. Additionally, we uncover notable discrepancies between theoretical effectiveness and practical deployment across popular downstream distributions, highlighting 4 underutilized hardening measures and misconfigurations in four major distributions. By illuminating these critical gaps and offering actionable insights, our work guides both kernel developers and security practitioners in enhancing defensive strategies and refining future security designs.**

I. Introduction

The Linux kernel, which underpins numerous distributions (e.g., Ubuntu, Debian, Fedora), receives thousands of patches daily. This high patch volume reflects the extensive workload on developers and maintainers, increasing the likelihood of oversights that can lead to newly introduced vulnerabilities [1]. Besides, recent data [2] further shows that Linux powers 100% of the world's top 500 supercomputers and 96% of the top one million servers, making the kernel an especially attractive target for adversaries. Memory errors account for most vulnerabilities in large C/C++ codebases [3, 4], and Linux kernel is no exception. Attackers typically induce an internal error in the kernel that escalates into memory corruption, enabling the construction of exploitation primitives and culminating in unauthorized privilege acquisition (e.g., privilege escalation or information disclosure).

To counter these threats, both Linux kernel community and security firms have devoted significant efforts to deploying various kernel hardening mechanisms, each targeting different stages of exploitation. For instance, Linux Kernel Self-Protection Project (KSPP) [5] incorporates a set of recommended kernel hardenings, aiming to make Linux kernel more resilient against vulnerabilities and exploitations.

Our investigation reveals that over 51 kernel hardening mechanisms have been introduced into Linux kernel, yet no comprehensive study has systematically evaluated their security. As exploitation techniques continuously evolve, the effectiveness of these defenses remains uncertain. Without a rigorous, security-focused analysis of existing offensive and defensive measures, researchers risk an incomplete understanding of common kernel attack vectors and the coverage of current kernel hardenings in addressing known exploitation methods (which we refer to as *defense coverage*). However, there is a lack of comprehensive research on the theoretical protective capabilities of Linux kernel hardenings, and the real-world impact of these mechanisms in production environments is unclear. Current security evaluations fail to provide a thorough offensive–defensive perspective. To address this gap, we introduce a systematic attack-decomposition framework for analyzing kernel exploitations and defenses.

We begin by examining kernel exploitation through an attacker-centric lens, dividing the process into three phases: triggering memory corruption, acquiring exploitation primitives, and achieving the goal. Through in-depth analysis of potential attack vectors in each phase, we offer a systematic framework for understanding and evaluating critical paths in the kernel exploitation. Next, we systematically categorize existing kernel defenses within this framework, assessing their coverage of critical attack vectors, inherent weaknesses, and how different mitigations interact—both redundantly and

✉Corresponding authors

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.241725
www.ndss-symposium.org

complementarily. This analysis clarifies which attack vectors remain entirely unprotected, identifies mitigations that current exploits can bypass, and measures the total number of exploits able to subvert existing protections. To address concerns of open science and reproducibility, our datasets and more detailed analysis are publicly available [1].

Finally, we evaluate the default hardening configurations of leading Linux distributions, comparing their theoretical defensive capabilities with real-world effectiveness. Our observations reveal the gap between theoretical and practical kernel security, providing the observations of our analysis along with recommendations to enhance the security of the Linux kernel and ecosystem.

As a result, we identify 23 unprotected kernel attack vectors that can be chained to exploit the latest Linux systems and note 8 frequently recurring attack paths that appear in more than 20 kernel exploits, reflecting limitations in the kernel's defense coverage. Our investigation further reveals 20 relatively secure hardening mechanisms alongside 31 insecure schemes widely deployed in practice. Among these, 12 measures defend against multiple attack vectors, with 3 classified as effective. We also identify 4 pairs of redundant hardenings that warrant integration, 3 obsolete defenses ripe for removal, and 4 pairs that must be applied in tandem for robust protection. Finally, a notable security gap emerges between upstream and downstream kernels: 4 effective protections remain underutilized, and 4 downstream distributions (out of 10) have deviant configurations.

To the best of our knowledge, this is the first systematic study to summarize Linux kernel exploitation techniques. It is also the first comprehensive research on Linux kernel hardening and rigorously evaluating its effectiveness. Our observations offer practical guidance for system configuration and provide researchers with clear directions for future hardening efforts. Drawing on real-world vulnerabilities and associated exploits, we also pinpoint the weakest existing mitigations that need to be enhanced. In summary, the paper makes three key contributions.

- **Systematic Attack Decomposition Framework:** We propose a systematic and extensible attack decomposition framework from an attacker's perspective, which divides the kernel exploitation process into three key phases. This framework enables comprehensive classification and analysis of potential attack vectors, serving as the foundation for all subsequent analyses and providing a unified reference for kernel security research.
- **Multi-Level Kernel Hardening Analysis:** We conduct a comprehensive analysis of existing kernel hardenings (51 defenses in total). By precisely mapping these protections to our proposed attack vectors, we reveal the specific defense coverage of each mechanism and its limitations, as well as redundancies and combinations between different defenses.
- **Revealing the Gap Between Theoretical and Practical Hardening Effectiveness:** Based on a detailed evaluation of downstream distributions, we identify the gap between theoretical design and practical deployment of kernel mitigations, indicating 4 underutilized hardenings and 4 distributions with inappropriate configurations. We also provide observations and recommendations to help developers avoid repeating these pitfalls in future usage.

[1]https://github.com/OS3Lab/sok-kernel-hardening

II. Study Scope & Related Work

A. Study Scope and Data Collection

Study Scope. This study evaluates Linux kernel hardening techniques aimed at defending against memory corruption vulnerabilities caused by internal software errors (e.g., race conditions). These issues account for 70% of kernel CVEs [6], making them the most prevalent and impactful threat in practice. Our analysis focuses on mitigations widely deployed in mainstream desktop and server Linux distributions and frameworks. We exclude experimental solutions (e.g., Rust for Linux) due to their lack of real-world adoption. *Our aim is to assess the real-world effectiveness of software-based kernel hardenings from both offensive and defensive perspectives.*

Threat Model. We consider local attacks where non-privileged users exploit internal kernel software vulnerabilities to compromise the kernel, such as by escalating privileges. We do not consider attacks requiring hardware-specific behaviors, privileged execution contexts (e.g., speculative execution, side-channel attacks), or filesystem-based vectors, as these require different assumptions and fall outside our software-based memory-corruption focus.

Exploitation Dataset. We collected 121 public exploits covering 102 Linux kernel vulnerabilities (one vulnerability can have several unique exploits); a breakdown is given in Tabs I to III. Our data set covers articles/write-ups from Pwn2Own/kernelCTF, Black Hat events, Linux Security Summit, top-tier academic conferences, blogs of white-hat hackers and red teams (e.g., Google Project Zero) to avoid selection biases. These exploits target the kernel through internal errors to achieve their ultimate goals, consistent with our defined threat model. It should be noted that we excluded those exploits released before 2015 simply to align with the state-of-the-art. Finally, it should also be noted that our study does not include the exploits backed by government agencies or black hat hackers, which are unavailable for public research. However, the lack of these exploits does not influence the security assessment of the mainline kernel hardenings. This is because white-hat hackers and enterprise red-teams have already unveiled many government-backed or black-hat hackers commonly adopted exploitation methods (e.g., recently disclosed watering hole attack [7]). After collecting kernel exploits, we categorize the attack process into three main phases and derive a systematic decomposition framework (Sec. III).

Hardening Dataset. We collected and studied 51 kernel hardenings from different representative sources: ❶ Kernel Self Protection Project (KSPP) [5], initiated by kernel maintainers to enhance Linux kernel security; ❷ PaX/Grsecurity [8, 9], a commercial solution to provide hardening patches, ❸ Features

highlighted by white-hat security practitioners [10] etc. In IV, we map each hardening to corresponding attack vectors, revealing their efficacy and interrelation to assess their defense coverage. Consistent with our study scope and threat model, we focus on deployable software-based kernel hardenings applicable to general-purpose Linux systems. Development-time sanitizers (e.g., `KASAN`, `KMSAN`) and vendor-specific hypervisor mechanisms (e.g., RKP [11]) are excluded due to their limited use in production environments.

Gaps between Expectation and Reality. To examine the discrepancy between theoretical and practical hardening effectiveness, we surveyed multiple major releases from the most popular downstream Linux distributions on DistroWatch [12]. We reveal a gap between the defenses' intended coverage and their real-world impact in Sec. V. Finally, we provide observations and suggestions for the gap in Sec. V and VI.

B. Related Work

While most previous works on hardening evaluation focus on performance overhead [13]–[16], in recent years, some discuss security effectiveness. [17] proposed an attack model demonstrating how the memory safety policy could be violated to perform attacks. A testing suite named CONFIRM [18] is proposed to test the compatibility and applicability of different Control Flow Integrity mechanisms. The evaluation of security guarantees of CFI is further improved in [19] where the authors developed two tools to identify runtime feasible targets of indirect control transfer and verify the effectiveness of CFI against typical attacks. Ahmed *et al.* proposed some metrics, including JIT-ROP gadget availability, quality, turing-complete expressiveness, and the upper bound of re-randomization intervals, to quantify the effectiveness of fine-grained code randomization schemes under the JIT-ROP threat model [20]. [21] analyzed 26 publicly available one-day exploits and 10 defense mechanisms targeting the Android kernel, highlighting the difference of hardening effectiveness across Android devices. Miller *et al.* [22] proposed seven exploitations that modify system registers via control-flow hijacking, including a `swapgs` based attack that achieves generic bypass of `FineIBT`.

Unlike prior works evaluating individual hardenings or downstream distributions, our work provides a comprehensive analysis of the security landscape of the upstream Linux kernel. *Specifically, we systematically assess exploitation techniques and defense mechanisms for the Linux kernel, offering a detailed analysis of the effectiveness and coverage of existing kernel hardening.* It should be noted that our three-phase attack framework does not contradict the model in [17], as they are based on different abstraction levels. We derive ours from a fine-grained analysis of attack vectors, later grouped into broader categories. The two frameworks are thus complementary rather than conflicting.

III. Attack Decomposition Framework

A. Overview

We analyze 121 publicly available exploits targeting 102 Linux kernel vulnerabilities, systematically decomposing each into phases associated with specific attack vectors (Tabs A.6–A.8). This analysis forms the attack-side foundation of our framework, which is later used in Section IV to evaluate kernel hardening mechanisms under the same phase/vector terminology. Fig. 1 presents an overview of the attack decomposition framework, by delineating kernel exploitation progression into three phases. ❶ **Phase I:** Attackers leverage an internal error (e.g., Error 8: reference miscount) to trigger memory corruption; ❷**Phase II:** By manipulating corrupted memory, attackers obtain essential exploitation primitives such as instruction pointer (IP) control or read/write (R/W) privileges; ❸**Phase III:** Exploitation primitives are used to escalate privileges, disclose sensitive information, or convert one primitive into another, ultimately fulfilling the attacker's goals. Each phase was labeled with specific attack vectors, through the meticulous analysis of hardening mechanisms detailed in Sec. IV, we classify these vectors based on their susceptibility to the kernel defenses we evaluate into three categories: 1) *Fully Blocked Vector*, that existing defenses can completely block; 2) *Partially Blocked Vector*, which can partially bypass existing defenses; 3) *Unmitigated Vector*, indicating no existing hardening we evaluate can address. In addition, we also quantify the number of vectors in each category.

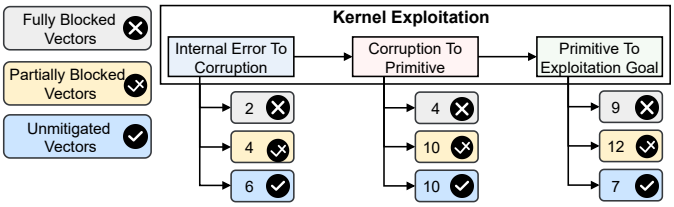


Fig. 1: Three Phases in the process of Kernel Exploitation.

To elaborate on this overview, Fig. 2 describes the comprehensive attack framework, both attack vectors and kernel defenses are shown by phase. Each directed edge links a pre-exploit error to a post-exploit outcome, with rounded rectangles representing attack vectors (e.g., "@3: Integer used as index/boundary" indicating the 3^{rd} attack vector of Linux kernel exploitations listed in Tab. I) and its defenses (e.g., "#2" indicating `PAX_SIZE_OVERFLOW` in Tab. I, the 2^{nd} kernel hardening). For fully blocked vectors, only their identifier numbers are retained in Fig. 2. The number in each rectangle, shown as X/Y, indicates that X out of Y applicable exploits can bypass kernel hardening. More details on vectors, hardenings, and exploits are provided in Tabs I, II, III.

Using the graph representation of exploitation and hardenings, we can pinpoint all exploitation chains for any given pair of pre-exploit errors and post-exploit consequences. Among them, chains composed solely of yellow rectangles reveal exploitation paths that bypass existing Linux kernel hardenings. By examining the numbers below each vector, we identify hardening mechanisms frequently bypassed, highlighting the weakest defenses. Unmitigated rectangles further indicate vectors that deserve attention for future hardening efforts. In the

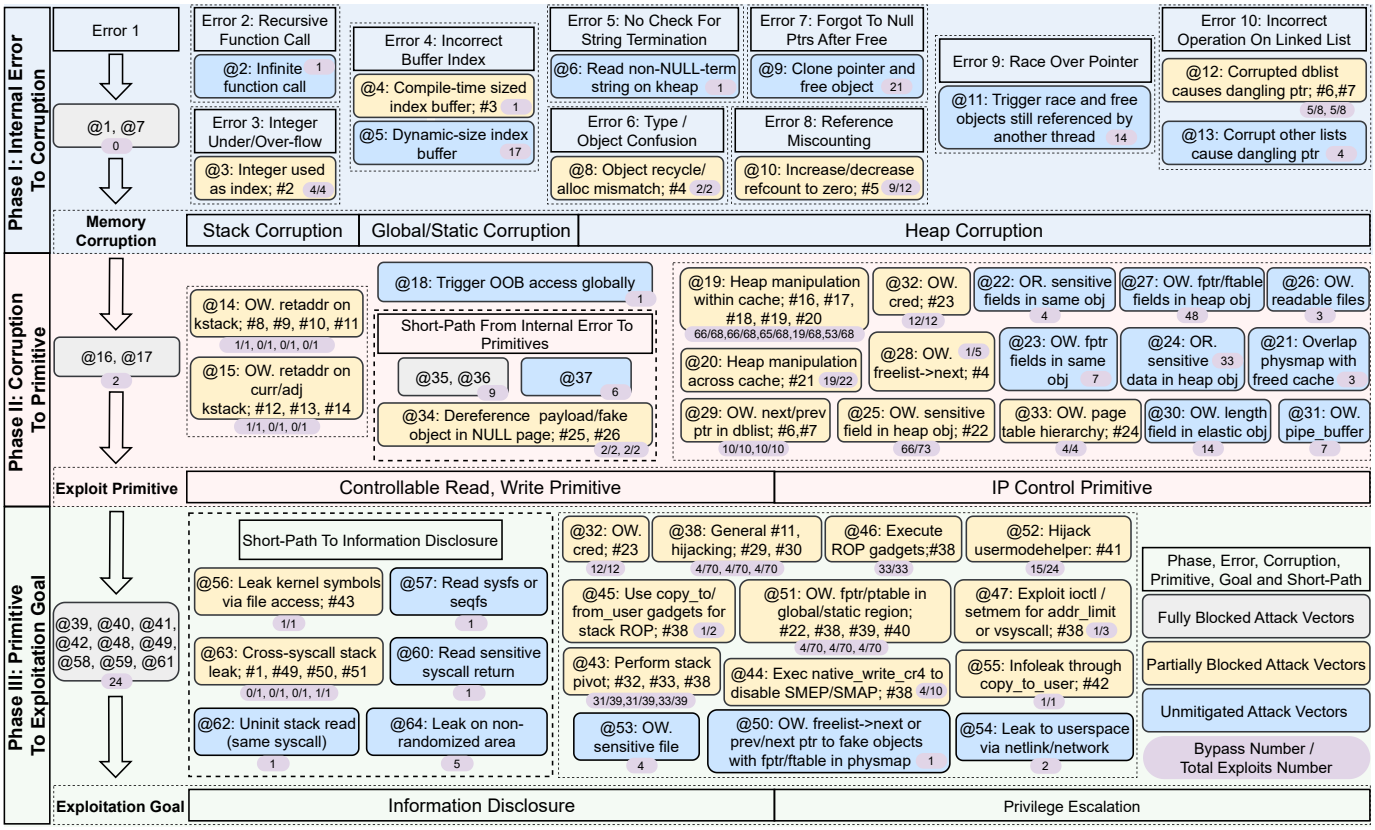


Fig. 2: Three-phase decomposition of kernel exploitations with attack vectors and their mitigation status at each phase. OR. and OW. denote Overread and Overwrite, respectively.

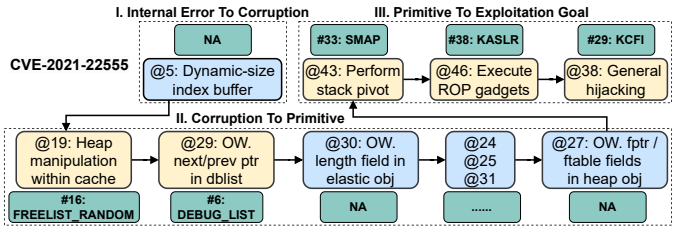


Fig. 3: Exploitation and defense decomposition example.

following, Sec. IV extends this framework to the analysis of kernel defenses.

To illustrate how a real exploit maps to the framework, Fig. 3 summarizes a representative exploit trace. Phase I: an internal error (vector @5) triggers a heap overflow that corrupts adjacent objects. Phase II: the initial overflow, combined with a crafted heap spray (@19), corrupts the doubly-linked `msg_msg` list (@29), causing two `msg_msg` objects to reference the same target `msg_msg` (victim); freeing victim converts the overflow into a UAF. The attacker then fills the freed slot of the elastic `msg_msg` (see [23]) with `sk_buff` objects and overwrites its length and other sensitive fields (@30 and @25). This results in out-of-bounds reads (@24) that leak heap addresses, which can be used to tamper with the damaged list pointers and bypass `DEBUG_LIST`(#6). Similarly, a `pipe_buffer`/`sk_buff` spray-and-read sequence can leak function pointers from a victim `pipe_buffer`, enabling recovery of kernel addresses and bypass of `KASLR`(#38). A subsequent spray can overwrite the leaked function pointer (@27 and @31), yielding an IP-control primitive. Phase III: the attacker performs a stack pivot (@43) and executes ROP gadgets (@46) to hijack kernel control flow (@38), ultimately achieving privilege escalation and container escape.

B. Internal Error To Corruption

As illustrated in Fig. 2, the first exploitation phase involves exploiting internal kernel errors to cause memory corruption, enabling unauthorized access to the kernel stack, heap, and global/static areas. We identify 10 common internal errors (9 with CWE IDs) used in kernel exploits. Below, we summarize these errors and their exploitation methods.

Error 1: Unlimited Variable-Length Array (CWE-789). The C99-standard Variable-Length Array (VLA) is allocated on the kernel stack (e.g., via `alloca`) and automatically freed upon function return. Because kernel stack space is limited (e.g., 8K on x86, 16K on x64), improperly checking the VLA length can cause stack overflow, corrupting memory regions beneath the kernel stack. As no public exploits have bypassed `PAX_MEMORY_STACKLEAK` (#1 in Tab. I), we conservatively regard this hardening as effective.

Phase	Exploitation Steps	Hardenings	Exploits	
			Bypass	CVE-ID
I - Internal Error To Corruption	@1: Length is too large and cross boundary	#1:PAX_MEMORY_STACKLEAK	0	NA
	@2: Infinite function call	NA	1	[24]
	@3: Integer used as index/boundary	#2:PAX_SIZE_OVERFLOW	4/4	[25]⋆, [26]⋆, [27]⋆, [28]
	@4: Index buffer whose size is determined at compile time	#3:CONFIG_FORTIFY_SOURCE	1/1	[29]⋆
	@5: Index buffer whose size is unknown at compile time	NA	17	[30], [31], [32], [33], e.g.
	@6: Read non-NULL terminated string on kernel heap	NA	1	[31]
	@7: Recycle objects to the wrong cache and corrupt freelist	#4:CONFIG_SLAB_FREELIST_HARDENED	0/1	[34]
	@8: Recycle objects mismatches the allocated objects	#4:CONFIG_SLAB_FREELIST_HARDENED	2/2	[35]⋆, [36]⋆
	@9: Clone pointer and free object	NA	21	[37], [38], [39, 40], e.g.
	@10: Increase/decrease refcount to zero	#5:CONFIG_REFCOUNT_FULL	9/12	[41, 42], [38], [43]⋆, e.g.
	@11: Trigger race and free objects still referenced by another thread	NA	14	[44], [45], [38], [46], e.g.
	@12: Corrupt doubly linked list and generate dangling pointer	#6:DEBUG_LIST, #7:LIST_HARDENED	5/8	[38]⋆, [42], [47], [48], e.g.
	@13: Corrupt other lists and generate dangling pointer	NA	4	[49], [50], [51], [52]

TABLE I: Exploitation steps, hardenings, and exploits in exploitation phase I. NA means no hardening or exploit. The complete list of exploits can be found in our datasets repository (noted in Sec. I).

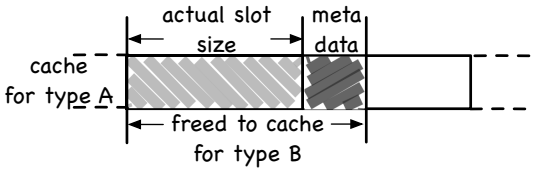
Error 2: Unsanitized Recursive Function Call (CWE-674). Stack overflow can result from unchecked recursive function calls. When the kernel stack is exhausted, corruption occurs even without VLAs. Current coding guidelines do not prohibit recursion [53], making it difficult to prevent overflow. We find no existing hardening for this error.

Error 3: Integer Underflow/Overflow (CWE-190/CWE-191). Integer wraparounds commonly trigger memory corruption in C programs, including the Linux kernel. They result from typecasting (e.g., assigning unsigned values to signed integers) or inadequate bounds (e.g., summing large values). If such underflow/overflow is used as an index (@3 in Fig. 2), the kernel can access out-of-bound memory on the stack, heap, or global/static regions. Although `PAX_SIZE_OVERFLOW` (#2) aims to detect these issues, it was bypassed in four exploits.

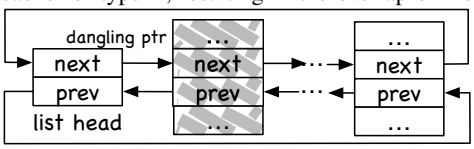
Error 4: Incorrect Buffer Index (CWE-118). Even without integer wraparounds, indexes can be miscalculated or maliciously assigned by user space. If not adequately validated, out-of-bound accesses occur (@4, @5). `FORTIFY_SOURCE` (#3) partially checks overflows at compile time but was bypassed (e.g., `CVE-2017-18344`). For runtime-sized buffers lacking static boundaries, no robust solutions exist, enabling 12 public exploits to induce corruption.

Error 5: Failing to Check a String is NULL-terminated (CWE-170). C strings must end with a NULL symbol, yet this assumption is occasionally violated in the kernel. Without explicit checks, the kernel may read beyond the intended boundary, causing corruption (e.g., `CVE-2022-0185`). Here, a missing NULL terminator causes incorrect length handling, leading to a heap overflow represented by @6, where the faulty read constitutes the internal error initiating later corruption. We find no countermeasure specifically mitigating this issue.

Error 6: Type Confusion (CWE-843). To enable functionality extensions in file systems and networking stacks, the Linux kernel uses an object-oriented design. For instance, `struct tcp6_sock` inherits from `struct tcp_sock` by embedding it as an internal field, resulting in many structurally similar types. Type confusion occurs when an object of type A (e.g., `struct tcp_sock`) is mistakenly treated as type B (e.g., `struct tcp6_sock`). This can lead to an object being recycled into an incorrect cache, allowing attackers to corrupt allocator metadata and trigger use-after-free vulnerabilities [34] (a detailed example given in Fig. 4a). Our study found that the kernel implemented cache membership validation in `SLAB_FREELIST_HARDENED` (#4) [54] to mitigate confusion.



(a) Example for Error 6: the object in type A is mistakenly recycled to the cache for type B, resulting in the overlap of metadata.



(b) Example for Error 10: the object in shadow is freed but not removed from the list.

Fig. 4: Illustrative Examples for Error 6 & 10.

Error 7: Failing to Nullify All Pointers When the Referenced Object is Freed (CWE-825). When the kernel allocates an object, multiple references may propagate (e.g., parent and child processes sharing the same socket fields). If one reference is nullified but others are overlooked after the object is freed, dangling pointers arise. Any subsequent dereference of these pointers results in use-after-free. We found no kernel defense mechanism addressing this issue.

Error 8: Reference Miscounting (CWE-911). The kernel relies on reference counters to manage the lifecycle of data objects. If developers neglect to increment counters or skip decrements, the counter may prematurely reach zero, causing the kernel to free an object that is still in use. To address this, PaX/Grsecurity introduced `PAX_REFCOUNT`, which the upstream Linux kernel adopted in version v4.13 as `REFCOUNT_FULL` (#5).

Error 9: Race Over Pointers (CWE-366). As the Linux kernel is a multi-thread system, race over pointers happens when, for example, one thread frees an object through a pointer and another thread simultaneously dereferences it. Without proper synchronization, a use-after-free can occur.

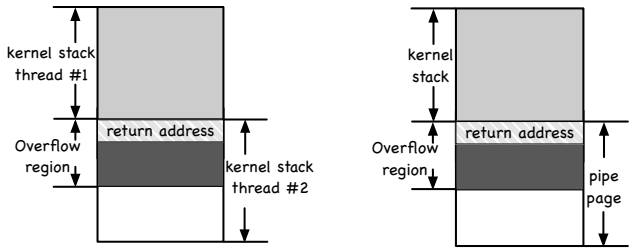
We observed 9 exploits exploiting this race condition, and no existing hardening specifically prevents this error.

Error 10: Incorrect Operation On Linked List. Linux kernel offers standard APIs for doubly and singly linked lists, yet APIs do not enforce structural integrity checks. Developers may erroneously insert an object twice, or free an object without removing it from the list—both yield dangling pointers. For instance, a freed object could still be linked in the list, rendering the `next` pointer of the list head a dangling pointer (see Fig. 4b). Such errors can be exploited (e.g., @13), yet effective mitigations are still lacking. We catalogue these unmitigated exploits in Tab. I to III and derive the following observation.

Observation 1: Unmitigated Vectors. We pinpointed 23 out of 64 attack vectors lacking corresponding mitigations. As a result, the upstream kernel widely used in Linux distributions remains unprotected, leaving these attack vectors open to in-field exploitation.

C. Corruption To Primitive

As shown in Fig. 2, the goal of second-phase exploitation is to escalate initial memory corruption into stronger capabilities—specifically, IP control or R/W privileges.



(a) Kernel stack of thread #1 overwrites that of thread #2

(b) Pipe page overwrites the stack frame

Fig. 5: Examples for overwriting return address in stack depth overflow.

1) Stack Corruption: Stack corruption proceeds in two primary ways: (1) *Stack Depth Overflow*-the kernel stack (kstack) is fully consumed, leading to writes below allocated stack space; (2) *Stack-Based Buffer Overflow*—the kernel accesses memory above the intended stack buffer. In both scenarios, attackers typically overwrite the return address.

In stack-based buffer overflows, hijacking the return address is straightforward because it resides above the overflowing buffer. In stack depth overflows, two approaches commonly arise: (1) *Cross-Process Kstack Overwrite.* Before kernel v4.9, kstacks were allocated from directly mapped pages. An attacker can spawn two threads (#1 and #2), arrange their kstacks contiguously, put #2 to sleep, and overflow its return address from #1. Resuming #2 sets the corrupted address as the PC, granting IP control. (2) *Pipe-Based Overwrite.* By allocating a pipe page directly below the kstack, attackers can overwrite the return address via writes to the pipe page.

2) Heap Corruption: To obtain primitives from heap corruption, in general, the attacker first manipulates heap layout to place critical variables within the corruption range and then triggers overread/overwrite to the critical variables. In the following, we describe the two steps sequentially.

Step 1: Heap Layout Manipulation. For out-of-bounds reads/writes, if critical variables reside in a vulnerable object's fields and lie within the corruption range, no layout manipulation is required. Otherwise, attackers place a "victim" object adjacent to the vulnerable one so that memory corruption targets the victim's critical variables—an approach called *heap fengshui* [92, 93]. In use-after-free scenarios, the attacker aims to reoccupy the freed memory region with a new object (*heap spray* [94]), allowing the new object to modify the original object's data or vice versa. In Linux, page-level memory management is handled by the buddy system, while SLAB/SLUB allocators subdivide pages into fixed-size slots. Freed slots form a linear freelist (bitmap-based in SLAB, singly linked in SLUB), and fully freed caches return pages to the buddy system. These pages directly map to physical memory, which can also be mapped into userspace via `mmap`. This design yields three principal strategies for manipulating heap layouts in the Linux kernel.

- **Intra-Cache Manipulation.** Attackers allocate and free objects in the same cache to position the vulnerable and victim objects adjacently. This layout adjustment allows corruption to target critical fields. Our analysis identifies this vector as one of the predominant attack paths, with 39 exploits leveraging this technique. To pinpoint the high-frequency vectors, we counted the exploits associated with each one and drew the following observation.

Observation 2: High-Frequency Vectors. Out of 64 attack vectors, 8 were exploited in 20+ known exploits, including heap manipulation and general hijacking. Generic objects like elastic objects, page tables, and credentials are frequent targets but lack dedicated protections.

- **Cross-Cache Manipulation (Buddy System).** Rather than remain within a single cache, attackers place the vulnerable object at the end of the pages belonging to one cache and the victim object(s) at the beginning of the pages in another cache. When these two caches lie adjacently, objects in them become adjacent as well, enabling an overread/overwrite across cache boundaries. Cross-Cache attacks can also facilitate *heap spray* by freeing vulnerable objects in bulk, returning their pages to the buddy system, and then allocating victim objects that reuse the same pages.
- **Physmap Overlap.** Because kernel heap pages map directly to the physical map (*physmap*), attackers free numerous vulnerable objects and invoke `mmap` from user space to occupy same pages. Modifying the page contents in user space overwrites freed kernel objects.

Step 2: Critical Variables Overwrite/Overread. Successful heap layout manipulation positions critical variables within the attacker's corruption range, enabling effective overwrites or overreads. Our analysis identifies seven common targets:

Phase	Exploitation Steps	Hardenings	Exploits	
			Bypass	CVE-ID
II - Corruption To Primitives	@14: Overwrite return address on kernel stack	#8:CONFIG_STACKPROTECTOR	1/1	[55]⋆
		#9:SHADOW_CALL_STACK	0/1	[55]
		#10:ARM64_PTR_AUTH, #11:PAX_RAP		
	@15: Overwrite return address in adjacent kstack or the current kstack	#12:CONFIG_SCHED_STACK_END_CHECK	1/1	[24]⋆
		#13:VMAP_STACK, #14:PAX_RANDSTACK	0/1	[24]
	@16: Overwrite restart_block.fn in thread_info	#12:CONFIG_SCHED_STACK_END_CHECK	0	NA
		#15:CONFIG_THREAD_INFO_IN_TASK		
	@17: Overwrite restart_block.addr_limit in thread_info	#12:CONFIG_SCHED_STACK_END_CHECK	0	NA
		#15:CONFIG_THREAD_INFO_IN_TASK		
	@18: Trigger Out-of-bound access globally	NA	1	[29]
	@19: Heap manipulation within cache	#16:CONFIG_SLAB_FREELIST_RANDOM	66/68	[32], [56], [37]⋆, [38]⋆, e.g.
		#17:unprivileged_userfaultfd=0	66/68	[37]⋆, [38]⋆, [57, 58]⋆, e.g.
		#18:slab_nomerge	65/68	[38], [59], [37], [45]⋆, e.g.
		#19:CONFIG_RANDOM_KMALLOC_CACHES	19/68	[60], [61], [52], [62], e.g.
		#20:CONFIG_SLAB_BUCKETS	53/68	[63], [64], [65], [66], e.g.
	@20: Heap manipulation across cache	#21:CONFIG_SHUFFLE_PAGE_ALLOCATOR	19/22	[34]⋆, [67, 68]⋆, [33], e.g.
	@21: Overlap physical memory with cache	NA	3	[48], [34], [69]
	@22: Overread sensitive fields in the same object	NA	4	[30], [31], [70], [71]
	@23: Overwrite fptr fields in the same object	NA	7	[25], [27], [62], [72], e.g.
	@24: Overread sensitive data in heap object	NA	33	[34], [60], [28], [61], e.g.
	@25: Overwrite sensitive field in heap object	#22:CONFIG_GCC_PLUGIN_RANDSTRUCT	66/73	[33], [44], [34], [32], e.g.
	@26: Overwrite readable files	NA	3	[59], [73], [51], [74], e.g.
	@27: Overwrite function pointer/table fields of structures on kernel heap	NA	48	[46], [57, 58], [47], e.g.
	@28: Overwrite freelist->next	#4:CONFIG_SLAB_FREELIST_HARDENED	1/5	[31], [56], [75], [36]⋆. e.g.
	@29: Overwrite the next and prev ptr in doubly linked list	#6:DEBUG_LIST, #7:LIST_HARDENED	10/10	[38]⋆, [49]⋆, [28]⋆, [52]⋆, e.g.
	@30: Overwrite length field in elastic object	NA	14	[32], [42], [47], [60], e.g.
	@31: Overwrite pipe_buffer	NA	7	[74], [76], [77], [78], e.g.
	@32: Overwrite cred	#23:CONFIG_DEBUG_CREDENTIALS	12/12	[34]⋆, [32]⋆, [79]⋆, [80]⋆, e.g.
	@33: Manipulate page table hierarchy	#24: PAGE_TABLE_CHECK	4/4	[81]⋆, [82]⋆, [83]⋆, [84]
	@34: Dereference payload/ fake object in NULL page	#25:CONFIG_DEFAULT_MMAP_MIN_ADDR	2/2	[85]⋆, [86]⋆
		#26:CONFIG_LSM_MMAP_MIN_ADDR		
	@35: Execute malformed eBPF program	#27: CONFIG_BPF_UNPRIV_DEFAULT_OFF>=1	0/9	[79], [80], [87], [88], e.g.
	@36: addr_limit set as KERNEL_DS in user mode	#28:TIF_FSCHECK	0	NA
	@37: Overwrite sensitive data on non-randomized cpu_entry_area stacks	NA	6	[55], [89], [90], [91], e.g.

TABLE II: Exploitation steps, hardenings, and exploits in exploitation phase II. NA means no hardening or exploit. The complete list of exploits can be found in our datasets repository (noted in Sec. I).

❶ *Function Pointers (`fptr`) and Function Tables (`ftable`).* Overwriting these pointers in the heap objects grants IP control once they are dereferenced. ❷ *Metadata Headers in Freed Slots (`freelist->next`).* This header stores addresses of subsequent freed slots. By overriding `freelist->next` with a malicious address, attackers deceive the allocator into assigning future allocations to user space or *physmap*, enabling arbitrary field modifications in newly allocated objects. ❸ *Pointers in Doubly Linked Lists (`prev, next`).* Similar to metadata headers, overwriting these pointers can link a malicious memory address as a "node", further corrupting kernel data structures. ❹ *Length Field in Elastic Objects.* Modifying length fields (e.g., [23]) misleads the kernel into overreading buffers, leaking extra data into user space. ❺ *Kernel Pipe Buffer.* This buffer includes a function pointer, a page pointer, and flags defining page attributes. Overwriting the page pointer can enable nearly arbitrary physical memory read/write. Forging the `flags` can achieve overwriting a read-only file (e.g., `/etc/passwd`) [74, 76]. ❻ *`uid` and `gid` in Credential Structures.* Overwriting these fields to zero escalates privileges, allowing the attacker to impersonate the root identity. ❼ *Page Table Hierarchy.* By redirecting page table offsets (e.g., to `commit_creds`), attackers can override checks and gain privilege without detection [95]. Alternatively, mapping a kernel text region into user space permits direct kernel code modification [81, 82], bypassing existing mitigations.

3) Short-Path To Primitives: Our analysis identifies four logic errors that grant exploitation primitives without causing explicit memory corruption: ❶ **NULL Pointer Dereference.** When the kernel dereferences a NULL pointer, attackers can map the zero address in user space with read/write/execute permissions, injecting malicious payloads. If the pointer references a function or function table, control flow can be hijacked on dereference. ❷ **eBPF Verifier Gaps.** eBPF runs user-provided programs in a kernel-level virtual machine, guarded by a verifier that simulates eBPF code before execution. Semantic discrepancies between simulation and actual runtime allow attackers to bypass sanity checks and execute arbitrary eBPF payloads, thereby obtaining both IP control and controllable R/W capabilities. ❸ **Unbalanced `set_fs`.** The `set_fs` function redefines `addr_limit` for data transfers between kernel and user space. This function shall be used in pairs. Otherwise, attackers can specify kernel addresses as data-transfer destinations, thereby achieving a controllable R/W primitive. ❹ **Non-randomized Area.** If `KASLR` fails to randomize areas like `cpu_entry_area` [89], attackers can manipulate kernel data at fixed addresses. By triggering an Interrupt Stack Table (IST) exception (e.g., via a hardware breakpoint), they can overwrite saved registers in `cpu_entry_area` with a payload (e.g., fake function tables).

D. Primitive To Exploitation Goal

After acquiring exploitation primitives in Phase II, attackers achieve ultimate goals, such as, primitive translation, privilege escalation, and critical information disclosure.

1) Start From IP Control: Once attackers acquire the IP control primitive, they gain extensive control over kernel execution. By hijacking IP, they direct the kernel toward "gadgets" that either escalate privileges directly or furnish additional controllable R/W capabilities.

Gadgets For Privilege Escalation. Attackers can hijack control flow to execute specialized "gadgets" that escalate

Phase	Exploitation Steps	Hardenings	Exploits	
			Bypass	CVE-ID
III - To Ultimate Exploitation Goal	@38: General hijacking	#11:PAX_RAP, #29:KCFI, #30:FineIBT	6/70	[49], [75], [28], [87]⋆, e.g.
	@39: Execute shellcode on kernel heap	#31:CONFIG_STRICT_KERNEL_RWX	0	NA
	@40: Execute shellcode in userspace	#32:PAX_MEMORY_UDEREF, #33:SMAP/PAN	0/12	[25], [85], [38], [24], e.g.
		#34:SMEP/PXN, #35:KPTI		
	@41: Execute shellcode in physmap	#36:CONFIG_DEBUG_WX	0	NA
	@42: Execute shellcode in BPF memory	#37:GRKERNSEC_JIT_HARDEN	1/3	[96], [83], [97]⋆
	@43: Perform stack pivot	#32:PAX_MEMORY_UDEREF, #33:SMAP/PAN	31/39	[60]⋆, [28]⋆, [61]⋆, [62]⋆, e.g.
		#38:KASLR	33/39	[28]⋆, [61]⋆, [27]⋆, [34]⋆, e.g.
	@44: Execute run_cmd and native_write_cr4 (disable SMEP/SMAP)	#38:KASLR	4/10	[44]⋆, [31]⋆, [46]⋆, [33]⋆, e.g.
	@45: Execute copy_to/from_user gadgets and perform stack ROP	#38:KASLR	1/2	[75]⋆, [55]
	@46: Execute ROP gadgets	#38:KASLR	33/33	[55]⋆, [89]⋆, [98]⋆, [99]⋆, e.g.
	@47: Execute kernel_sock_ioctl or set_memory_rw	#38:KASLR	1/3	[34]⋆, [41, 42], [45]
	@48: Execute malicious eBPF program	#27: CONFIG_BPF_UNPRIV_DEFAULT_OFF>=1	0/3	[87], [35], [100]
	@49: Overwrite freelist->next or prev/next ptr to fake objects with fptr/ftable in userspace	#32:PAX_MEMORY_UDEREF	0/4	[38], [38, 101], [56], e.g.
		#33:SMAP/PAN	0/4	
	@50: Overwrite freelist->next or prev/next ptr to fake objects with fptr/ftable in physmap	NA	1	[75]
	@51: Overwrite and dereference fptr/ftable in global/static region	#22:GCC_PLUGIN_RANDSTRUCT, #38:KASLR	1/3	[31]⋆, [67, 68], [49]
		#39:post-init read-only memory		
		#40:PAX_CONSTIFY_PLUGIN	3/3	[31]⋆, [67, 68]⋆, [49]⋆
	@52: Hijack call_usermodehelper_exec	#41:CONFIG_STATIC_USERMODEHELPER	15/24	[45], [79], [36]⋆, [98]⋆, e.g.
	@32: Overwrite cred	#23:CONFIG_DEBUG_CREDENTIALS	12/12	[88]⋆, [102]⋆, [73]⋆, [51]⋆, e.g.
	@53: Overwrite sensitive files(/etc/passwd, e.g.)	NA	4	[103], [76], [73], [51]
	@54: Read data to userspace via netlink and general networking	NA	2	[32], [30]
	@55: Read data to userspace via copy_to_user	#42:PAX_USERCOPY	1/1	[34]⋆
	@56: Read /proc/kallsyms, /boot, /lib/modules and other files including kernel symbols	#43:GRKERNSEC_HIDESYM	1/1	[104]⋆
	@57: Read sysfs or seqfs	NA	1	[31]
	@58: Read the kernel syslog	#44:GRKERNSEC_DMESG	0/4	[44], [33], [46], [27]
	@59: Read ptr value in /proc interface	#45:kptr_restrict	0/1	[105]
	@60: Read sensitive data returned from syscall	NA	1	[106]
	@61: Read sensitive data before heap slot recycle	#46:init_on_alloc, #47:init_on_free	0	NA
		#48:CONFIG_PAGE_POISONING		
	@62: Read sensitive data on stack within the same system call	NA	1	[107]
	@63: Read sensitive data remained in the last system call	#1:CONFIG_GCC_PLUGIN_STACKLEAK	0/1	[108]
		#49:CONFIG_INIT_STACK_ALL_PATTERN		
		#50:CONFIG_INIT_STACK_ALL_ZERO		
		#51:PAX_MEMORY_STRUCTLEAK	1/1	[108]⋆
	@64: Overread sensitive data on non-randomized cpu_entry_area stacks	NA	5	[55], [89], [109], [110], e.g.

TABLE III: Exploitation steps, hardenings, and exploits in exploitation phase III. NA means no hardening or exploit. The complete list of exploits can be found in our datasets repository (noted in Sec. I).

privileges. One common approach is to prepare shellcode in the kernel heap, user space, *physmap*, or eBPF memory, and redirect execution to this shellcode.

However, faced with multiple hardening techniques, attackers often opt for *kernel code reuse*. In particular, if PAX_MEMORY_UDEREF or SMAP/PAN (#31, #32 in Tab. III) is absent, an attacker may pivot the stack [111]—so that user-space payload can control kernel execution or jump to a function such as run_cmd to spawn a root shell. Additionally, disabling certain hardenings is another tactic: by hijacking the control flow to native_write_cr4, attackers can disable SMAP/SMEP altogether (@44 in Tab. III). Further refinement of this strategy appears in KEPLER [112]. Attackers start with an IP control obtained through heap corruption, redirecting execution to copy_to_user to leak the stack canary (i.e., CONFIG_STACKPROTECTOR), then use copy_from_user to inject a ROP payload (along with the leaked canary) onto the kernel stack. During these copy_to/from_user functions, SMAP/SMEP is temporarily disabled, allowing the ROP injection to bypass existing mitigations.

Gadgets For Controllable R/W. In addition to executing malformed eBPF programs, attackers can leverage existing kernel code gadgets to gain a controllable R/W primitive. By invoking set_fs, the addr_limit boundary between user space and kernel space can be reset to its maximum value, thereby permitting writes to kernel space. Furthermore, invoking set_memory_rw enables modification of the permissions for the vsyscall segment, which is normally mapped as executable. Changing these permissions to R/W allows crafting a fake ctl_table object within the vsyscall segment, with its data field pointing to the target address for reading or writing. Subsequently, hijacking control flow to invoke register_sysctl_table dereferences the data field, achieving a controllable R/W primitive.

Beyond these functions, attackers can also resort to ROP gadgets for R/W operations. For instance, when rdx and rsi are under attacker control, a gadget such as mov [rdx], rsi; ret can write data from rsi to the address specified in rdx. Conversely, a gadget like mov rax, [rdx]; ret can read kernel data when rdx is similarly controlled.

2) Start From Controllable R/W: **To IP Control.** When attackers possess a controllable R/W primitive, they can escalate to IP control by overwriting and dereferencing function pointers (fptr) or pointers to function tables (ftable). Based on their locations, these pointers fall into three categories:

- **Pointers in the User Space.** By corrupting metadata headers (e.g., prev and next) or other structures that reference user-space memory (see Sec. III-C2), the attacker can create a *fake object* in user space and link it into the kernel's freelist or doubly linked list. Since user-space pointers are easily manipulated, any future dereference by the kernel can be hijacked, leading to IP control.
- **Pointers in the Physical Map (physmap).** The *physmap* is a kernel region directly mapped to physical memory. Attackers can first map a page frame, craft a fake object within it,

then release the page back to the kernel, effectively injecting the fake object. This technique evades defenses that rely solely on restricting user-space references.

• **Pointers in the Global/Static Region.** Finally, attackers can overwrite single pointers or pointer fields in global/static data structures if their addresses are known. Although KASLR (#37 in Tab. III) randomizes kernel addresses and GCC_PLUGIN_RANDSTRUCT (#22) randomizes field order in structures, both measures can be undermined by information leaks or other evasion techniques (see Sec. III-C2). Once these pointers are corrupted and dereferenced, attackers gain the IP control primitive, enabling arbitrary code execution or other privilege escalation steps.

To Privilege Escalation. A straightforward method for privilege escalation involves overwriting the uid and gid fields within a process's credential structure. Specifically, after acquiring a R/W primitive with global access, an attacker can locate the init_pid_ns variable and traverse the task radix tree to find the current task_struct. Once found, overwriting the credential within task_struct grants elevated privileges.

Privilege escalation can also be achieved by hijacking call_usermodehelper_exec (@52), which executes commands with root privileges. Three critical kernel variables are involved in this hijack. First, by overwriting the static variable modprobe_path, an attacker can corrupt the path of the binary loaded for helper calls, thereby creating a root-privileged process. In response, STATIC_USERMODEHELPER (#41) was introduced to force user-mode helper calls to a single predefined binary. Second, modifying the static binfmt list allows attackers to manipulate the binary format handler search process, potentially gaining IP control. Third, overwriting core_pattern enables one to execute a malicious command as root by triggering a crash.

Finally, privilege escalation can also be accomplished by modifying sensitive read-only files, such as /etc/shadow, which governs user privileges.

To Information Disclosure. By leveraging a read primitive, attackers can exfiltrate kernel data to user space through channel functions. According to prior work [23], three main categories of channel functions exist: copy_to_user, helper functions in the netlink module, and general networking functions. Each typically requires three arguments: a data source, a destination, and a data length. By specifying the attacker-controlled kernel address as the source argument, sensitive information is disclosed to user space.

3) Short Paths To Information Disclosure: Three primary techniques can achieve information disclosure without relying on IP control or controllable R/W primitives:

Unrestricted Access Control. Although the Linux Security Module (LSM) system (e.g., SELinux, AppArmor) enforces strict policies, kernel pointers often appear, in cleartext, within files such as /proc/kallsyms, /boot, /lib, syslog, and various sysfs/seqfs/notes entries [113]. Attackers can simply read these pointers to bypass randomization-based defenses (e.g., KASLR). In some cases [106], sensitive data is also inadvertently returned to user space by a system call.

Uninitialized Heap/Stack. When variables remain uninitialized, residual data persists in their memory region. Attackers can systematically prepare sensitive data in memory and then leak it via the channel functions discussed in Sec. III-D2. On the heap, sensitive objects are allocated, freed, and reallocated as an uninitialized object with identical offsets, exposing previous data. On the stack, a "stack spray" [114] approach prepares sensitive data with one system call, then leaks it through another. More recently, eBPF stacks (part of the kernel stack) have been used to facilitate stack spray [115].

Non-randomized Area. As detailed in Sec. III-C3, attackers can overwrite saved registers on cpu_entry_area stacks [89], which also enables information disclosure. For example, corrupting rcx, which specifies the data size for copy_to/from_user, allows control over the amount of data transferred between kernel and user space.

IV. Multi-Level Analysis of Kernel Hardening

To evaluate the security effectiveness of kernel defenses, we propose a multi-level analysis of kernel hardenings, *i.e.,* categorize kernel hardenings based on our decomposition framework and analyze their defenses against various vectors, highlighting each technique's strengths and limitations.

As shown in Fig. 6, we categorize kernel mitigation mechanisms based on their design objectives: ❶ *Isolation-based Protections*: focus on segregating data transfers and sensitive operations between privileged and non-privileged spaces, restricting non-privileged user actions; ❷ *Control Flow Integrity*: aim to prevent unintended kernel code modifications and enforce control flow integrity; ❸ *Data Flow Integrity*: protect critical variables from malicious corruption and ensure that their usage remain within the intended boundaries; ❹ *Entropy-based Protections*: increase unpredictability in code, stack-/heap layouts, sensitive structures, and etc.; ❺ *Miscellaneous Protections*: cover a range of hardenings, with many preventing sensitive information leakage and others targeting specific vulnerabilities. By mapping these mechanisms to the above-summarized attack vectors, Fig. 6 provides a new perspective on the defense coverage across various attack paths. Besides, each hardening's effectiveness is labeled as "*effective*" (unable to be bypassed by any collected exploit), "*moderate*" (partially bypassed), or "*unsafe*" (completely bypassed). We also highlight if a hardening can counter multiple attack paths, revealing its strengths and limitations, as detailed in Sec. IV-A3.

Observation 3: Effective Hardening. We identified 20 effective hardening blocking these exploits in our study. Though not flawless, they provide meaningful protections, primarily through isolation and permission enforcement. Out-of-tree mechanisms, such as PAX_RANDSTACK offer strong protection but remain unavailable to most distributions due to their lack of upstream integration.

A. Defense Coverage Analysis

We analyze kernel defenses across the exploitation framework and organize coverage by phase and vector to clarify the relationship between exploitations and defenses.

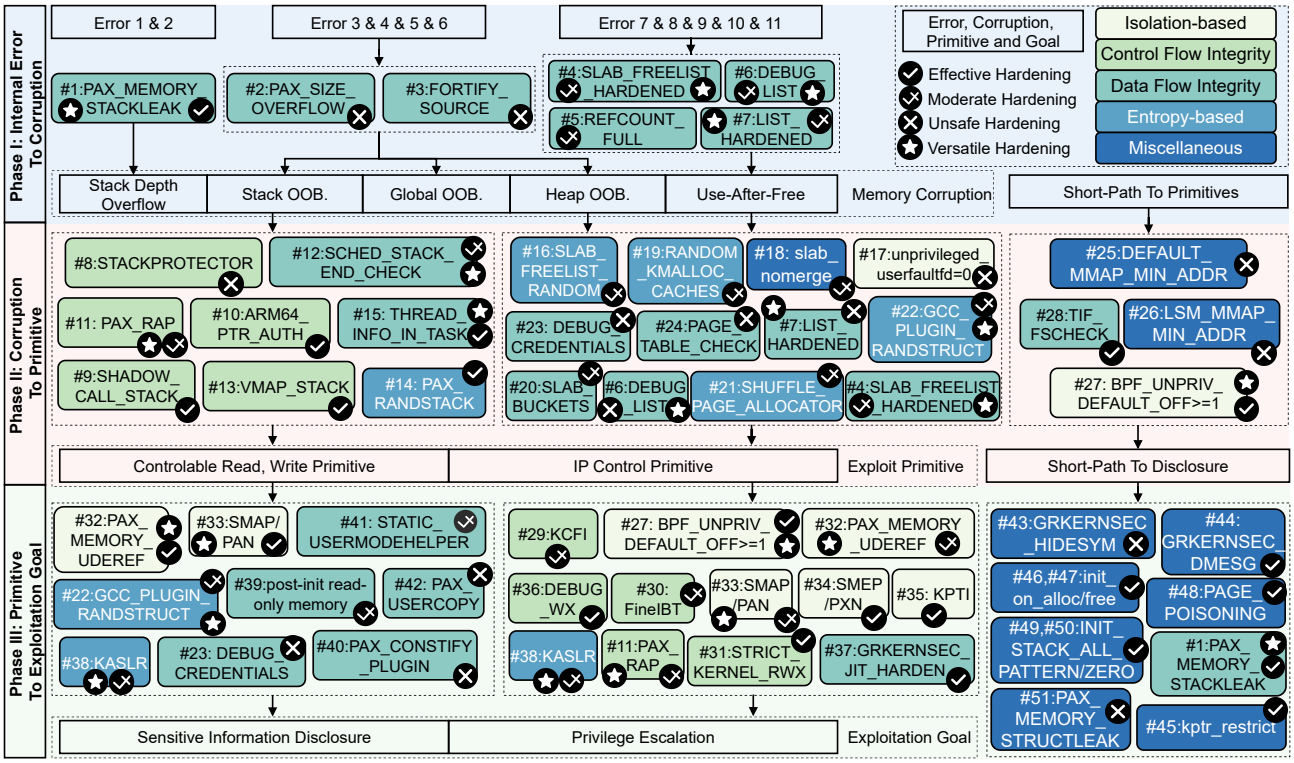


Fig. 6: The taxonomy and defense coverage of kernel hardenings. OOB. denotes Out-of-Bound.

1) Mitigating Internal Error To Corruption: Eliminating internal errors during the early stages of an attack is the most effective approach to preventing kernel exploitation. Six hardenings can achieve this objective. To prevent stack depth overflow errors (@1 in Fig. 2), PaX/Grsecurity [8] introduced `PAX_MEMORY_STACKLEAK` (#1 in Fig. 2 and 6), which checks stack allocations to prevent overflow. However, this feature was rejected by upstream kernel developers due to the removal of VLAs in v4.20 [116]. No public exploits in our analysis can bypass `PAX_MEMORY_STACKLEAK`, thus we assume it is effective.

To prevent integer underflow/overflow (@3), `PAX_SIZE_OVERFLOW` (#2) checks function arguments for potential overflows by doubling integer precision. This is effective for errors caused by careless calculations but cannot defend against overflows resulting from typecasting (e.g., `CVE-2016-9793`) or unpassed arguments (e.g., `ASA-2018-00053` [26], `CVE-2017-1000112` [27]).

To detect incorrect buffer index errors (@4), `FORTIFY_SOURCE` (#3) checks buffer overflows in memory operations (e.g., `memcpy`, `strcpy`). However, it cannot handle cases where invalid indices are used directly (e.g., `CVE-2017-18344` [29]).

As discussed in Sec. III-B, type confusion errors often cause misalignment and allocator metadata corruption (@7, @8). `SLAB_FREELIST_HARDENED` (#4) protects metadata integrity by encrypting the freelist header. However, it can be bypassed if the heap secret and object address leak [36].

To mitigate unbalanced reference counters (@10), `REFCOUNT_FULL` (#5) validates reference counts on increment/decrement. However, this approach assumes matching increment and decrement operations, which may not always be the case. If an increment operation is missing and the counter reaches zero, the kernel will free the object still in use, without being detected by `REFCOUNT_FULL` [43]. Additionally, arbitrary write primitives can be exploited to forge malicious refcounts.

To protect standard doubly linked list (@12), `DEBUG_LIST` (#6) and its lightweight subset `LIST_HARDENED` (#7) were introduced to perform validation. The superset `DEBUG_LIST` poisons the `prev` and `next` pointers when the list entry is deleted and detects the pointers corruption when adding/deleting list entries. However, in our analysis, we find that this defense is insufficient. Attackers can exploit the time window between corruption and detection to manipulate the list (e.g., `CVE-2016-10150` [38]). Comprehensive protection should be combined with another hardening, for example, `SMAP` as we will discuss in Sec. IV-A2. Furthermore, similar protections are lacking for singly-linked and developer-customized lists. We counted the cases of such incomplete hardening and drew the following observation:

Observation 4: Incomplete Hardening. Out of 51 mitigations, 31 were found to be moderate or unsafe, meaning they can be bypassed by at least one public exploit. While they raise the exploitation bar, they remain vulnerable to certain attacks. Our observation just represents a lower bound of effectiveness, as unpublished exploits may exist.

To understand why these mitigations are incomplete, we extracted hardening bypass techniques from 121 publicly disclosed exploits and identified three common scenarios that undermine kernel defenses: ❶ **Invariant violations.** Security hardenings can be bypassed if an attacker violates the key invariants on which the defense relies. Our study identifies 10 such violations. For example, KCFI assumes that the kernel code remains unmodified. However, page table hierarchy attacks [95] break this assumption, allowing attackers to manipulate kernel text, thus redirecting execution. ❷ **Breaking the integrity of hardening itself.** When security hardenings thwart an exploit, attackers often pivot to compromising them by either (1) permanently disabling the active protections or (2) manipulating the critical components on which the mitigations rely, subverting their behavior to aid the exploitation. We identified 20 such violations. ❸ **Abusing the rationale behind hardening.** Once an exploit leaks or controls the metadata required by the verification process, the mandatory sanity check fails to detect the unexpected behavior, regardless of how thoroughly the mitigation is designed and implemented. 5 such violations are identified in our study. Details are provided in Tab. IV.

Bypass Techniques	Hardenings
Invariant violations	#8, #11, #12, #20, #22, #23, #29, #30, #38, #41
Breaking the integrity	#2, #3, #5, #6, #7, #8, #16, #17, #18, #19, #22, #23, #25, #26, #38, #39, #40, #42, #43, #51
Abusing the rationale	#4, #32, #33, #34, #38

TABLE IV: Three common scenarios of hardening bypass techniques.

2) Defense Corruption To Primitive: **Mitigating Stack Corruption.** As analyzed in Sec. III-C1, stack corruption attacks primarily focus on corrupting the return address (@14, @15 in Fig. 2). Several hardenings protect the return address and ensure backward-edge integrity. STACKPROTECTOR (#8 in Fig. 2 and 6) has been a long-standing feature in the Linux kernel, enhanced over time by increasing the random range to 64 bits and setting the stack canary as per-task [117]. On ARM64, SHADOW_CALL_STACK (#9) uses a shadow stack to store the return address securely. ARM64_PTR_AUTH and PAX_RAP (#10 and #11) encrypts the return address with hardware and software techniques, respectively.

In addition to these, SCHED_STACK_END_CHECK (#12), derived from GRKERNSEC_KSTACKOVERFLOW in PaX/Grsecurity, prevents stack depth overflows by placing a magic constant at the end of the kernel stack and checking its integrity during thread scheduling. However, this can be bypassed if corruption is controllable [24]. To mitigate this, VMAP_STACK (#13) ensures non-contiguous kernel stack mappings, making it harder for attackers to predict stack locations.

Furthermore, PAX_RANDSTACK (#14) randomizes the kernel stack offset on every system call, hindering the attacker's ability to guess the return address location. A similar feature, RANDOMIZE_KSTACK_OFFSET_DEFAULT, has been merged into the upstream Linux kernel [118]. However, its low entropy (5 bits [119]) makes it vulnerable to attacks using hardcoded offsets and ROP gadgets [120]. If panic_on_oops is not enabled, this protection can be bypassed by repeatedly creating exploit threads with tailored payloads.

Given the bypassability of SCHED_STACK_END_CHECK, THREAD_INFO_IN_TASK (#15) further mitigates risk by relocating thread_info into task_struct, ensuring that the kernel stack does not overlap with thread_info.

Mitigating Heap Layout Manipulation. To prevent heap fengshui within the cache (@19), SLAB_FREELIST_RANDOM (#16) randomizes the order of slots in newly created caches, making it harder for attackers to predict object placement. Although this hardening complicates heap fengshui, it can be bypassed by techniques such as heap grooming [31, 121], where attackers allocate objects with critical fields, free some to leave "holes", and later allocate vulnerable objects to fill those holes.

For heap spray prevention within cache (@19), we found RANDOM_KMALLOC_CACHES (#19) to be effective. It creates multiple independent copies of slab caches, chosen randomly based on the address of the calling kmalloc and a per-boot random seed, preventing heap objects from being sprayed into the same cache. However, it can be bypassed if the allocation size exceeds KMALLOC_MAX_CACHE_SIZE [122]. Details of the similar SLAB_PER_SITE [123] hardening are omitted, as it offers redundant protection and remains out-of-tree. Other hardenings, while primarily targeting specific exploitation techniques, also effectively suppress heap spraying within caches. The unprivileged_userfaultfd restriction (#17) limits the access of userfaultfd syscall by unprivileged users, thereby preventing its misuse to freeze objects and win race conditions during layout manipulation. The slub/slab_nomerge boot parameter (#18) disables cache merging, reducing the attack surface by preventing the creation of large, predictable allocation regions. Finally, SLAB_BUCKETS (#20) protects sensitive objects (e.g., struct msg_msg) from heap spraying by isolating them into dedicated caches.

For cross-cache manipulation (@20), the defense SHUFFLE_PAGE_ALLOCATOR (#21) randomizes the allocation of physical pages on a 4MB granularity. While originally intended as a performance optimization, it can hinder cross-cache attacks [124]. However, it does not fully protect against issues like CVE-2018-9568, where kernel objects are recycled into the wrong cache, nor does it prevent heap spray attacks based on cross-cache manipulation [81]. While SLAB_VIRTUAL [125] is designed to mitigate this attack, we exclude it from our scope as it remains out-of-tree.

Mitigating Heap Critical Variables Overwriting. Once the heap layout aligns with the attacker's expectations, sensitive fields in heap objects are vulnerable to overwriting (@25).GCC_PLUGIN_RANDSTRUCT (#22) randomizes the field order within data structures at compile time. This makes it more difficult for attackers to predict the offsets of critical fields. However, this protection relies on a random seed, which, if exposed (as required by Linux distributions for third-party kernel modules), compromises security [126]. Consequently, we consider this defense unsafe, and most distributions do not enable it, as observed in our analysis.

In addition, we identify 7 critical heap variables frequently overwritten in kernel exploits (@27–@33), with 4 corresponding hardenings aimed at protecting them: (1) SLAB_FREELIST_HARDENED, as previously mentioned, safeguards metadata headers, protecting the second critical target. (2) To protect the third target, the prev and next pointers, DEBUG_LIST is intended to help. However, as discussed in Sec. IV-A1, this hardening is insufficient on its own. Attackers can exploit timing gaps to fabricate a complete doubly linked list, meaning this defense must be combined with others to be effective. For example, combined with SMAP/SMEP, the attacker cannot easily counterfeit a complete list in user space and bypass it. (3) For the uid and gid fields in the credential structure, DEBUG_CREDENTIALS (#23) introduces a magic number before the uid field. The kernel checks that this number is not corrupted when using the credentials. However, since this number is constant, attackers can easily bypass this defense by controlling the overwriting value. As a result, DEBUG_CREDENTIALS has been removed from the upstream Linux kernel due to its ineffectiveness. (4) PAGE_TABLE_CHECK (#24) protects page table objects (the seventh target) by checking for double mappings when entries are added or removed. However, this defense is limited, as it cannot detect corruption of page tables during their usage.

Mitigating Short-Path From Internal Error To Primitives. DEFAULT_MMAP_MIN_ADDR (#25) and LSM_MMAP_MIN_ADDR (#26) prevent the mapping of addresses below mmap_min_addr [127]. However, we found that NULL pointer dereference attacks remain possible when two conditions are met: a logic error maps NULL pages, and a NULL pointer dereference occurs [85, 86]. Thus, we consider this hardening insufficient. To secure eBPF programs, the kernel includes bpf_jit_harden (#36) for enhanced verification and BPF_UNPRIV_DEFAULT_OFF (#27) to restrict unprivileged access, reducing the risk of eBPF-based attacks. For unbalanced set_fs operations, the kernel introduces TIF_FSCHECK (#28), which sets a flag in thread_info. This flag is checked when returning to user space, and terminate the process if uncleared. However, a time window exists between the unbalanced set_fs and the check, leading exploitation in this gap [128]. In response, kernel developers removed all instances of set_fs in v5.10-rc1 [129], addressing this vulnerability. With the removal of set_fs, the TIF_FSCHECK is now obsolete. Counting all such cases, we find three deprecated hardening schemes.

Observation 5: Obsolete Hardening. Three protections are no longer effective, yet two still appear in KSPP and are used as bypass targets. PAX_MEMORY_STACKLEAK lost relevance after VLAs were removed in v4.20; TIF_FSCHECK became redundant with the elimination of set_fs in v5.10-rc1; and DEBUG_CREDENTIALS was dropped in v6.7.

3) Mitigating Primitive To Exploitation Goal: **Mitigating Controlable R/W.** To prevent adversaries from elevating a controllable R/W primitive to IP control, the Linux kernel implements various hardening strategies:

• **User-Space Access Protections.** (1) PAX_MEMORY_UDEREF (#32 in Fig. 2 and 6), SMAP/PAN (#33), and SMEP/PXN (#34) prohibit the kernel from accessing user-space memory, thwarting fake-object injection from user space. (2) KPTI (#35) was originally introduced to mitigate the Meltdown vulnerability. It also segregates user-space and kernel-space page tables, impeding the kernel's ability to reference user-space pointers.

• **Global/Static Pointer Protections.** (1) KASLR (#38) randomizes kernel addresses to obscure the precise location of global/static pointers. However, its low entropy makes it susceptible to information disclosure attacks. (2) GCC_PLUGIN_RANDSTRUCT (#22) adds another layer of uncertainty for attackers seeking to overwrite pointer fields. Despite this, attackers can still bypass it once they discover the randomization seed or perform targeted leaks. (3) post-init read-only memory (#39) and PAX_CONSTIFY_PLUGIN (#40) preserve certain function pointers or tables as read-only either after system boot (*post-init*) or from compilation time (*constify*). While they effectively protect a subset of pointers, they cannot cover kernel structures that must remain writable for runtime modification.

Mitigating IP Control. A key strategy to mitigate attacks involving IP control is enforcing CFI. Mitigations like KCFI (#29) [130] protect the forward-edge by only enabling transfer to the entry of targeted functions. Similarly, PAX_RAP (#11), introduced by PaX/Grsecurity, follows a similar design, while encrypts return addresses. Another forward-edge CFI scheme, FineIBT (#30) [131], relies on Intel CET [132], employing instruction ENDBR to mark valid targets for each indirect call/jump. However, since eBPF instructions execute just-in-time within a virtual machine, these defenses do not address eBPF-based IP control.

Besides, Linux kernel offers several hardening features to prevent shellcode execution and code-reuse attacks: ① STRICT_KERNEL_RWX (#31): Enforces data execution prevention (DEP) in the kernel by disallowing executable kernel heap regions. ② PAX_MEMORY_UDEREF, SMAP/PAN, SMEP/PXN, KPTI (#32, #33, #34, #35): Block the kernel from accessing or executing user-space memory, mitigating attacks that rely on user-space payloads. ③ DEBUG_WX (#36): Alerts when pages are set to both writable and executable at boot, thwarting shellcode execution in *physmap*. ④ GRKERNSEC_JIT_HARDEN (#37) and bpf_jit_harden: Harden JIT-compiled eBPF code by preprocessing instructions and "blinding" constants, thereby preventing attackers from injecting shellcode through eBPF programs. SMAP exemplifies broad hardening beyond narrow countermeasures, motivating us to measure how often similarly versatile defences appear.

Observation 6: Versatile Hardening. We identified 12 of 51 hardening techniques that can prevent multiple attack vectors, such as SMAP/PAN. By addressing design flaws across multiple vectors, these hardenings achieve broad coverage but exhibit varying effectiveness across different paths, ranging from "effective" to "moderate" or "unsafe."

Mitigating Information Disclosure. To prevent leaking, in `copy_to_user` function, `PAX_USERCOPY` (#42) from PaX/Grsecurity enforces checking that the data length should not exceed the size of heap slot or stack frame size. Similarly, `HARDENED_USERCOPY` does the same thing. However, this checking is not restrictive enough because sensitive data can reside in the heap slot as well as current stack frame. Besides, this hardening only employs checking in `copy_to_user` function. Another two channels are not covered.

Unrestricted Access Control. To mitigate sensitive information leakage, `GRKERNSEC_HIDESYM` (#43) restricts access to `/proc/kallsyms`, `/boot`, and `/lib/` files to privileged users. Similarly, `GRKERNSEC_DMESG` (or `dmesg_restrict` in the upstream Linux kernel, #44) adds privilege checks in `do_syslog`, allowing only users with `CAP_SYSLOG` or `CAP_SYS_ADMIN` to read syslog. `kptr_restrict` (#45) introduces the `%pK` format specifier in the `printk` function to obfuscate pointer values when printed. To prevent uninitialized heap leakage, `init_on_alloc/free` (#46, #47) zero out kernel object slots upon allocation and deallocation, ensuring that sensitive data is not leaked. `PAGE_POISONING` (#48) similarly poisons heap slots with a magic number upon deallocation. For uninitialized stack protection, four hardenings are used. `INIT_STACK_ALL` (#49,#50) erases the entire kernel stack, while `GCC_PLUGIN_STACKLEAK` (#1) clears only the stack space used during the current system call before returning to user space. However, these measures do not fully mitigate uninitialized stack leakage in single system calls, as sensitive data can be leaked in one step (e.g., [107]). `PAX_MEMORY_STRUCTLEAK` (#51), requires developers to annotate structures containing potentially readable data to user space, ensuring initialization of these variables. However, it relies on user annotations, which may be incomplete, as evidenced by the exploit of `CVE-2017-7616` [108]. This functional overlap among protections indicates that several schemes end up addressing the same attack vector, motivating the following observation.

Observation 7: Redundant Hardening. We distinguished 4 pairs of hardening, each targeting the same attack path. Specifically, `INIT_STACK_PATTERN/ZERO` and `GCC_PLUGIN_STACKLEAK` both mitigate stack information disclosure, but differ in implementation.

V. Kernel Hardening Deployment in the Wild

To assess real-world adoption of kernel hardening, we empirically profile ten popular distributions (ranked by DistroWatch [12]). Our study offers: ❶ a cross-sectional comparison across distributions; ❷ a temporal trend analysis of their hardening deployment; and ❸ a comparison with the upstream kernel. To better understand these deployment strategies, we contacted these downstream vendors to investigate the causes of the gap. However, we received only responses from the administrators of `Fedora`, `MX`, and `openSUSE` thus far. Note that the goal of this study is to foster a healthier Linux kernel ecosystem, not to make security judgments about specific distributions.

Downstream VS Downstream. For our analyses in this section, we collected a data snapshot in June of each year from 2023 to 2025. This three-year period covers three consecutive LTS kernel cycles: v6.1 (Dec 2022), v6.6 (Oct 2023) and the recently designated v6.12 (Nov 2024). This means that each snapshot reflects a stable, industry-supported baseline, while avoiding obsolete branches. This multiyear dataset allows us to perform a cross-sectional analysis using the most recent data, while also providing a view of temporal trends to mitigate single-point-in-time biases. For each snapshot, we installed the latest official x86_64 image, upgraded to the newest stable kernel, and recorded its hardening settings.

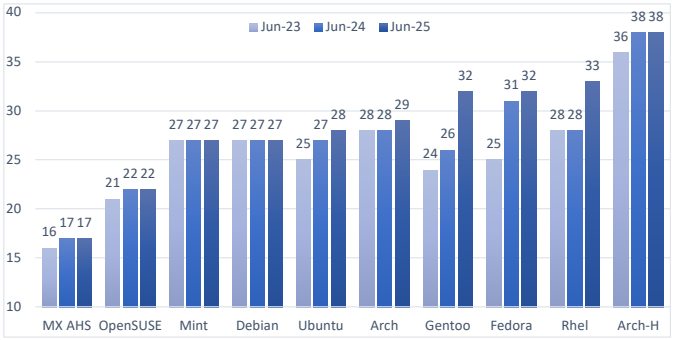


Fig. 7: Hardenings enabled by distributions from 2023 to 2025. Arch-H stands for Arch Hardened.

Using this dataset, we rank distributions by the number of hardening features enabled in June 2025. Fig. 7 shows the ranking and adds data from 2023 to 2025 to illustrate historical trends. We keep the same ranking and detail each leading distribution's latest hardening choices (see Tab. V). We observe that while most distributions enable a common baseline of hardening features, yet adoption of advanced protections varies markedly. Notably, `Arch Hardened` stands out for deploying the highest number of protections, including `DEBUG_VIRTUAL`, `GCC_PLUGIN_LATENT_ENTROPY`, `GCC_PLUGIN_STACKLEAK`, and `init_on_free` by default, reflecting its strong commitment to security. `RHEL`, `Fedora`, and `Gentoo` lead the standard distributions in security hardening. `Gentoo` demonstrates the most significant progress, having adopted the highest number of features over the past three years, while the latest `RHEL` release also marks a notable improvement. Furthermore, `Fedora` maintainer shared valuable insights into their kernel defense review process, shedding light on the gap between theoretical and practical hardening effectiveness. Meanwhile, `Arch`, `Ubuntu`, `Debian`, and `Mint` form a distinct cluster, exhibiting consistent hardening baselines.

Although the kernel configuration used by the standard `MX Linux` distribution is the same as that of the same version of `Debian`, the security capability of `MX Linux AHS` (a customized release for advanced hardware support) is not satisfactory. Its kernel configuration is different from that of `MX Linux`, with many critical protections, such as `FORTIFY_SOURCE`, `KASLR`, and `SLAB_FREELIST_HARDENED`, disabled by default. Disabling these powerful protections is undoubtedly imprudent, which

Hardenings	MX AHS	OpenSUSE	Mint	Debian	Ubuntu	Arch	Gentoo	Fedora	Rhel	Arch-H
CONFIG_DEBUG_LIST				✓			✓	✓	✓	✓
CONFIG_DEBUG_VIRTUAL										✓
CONFIG_DEBUG_WX⋆			✓	✓	✓	✓	✓	✓	✓	✓
CONFIG_FORTIFY_SOURCE		✓	✓	✓	✓	✓	✓	✓	✓	✓
CONFIG_GCC_PLUGIN_LATENT_ENTROPY										✓
CONFIG_GCC_PLUGIN_STACKLEAK										✓
CONFIG_HARDENED_USERCOPY	✓		✓	✓	✓	✓	✓	✓	✓	✓
CONFIG_INIT_ON_ALLOC_DEFAULT_ON	✓		✓	✓	✓	✓	✓	✓		✓
CONFIG_INIT_ON_FREE_DEFAULT_ON										✓
CONFIG_INIT_STACK_ALL_ZERO⋆			✓	✓	✓	✓	✓	✓	✓	✓
CONFIG_LIST_HARDENED						✓	✓	✓	✓	✓
CONFIG_PAGE_POISONING			✓	✓	✓	✓	✓	✓	✓	
CONFIG_PAGE_TABLE_CHECK		✓								
CONFIG_RANDOMIZE_BASE⋆		✓	✓	✓	✓	✓	✓	✓	✓	✓
CONFIG_RANDOM_KMALLOC_CACHES			✓		✓		✓	✓		✓
CONFIG_SCHED_STACK_END_CHECK		✓	✓	✓	✓	✓	✓	✓	✓	✓
CONFIG_SLAB_BUCKETS					✓	✓	✓	✓	✓	✓
CONFIG_SLAB_FREELIST_HARDENED		✓	✓	✓	✓	✓	✓	✓	✓	✓
CONFIG_X86_KERNEL_IBT⋆						✓	✓	✓	✓	✓
bpf_jit_harden	0	0	0	0	0	0	0	0	1	2
kptr_restrict	0	1	1	0	1	0	0	0	1	2
slab_nomerge	0	0	0	0	0	0	1	1	1	1

TABLE V: Hardenings deployment differences across major Linux distributions. The selected distributions are all the latest release versions as of June 2025, ranked according to their amount of kernel hardening features enabled. The detailed kernel versions are MX Linux 23.3 AHS (6.14.2-1-liquorix), openSUSE Leap 15.6 (6.4.0-150600.23.53-default), Linux Mint 22 cinnamon (6.8.0-62-generic), Debian 12 (6.1.0-37), Ubuntu 24.04 LTS (6.11.0-26-generic), Arch Linux (6.14.9-arch1-1), Gentoo (6.12.28-gentoo), Fedora Linux 42 Workstation Edition (6.14.11-300.fc42), Red Hat Enterprise Linux 10 (6.12.0-55.17.1.el10_0), Arch Linux Hardened (6.14.9-hardened1-1-hardened). Hardenings enabled by default in the upstream kernel is marked as ⋆ in "Hardenings" column.

would significantly downgrade the system security and make it an attractive target for adversaries. To analyze the reason behind the inappropriate configurations, we contacted the `MX AHS` administrator but have not received an explanation regarding the hardening deployment of `MX AHS`.

Similarly, `openSUSE` lacks robust kernel safeguards. While it is the only distribution to apply `PAGE_TABLE_CHECK`, it disables `HARDENED_USERCOPY`, thereby eliminating bounds checking for heap and stack memory access violations. We contacted the `openSUSE` team and received an acknowledgment but no follow-up. Besides, it also disables `init_on_alloc/free` and `PAGE_POISONING`, leaving the system vulnerable to sensitive information leakage.

Downstream: Then VS Now. As shown in Fig. 7, several distributions have increased their enabled hardening features. To trace the temporal trend, we analysed long-term-support releases from 2015 onward and found a steady rise across all distributions (details are provided in Figs.A.9 to A.11). To understand the driving force behind this trend, we correlate it with the hardening activity of the upstream kernel. A peak in the number of new hardenings accepted by the upstream kernel in 2016, which matches the growth peak of hardening deployment in downstream distributions (details in Fig. 8). This finding indicates that the deployment pace of downstream distributions is heavily dependent on the upstream kernel's defense integration, albeit with a noticeable delay.

Downstream VS Upstream. To further explore the gap in the kernel hardening deployment, we compared the hardening strategies between upstream and downstream kernels. By

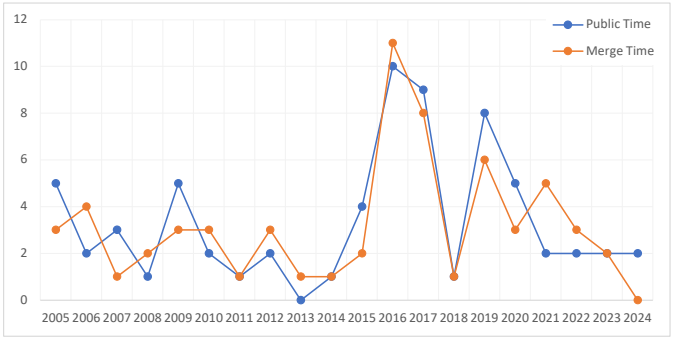


Fig. 8: Kernel-hardening patches merged into upstream, 2005–2024.

correlating each distribution with its corresponding upstream kernel version (v6.1 to v6.14) and default hardening settings, we identified several key differences. Our analysis revealed that the upstream kernels, particularly from v6.1 to v6.14, showed conservative updates in default hardening settings, with notable variation in the enablement of `X86_KERNEL_IBT`.

Additionally, upstream kernel's default configuration (defconfig) omits several fundamental hardening options, such as `HARDENED_USERCOPY` and `init_on_alloc`, which are essential for robust kernel safety. This observation implies that for a mature downstream distribution, merely adopting the upstream defconfig verbatim is insufficient to build a secure system. However, the introduction of a recommended hardening configuration in kernel v6.7 [133] now provides a valuable base-

line for distributions seeking to balance security with runtime performance. More details can be found in the Tab. V.

Further comparison uncovered that certain protections enabled by default in upstream kernels were absent in downstream distributions. For instance, `MX AHS`, which performed poorly in our security analysis, had `KASLR` disabled—protections enabled by default in upstream. Similarly, `DEBUG_WX` and `INIT_STACK_ALL_ZERO` were not enabled in `openSUSE` or `MX Linux`. Unfortunately, we received no explanation from `MX AHS` or `openSUSE` for these gaps.

Observation 8: Deployment Divergence. Our analysis revealed a significant stratification in the hardening deployment of top distributions. `Arch Hardened` leads with the broadest protection set; `RHEL`, `Fedora`, and `Gentoo` have recently closed much of the gap. By contrast, `MX AHS` turns off basics such as `KASLR`, and `openSUSE` uniquely disables `HARDENED_USERCOPY`. The divergence demonstrates that the popularity is not a reliable indicator of system security.

VI. Suggestions

Addressing Attack Vectors. Defenses should go beyond optimizing existing hardenings for performance and memory overhead. Security teams must prioritize developing new protections for unmitigated vectors and reinforcing defenses against frequently exploited ones. Although efforts like `SLAB_VIRTUAL` [125] and `SLAB_PER_SITE` [123] remain unmerged as of v6.16, they are promising to cover high-impact vectors.

Strengthening Hardening Mechanisms. To improve reliability, defenders should prioritize isolation- and permission-based protections over randomization where possible. When randomization is used, entropy must be maximized and random seeds securely stored. We recommend hardening fragile mechanisms—for example, replacing static magic constants with per-boot random values in critical structures like `uid` and `gid` (`DEBUG_CREDENTIALS`).

Securing Downstream Kernels. Distribution vendors and system administrators should adopt upstream default hardenings and validate their configurations rigorously. Additional hardenings should align with KSPP guidance to ensure correctness. Effective and general-purpose protections should be enabled by default and grouped into unified configuration options to reduce misconfiguration risk. Redundant or obsolete hardenings should be removed to avoid unnecessary overhead.

VII. Conclusion

In this study, we analyze and evaluate kernel hardening mechanisms for Linux kernel from the offensive-defensive perspective. By proposing a systematic exploitation decomposition framework, we assess common exploit methods, defense coverage, and limitations of existing hardenings. We also identify gaps between theoretical designs and practical deployments, offering recommendations to help developers avoid pitfalls in future kernel integration. We conclude that despite substantial efforts from the Linux community and security firms, existing hardening mechanisms remain insufficient and vulnerable to exploitation, underscoring the ongoing challenge of securing the Linux kernel.

Acknowledgment

We would like to thank the shepherd and anonymous reviewers for their insightful comments and suggestions that greatly improved the quality of this paper. We thank Xinyu Xing (Northwestern University) and Yueqi Chen (University of Colorado Boulder) for their insightful comments and feedback on this work. We also thank Tianyi Jing (Huazhong University of Science and Technology) and Quan Sun (University of Electronic Science and Technology of China) for their valuable assistance and support. This work was supported by the National Natural Science Foundation of China (No. 62102154, No.62502468), State Key Lab of Processors, Institute of Computing Technology, CAS under Grant No. CLQ202301 and Zhongguancun Laboratory.

References

[1] M. Jiang, J. Jiang, T. Wu, Z. Ma, X. Luo, and Y. Zhou, "Understanding vulnerability inducing commits of the linux kernel," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 7, Sep. 2024. [Online]. Available: https://doi.org/10.1145/3672452

[2] N. Galov, "111+ linux statistics and facts – linux rocks!" 2022, https://webtribunal.net/blog/linux-statistics.

[3] "Google queue hardening," https://security.googleblog.com/2019/05/queue-hardening-enhancements.html, 2019.

[4] M. Miller, "Trends challenges and strategic shifts in the software vulnerability mitigation landscape," https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf, 2019.

[5] "Kernel self protection project," 2015, https://kspp.github.io/.

[6] G. Thomas and A. Gaynor, "Writing linux kernel modules in safe rust," in *Linux Security Summit North America*, 2019.

[7] P. Zero, "Introducing the in-the-wild series," 2021, https://googleprojectzero.blogspot.com/2021/01/introducing-in-wild-series.html.

[8] Grsecurity, "Memory corruption defenses," 2021, https://grsecurity.net/featureset/memory_corruption.

[9] ——, "Gcc plugins," 2021, https://grsecurity.net/featureset/gcc_plugins.

[10] A. Popov, "Linux kernel defence map," 2020, https://a13xp0p0v.github.io/2018/04/28/Linux-Kernel-Defence-Map.html.

[11] A. M. Azab1, P. Ning, J. Shah1, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[12] "An overview of major linux distributions and freebsd," 2021, https://distrowatch.com/dwres.php?resource=major.

[13] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity." in *NDSS*, vol. 26, 2015, pp. 27–30.

[14] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: Cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 941–951.

[15] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 299–308.

[16] S. McCamant and G. Morrisett, "Evaluating sfi for a cisc architecture." in *USENIX Security Symposium*, vol. 10, 2006, pp. 1 267 336–1 267 351.

[17] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[18] X. Xu, M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin, "{CONFIRM}: Evaluating compatibility and relevance of control-flow integrity protections for modern software," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1805–1821.

[19] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu, "Finding cracks in shields: On the security of control flow integrity mechanisms," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1821–1835.

[20] S. Ahmed, Y. Xiao, K. Z. Snow, G. Tan, F. Monrose, and D. Yao, "Methodologies for quantifying (re-) randomization security and timing under jit-rop," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1803–1820.

[21] L. Maar, F. Draschbacher, L. Lamster, and S. Mangard, "Defects-in-Depth: Analyzing the integration of effective defenses against One-Day exploits in android kernels," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4517–4534. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/maar-defects

[22] J. Miller, M. Ghandat, K. Zeng, H. Chen, A. H. Benchikh, T. Bao, R. Wang, A. Doupé, and Y. Shoshitaishvili, "System register hijacking: Compromising kernel integrity by turning system registers against the system," in *34rd USENIX Security Symposium (USENIX Security 25)*, 2025.

[23] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[24] P. Zero, "Exploiting recursion in the linux kernel," 2016, https://googleprojectzero.blogspot.com/2016/06/exploiting-recursion-in-linux-kernel_20.html.

[25] DANGOKYO, "Analysis on CVE-2016-9793," 2017, https://dangokyo.me/2017/11/05/analysis-on-cve-2016-9793/.

[26] ww9210, "writeup for asa-2018-00053," 2019, https://github.com/ww9210/kernel4.20_bpf_LPE.

[27] M. Labs, "Linux kernel vulnerability can lead to privilege escalation: Analyzing CVE-2017-1000112," 2017, https://www.mcafee.com/blogs/other-blogs/mcafee-labs/linux-kernel-vulnerability-can-lead-to-privilege-escalation-analyzing-cve-2017-1000112/.

[28] W. Root, "CVE-2022-0185 - winning a $ 31337 bounty after pwning ubuntu and escaping google's kctf containers," 2022, https://www.willsroot.io/2022/01/cve-2022-0185.html.

[29] IMMUNITY, "Kernel memory disclosure & canvas part 2 - CVE-2017-18344 analysis & exploitation notes," 2018, https://www.immunityinc.com/downloads/Kernel-Memory-Disclosure-and-Canvas_Part_2.pdf.

[30] J. Park, "Linux kernel 4.8 (ubuntu 16.04) - leak sctp kernel pointer," 2018, https://www.exploit-db.com/exploits/45919.

[31] GRIMM, "New old bugs in the linux kernel," 2021, https://grimmcyber.com/new-old-bugs-in-the-linux-kernel/.

[32] H. Zhao, "Pwn2own 2017 analysis of linux kernel escalation," 2017, https://zhuanlan.zhihu.com/p/26674557.

[33] A. Konovalov, "Exploiting the linux kernel via packet sockets," 2017, https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[34] Y. Wang, "From zero to root: Building universal android rooting with a type confusion vulnerability," 2019, .

[35] chompie, "Put an io_uring on it - exploiting the linux kernel," 2022, https://chomp.ie/Blog+Posts/Put+an+io_uring+on+it+-+Exploiting+the+Linux+Kernel.

[36] K. Zeng, "[CVE-2022-1786] a journey to the dawn," 2022, https://blog.kylebot.net/2022/10/16/CVE-2022-1786/.

[37] HadarManor, "CVE-2020-16119," 2020, https://github.com/HadarManor/Public-Vulnerabilities/blob/master/CVE-2020-16119/CVE-2020-16119.md.

[38] ww9210, "Linux kernel exploits," 2018, https://github.com/ww9210/Linux_kernel_exploits.

[39] 0x3f97, "cve-2017-8890 root case analysis," 2018, https://0x3f97.github.io/exploit/2018/08/13/cve-2017-8890-root-case-analysis/.

[40] Mzi, "cve-2017-8890 vulnerability analysis and exploitation," 2018, https://www.freebuf.com/articles/terminal/160041.html.

[41] HardenedLinux, "Exploiting on CVE-2016-6787," 2017, https://hardenedlinux.github.io/system-security/2017/10/16/Exploiting-on-CVE-2016-6787.html.

[42] D. Shen, "The art of exploiting unconventional use-after-free bugs in android kernel," 2017, https://speakerdeck.com/retme7/the-art-of-exploiting-unconventional-use-after-free-bugs-in-android-kernel.

[43] L. Leong, "CVE-2021-20226: A reference-counting bug in the linux kernel io_uring subsystem," 2021, https://www.zerodayinitiative.com/blog/2021/4/22/cve-2021-20226-a-reference-counting-bug-in-the-linux-kernel-iouring-subsystem.

[44] J. Horn, "A cache invalidation bug in linux memory management," 2018, https://googleprojectzero.blogspot.com/2018/09/a-cache-invalidation-bug-in-linux.html.

[45] rebel, "linux af_packet race condition exploit for ubuntu 16.04 x86_64," 2016, https://github.com/LakshmiDesai/CVE-2016-8655/blob/master/CVE-2016-8655.c.

[46] A. Popov, "CVE-2019-18683: Exploiting a linux kernel vulnerability in the v4l2 subsystem," 2020, https://a13xp0p0v.github.io/2020/02/15/CVE-2019-18683.html.

[47] huahuaisadog, "CVE-2017-10661 exploitation," 2018, https://paper.seebug.org/596/.

[48] W. Xu and Y. Fu, "Own your android! yet another universal root," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[49] O. Nimron, "Ssd advisory – irda linux driver uaf," 2018, https://ssd-disclosure.com/ssd-advisory-irda-linux-driver-uaf/.

[50] A. Popov, "CVE-2017-2636: exploit the race condition in the n_hdlc linux kernel driver bypassing smep," 2017, https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html.

[51] Z. Lin, "Dirtycred exploitation on CVE-2022-2588," 2022, https://github.com/Markakd/CVE-2022-2588.

[52] C. Halbronn, "Settlers of netlink: Exploiting a limited uaf in nf_tables (CVE-2022-32250)," 2022, https://www.nccgroup.com/research-blog/settlers-of-netlink-exploiting-a-limited-uaf-in-nf_tables-cve-2022-32250/.

[53] T. kernel development community, "Basic operating systems terms and concepts," 2021, https://linux-kernel-labs.github.io/refs/heads/master/lectures/intro.html?#kernel-stack.

[54] K. Cook, "[0/3] mm/slab: Improved sanity checking," 2019, https://patchwork.kernel.org/project/linux-mm/cover/20190530045017.15252-1-keescook@chromium.org/.

[55] S. Jenkins, "Exploiting CVE-2022-42703 - bringing back the stack attack," 2022, https://googleprojectzero.blogspot.com/2022/12/exploiting-CVE-2022-42703-bringing-back-the-stack-attack.html.

[56] V. Dronov, "CVE-2017-7533 inotfiy linux kernel vulnerability," 2017, https://github.com/hardenedlinux/offensive_poc/tree/master/CVE-2017-7533.

[57] 0x3f97, "cve-2017-6074 briefly analyze," 2018, https://0x3f97.github.io/exploit/2018/08/16/cve-2017-6074-briefly-analyze/.

[58] A. Konovalov, "CVE-2017-6074," 2020, https://github.com/xairy/kernel-exploits/tree/master/CVE-2017-6074.

[59] J. Horn, "Issue 808: Linux: Uaf via double-fdput() in bpf(bpf_prog_load) error path," 2016, https://bugs.chromium.org/p/project-zero/issues/detail?id=808.

[60] A. Nguyen, "CVE-2021-22555: Turning \x00\x00 into 10000$," 2021, https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html.

[61] Bonfee, "CVE-2022-0995 exploit," 2022, https://github.com/Bonfee/CVE-2022-0995.

[62] C.-A. Lee, "nftables adventures: Bug hunting and n-day exploitation (CVE-2023-31248)," 2023, https://starlabs.sg/blog/2023/09-nftables-adventures-bug-hunting-and-n-day-exploitation/.

[63] lonialcon2 and conlonial, "Exploit detail about CVE-2024-1085," 2024, https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2024-1085_lts/docs/exploit.md.

[64] ——, "Exploit detail about CVE-2024-26581," 2024, https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2024-26581_lts_cos_mitigation/docs/exploit.md.

[65] NLQuy, "CVE-2024-26582," 2025, https://github.com/google/security-research/blob/5606368895ee56e64d099499709c96772ae25b8a/pocs/linux/kernelctf/CVE-2024-26582_mitigation/docs/exploit.md.

[66] V4bel-theori, "CVE-2025-21756," 2025, https://github.com/google/security-research/blob/f7dbb569a8275d4352fb1a2fe869f1afa79d4c28/pocs/linux/kernelctf/CVE-2025-21756_lts_cos/docs/exploit.md.

[67] W. N. James Fang, Di Shen, "Talk is cheap, show me the code," 2016, https://speakerdeck.com/retme7/talk-is-cheap-show-me-the-code.

[68] W. Shen, "Kernel pipe iov cve (CVE-2015-1805) exploit analysis," 2016, https://wenboshen.org/posts/2016-04-25-1805-cve.html.

[69] nightuhu, "CVE-2024-0582," 2025, https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2024-0582_mitigation/docs/exploit.md.

[70] kevinrich1337, "CVE-2024-0193," 2024, https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2024-0193_lts/docs/exploit.md.

[71] liona24, "CVE-2024-39503," 2025, https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2024-39503_lts_cos/docs/exploit.md.

[72] Awarau and D. Bouman, "CVE-2022-29582 an io_uring vulnerability," 2022, https://ruia-ruia.github.io/2022/08/05/CVE-2022-29582-io-uring/.

[73] Z. Lin, "Dirtycred exploitation on CVE-2021-4154," 2022, https://github.com/Markakd/CVE-2021-4154/blob/master/WRITEUP.md#dirtycred-exploitation.

[74] veritas501, "CVE-2021-22555 pipe version," 2022, https://github.com/veritas501/CVE-2021-22555-PipeVersion.

[75] V. Nikolenko, "CVE-2016-6187: Exploiting linux kernel heap off-by-one," 2016, https://duasynt.com/blog/cve-2016-6187-heap-off-by-one-exploit.

[76] M. Kellermann, "The dirty pipe vulnerability," 2022, https://dirtypipe.cm4all.com/.

[77] veritas501, "CVE-2022-0185 pipe version," 2022, https://github.com/veritas501/CVE-2022-0185-PipeVersion.

[78] ——, "CVE-2022-25636 - netfilter nf_dup_netdev heap oob write," 2022, https://github.com/veritas501/CVE-2022-25636-PipeVersion.

[79] ZDI, "CVE-2020-8835: Linux kernel privilege escalation via improper ebpf program verificatio," 2019, https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification.

[80] xmzyshypnc, "CVE-2020-27194," 2021, https://github.com/xmzyshypnc/CVE-2020-27194.

[81] N. Wu, "Dirty pagetable: A novel exploitation technique to rule linux kernel," 2023, https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html.

[82] notselwyn, "Flipping pages: An analysis of a new linux vulnerability in nf_tables and hardened exploitation techniques," 2024, https://pwning.tech/nftables/.

[83] qwerty theori, "CVE-2024-50264," 2025, https://github.com/google/security-research/blob/7cccb5605a0470d3447baae466c5cc452c2b16c0/pocs/linux/kernelctf/CVE-2024-50264_lts_cos/docs/exploit.md.

[84] d4em0n, "CVE-2025-40364," 2025, https://github.com/google/security-research/blob/724d93a311e336db1d48ae8dd41e9c370c81c5a5/pocs/linux/kernelctf/CVE-2025-40364_lts_cos/docs/exploit.md.

[85] pmgsbl, "Linux kernel vulnerabilty exploitation: CVE-2019-8956 and CVE-2019-9213," 2019, https://xz.aliyun.com/t/6570.

[86] houjingyi, "Analysis of CVE-2019-9213," 2019, https://cert.360.cn/report/detail?id=58e8387ec4c79693354d4797871536ea.

[87] 360CERT, "CVE-2017-16995: Ubuntu local privilege escalation," 2018, https://www.anquanke.com/post/id/101923.

[88] M. Paul, "CVE-2021-31440: An incorrect bounds calculation in the linux kernel ebpf verifier," 2021, https://www.zerodayinitiative.com/blog/2021/5/26/cve-2021-31440-an-incorrect-bounds-calculation-in-the-linux-kernel-ebpf-verifier.

[89] R. Li, "Stackrot (CVE-2023-3269): Linux kernel privilege escalation vulnerability," 2023, https://github.com/lrh2000/StackRot.

[90] mingi and M. Cho, "CVE-2024-0193," 2024, https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2024-0193_mitigation/docs/exploit.md.

[91] liona24, "CVE-2024-53164," 2025, https://github.com/google/security-research/blob/b0d0b003f2e1d6d977a99e00725b9415def818bc/pocs/linux/kernelctf/CVE-2024-53164_lts_cos_mitigation/docs/exploit.md.

[92] S. Heelan, T. Melham, and D. Kroening, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[93] Y. Chen and X. Xing, "Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[94] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[95] S. Han, "Lost control: Breaking hardware-assisted kernel control-flow integrity with page-oriented programming," in *BlackHat USA 2023*, 2023.

[96] st424204 and artmetla, "CVE-2024-36972," 2025, https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2024-36972_lts_cos/docs/exploit.md.

[97] liona24, "CVE-2025-21700," 2025, https://github.com/google/security-research/blob/430b8636156c1d81e48668a897df2c703b6a9c5a/pocs/linux/kernelctf/CVE-2025-21700_lts_cos_mitigation/docs/exploit.md.

[98] B.-J. B. Jheng and M. Ramdhan, "CVE-2023-4622_lts exploit tech overview," 2024, https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2023-4622_lts/docs/exploit.md.

[99] quanggle97, "Forget to reset pointer to null eventually leading to double free vulnerability in smbfs subsystem," 2024, https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2023-5345_lts_mitigation/exploit/lts-6.1.52/exploit.md.

[100] conlonial, "Exploit detail about CVE-2024-53125," 2025, https://github.com/google/security-research/blob/d9e9febbb748e2f258ca61445c9b13ef2abed1e0/pocs/linux/kernelctf/CVE-2024-53125_lts/docs/exploit.md.

[101] xorl, "CVE-2017-17053: Linux kernel ldt use after free," 2017, https://xorl.wordpress.com/2017/12/03/cve-2017-17053-linux-kernel-ldt-use-after-free/.

[102] chompie, "Kernel pwning with ebpf - a love story," 2021, https://chomp.ie/Blog+Posts/Kernel+Pwning+with+eBPF+-+a+Love+Story.

[103] javierprtd, "CVE-2020-27786 exploitation userfaultfd + patching file struct etc passwd," 2023, https://soez.github.io/posts/CVE-2020-27786-exploitation-userfaultfd-+-patching-file-struct-etc-passwd/.

[104] N. Asrir, "Linux kernel gsm multiplexing race condition local privilege escalation vulnerability," 2024, https://github.com/Nassim-Asrir/ZDI-24-020.

[105] marcograss, "Exploiting a linux kernel infoleak to bypass linux kaslr," 2016, https://marcograss.github.io/security/linux/2016/01/24/exploiting-infoleak-linux-kaslr-bypass.html.

[106] spender, "Exploit for CVE-2017-14954 from grsecurity," 2017, https://grsecurity.net/~spender/exploits/wait_for_kaslr_to_be_effective.c.

[107] TheOfficialFloW, "Linux: Stack-based information leak in a2mp (bleedingtooth)," 2020, https://github.com/google/security-research/security/advisories/GHSA-7mh3-gq28-gfrq.

[108] B. Spengler, "The infoleak that (mostly) wasn't," 2017, https://grsecurity.net/the_infoleak_that_mostly_wasnt.

[109] liona24, "CVE-2023-6817," 2024, https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2023-6817_mitigation/docs/exploit.md.

[110] conlonial, "Exploit detail about CVE-2024-26642," 2025, https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2024-26642_mitigation/docs/exploit.md.

[111] A. Prakash and H. Yin, "Defeating rop through denial of stack pivot," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, 2015.

[112] W. Wu, Y. Chen, X. Xing, and W. Zou, "Kepler: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019.

[113] N. Asrir, "Bypassing kaslr with startup_xen," 2024, https://github.com/Nassim-Asrir/ZDI-24-020?tab=readme-ov-file#exploitation-walkthrough.

[114] K. Lu, M.-T. Walter, D. Pfaff, and S. Nürnberger and Wenke Lee and Michael Backes, "Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying," in *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, 2017.

[115] H. Cho, J. Park, J. Kang, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn, "Exploiting uses of uninitialized stack variables in linux kernels to leak kernel pointers," in *14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[116] J. Edge, "Trying to get stackleak into the kernel," 2018, https://lwn.net/Articles/764325/.

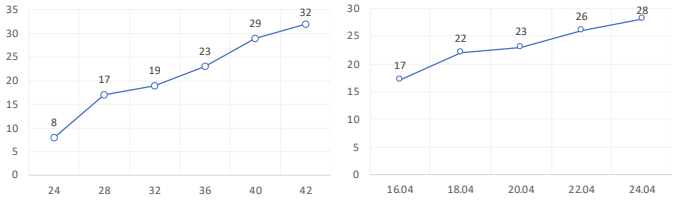
[117] D. Micay, "stackprotector: Increase the per-task stack canary's random range from 32 bits to 64 bits on 64-bit platforms," 2017, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=5ea30e4e58040cfd6434c2f33dc3ea76e2c15b05.

[118] K. Cook, "[patch v4 3/5] stack: Optionally randomize kernel stack offset each syscall," 2020, https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg2206197.html.

[119] E. Reshetova, "randomize kernel stack offset upon syscall," 2019, https://lwn.net/Articles/785484/.

[120] K. Zeng, Z. Lin, K. Lu, X. Xing, R. Wang, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Retspill: Igniting user-controlled data to burn away linux kernel protections," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23, New York, NY, USA, 2023, p. 3093–3107.
[121] A. Labs, "Grooming the ios kernel heap," 2020, https://azeria-labs.com/grooming-the-ios-kernel-heap/.
[122] M. Cho and W. Lee, "Utilizing cross-cpu allocation to exploit preempt-disabled linux kernel," 2024, https://www.hexacon.fr/slides/Cho_Lee-Utilizing_Cross-CPU_Allocation_to_Exploit_Preempt-Disabled_Linux_Kernel.pdf.
[123] K. Cook, "Per-call-site slab caches for heap-spraying protection," 2024, https://lwn.net/Articles/986174/.
[124] ——, "security things in linux v5.2," 2019, https://outflux.net/blog/archives/2019/07/17/security-things-in-linux-v5-2/.
[125] M. Rizzo, "Prevent cross-cache attacks in the slub allocator," 2023, https://lwn.net/Articles/944647/.
[126] N. Hussein, "Randomizing structure layout," 2017, https://lwn.net/Articles/722293/.
[127] J. Corbet, "Fun with null pointers, part 1," 2009, https://lwn.net/Articles/342330.
[128] D. Rosenberg, "Linux kernel 2.6.37 (redhat / ubuntu 10.04) - 'full-nelson.c' local privilege escalation," 2010, https://www.exploit-db.com/exploits/15704.
[129] J. Corbet, "A farewell to set_fs()?" 2017, https://lwn.net/Articles/722267.
[130] S. Tolvanen, "Kcfi support," 2022, https://lwn.net/Articles/907639/.
[131] joao AT-overdrivepizza.com, "Kernel fineibt support," 2022, https://lwn.net/Articles/891976/.
[132] B. Patel, "Intel releases new technology specifications to protect against rop attacks," *Retrieved March*, vol. 1, p. 2017, 2016.
[133] K. Cook, "hardening: Provide kconfig fragments for basic options," 2023, https://lore.kernel.org/linux-hardening/20230825050618.never.197-kees@kernel.org.

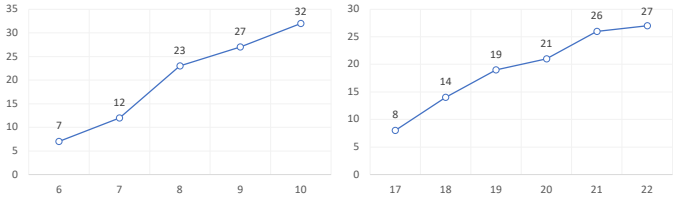
APPENDIX



(a) Hardenings enabled by Fedora from 24 to 42.

(b) Hardenings enabled by Ubuntu from 16.04 to 24.04.

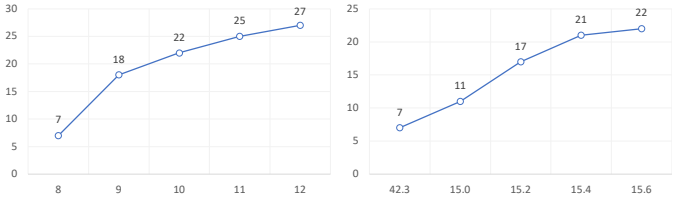
Fig. A.9: Comparison of hardening deployment trends in Fedora and Ubuntu.



(a) Hardenings enabled by RHEL from 6 to 10.

(b) Hardenings enabled by Linux Mint from 18 to 22.

Fig. A.10: Comparison of hardening deployment trends in RHEL and Linux Mint.



(a) Hardenings enabled by Debian from 8 to 12.

(b) Hardenings enabled by OpenSUSE from 42.3 to 15.6.

Fig. A.11: Comparison of hardening deployment trends in Debian and OpenSUSE.