The Multi-Stage Complexity of Exploiting User-Space Use-After-Free Vulnerabilities in Linux

I. Introduction: The Temporal Flaw in User-Space Memory Management

A. Defining Use-After-Free (UAF) in the Context of Linux User-Space

A Use-After-Free (UAF) vulnerability, classified under CWE-416, represents a fundamental temporal memory safety error. This flaw manifests when a program continues to reference memory via a dangling pointer after the underlying memory chunk has been explicitly returned to the heap allocator (freed).[1] While a simple UAF might only cause a program crash (a denial of service), the true danger emerges when the freed memory region is subsequently reallocated by the system and used by a new object.[3] In this scenario, the original dangling pointer now references the data structure of the new, unintended object, creating a critical condition known as type confusion.[3]

The exploitation process transforms this error into predictable, controlled memory corruption. The methodology hinges on three critical steps: identifying the freed memory location referenced by the dangling pointer, precisely manipulating the heap allocation patterns (known as heap grooming) to ensure a malicious object occupies that address, and finally, injecting payloads that corrupt metadata or overwrite critical pointers when the dangling pointer is dereferenced.[2] The system's propensity for performance optimization, particularly within modern Linux kernel and user-space applications, frequently introduces these temporal mistakes, which are easily leveraged by attackers to execute arbitrary code or bypass security mechanisms.[2]

B. Architectural Constraints: User-Space versus Kernel-Space

Linux employs a fundamental division between user-space and kernel-space to enforce security, stability, and performance boundaries.[4] User-space applications operate in a sandboxed environment known as user mode, where each process maintains its own independent, isolated virtual address space.[5] Applications must rely on the kernel to access resources, including requesting and releasing large blocks of memory via system calls like mmap and munmap.[6]

This separation dictates the complexity and consequence of exploitation. Exploiting a user-space UAF typically yields control solely over the compromised process, limiting the initial impact

unless the process operates with elevated privileges. In contrast, a kernel-space UAF, which targets kernel-specific memory allocators like SLUB, grants immediate control over system integrity and facilitates privilege escalation across all users.[7] Therefore, the central focus of a user-space UAF exploit campaign is the specific architecture and idiosyncrasies of the process's memory manager, which, in the Linux environment, is predominantly the GNU C Library's (glibc) implementation of dynamic allocation, known as ptmalloc.[9]

The comparative challenges of exploiting UAFs across these two domains are summarized below.

Comparative Challenges: User-Space vs. Kernel-Space UAF Exploitation

Feature/Challenge	**Linux User-Space UAF (GLIBC ptmalloc)**	**Linux Kernel-Space UAF (SLUB/SLAB)**
Memory Manager	Highly complex, multi-layered ptmalloc3 (Tcache, Fastbins, Arena).[9]	Simpler, dedicated object caches (SLUB/SLAB), designed for contiguous physical memory management.[7]
Information Leak Target	Libc (Unsorted Bin pointers) and Heap Base address offsets.[11]	Kernel object metadata (structs), page table entries, or side-channel leakage.[12]
Key Mitigations	ASLR/PIE, Safe Linking, DEP (NX), Hardened Allocator Checks.[13]	KASLR, SMEP/SMAP, Write-Protection/Reference Counting.[12]
Exploit Goal	Arbitrary R/W in process memory leading to control flow hijack (RCE).[1]	Arbitrary R/W in kernel memory leading to privilege escalation (Root).[8]

C. The Multi-Layered Challenge of Modern Exploitation

Successfully exploiting a contemporary user-space UAF requires overcoming a sophisticated, multi-layered defense architecture. This architecture includes system-level randomization measures such as Address Space Layout Randomization (ASLR) and Position Independent Executables (PIE), and specialized allocator-level integrity checks such as Safe Linking.[13]

The existence of highly optimized memory structures, such as the Thread-Local Cache (Tcache)

in glibc, demonstrates a critical dynamic in software development: performance prioritization often enhances exploitability. The Tcache, implemented for speed and multithreaded efficiency, utilizes singly-linked lists and avoids expensive synchronization locks.[9] This design choice means that if a UAF allows for the corruption of just a single pointer in the list, an attacker can control the entire subsequent allocation sequence (Tcache poisoning). The pursuit of faster memory access inherently creates predictable, exploitable weaknesses, transferring the vulnerability focus from robust heap structures to fast, minimally protected caches.

Consequently, modern exploitation campaigns necessitate a systematic approach: first, establishing memory knowledge (information leakage) to defeat randomization, and second, utilizing meticulous heap control techniques (grooming) to subvert the protective integrity checks built into the allocator.[11]

II. The Mechanics of GLIBC ptmalloc: The Exploiter's Blueprint

The heart of user-space UAF exploitation lies in manipulating ptmalloc3, the default heap allocator used by glibc. This allocator manages dynamic memory through a complex structure of arenas and bins, strategically categorizing freed memory chunks for rapid reuse.

A. Overview of Dynamic Allocation Management

When a user program requests a small allocation (malloc), ptmalloc services it from one of its internal bins. For very large requests, or when internal bins are exhausted, ptmalloc defers to the operating system using mmap to obtain large, independent memory regions, which are later returned using munmap.[6] However, the vast majority of memory corruption attacks focus on the internal bin structures that hold small-to-medium sized chunks, as these are reused frequently and governed by exploitable logic.

B. The Thread-Local Cache (tcache) — The Primary Vector

The Tcache, introduced in GLIBC 2.26, fundamentally changed heap exploitation strategy.[9] It serves as a per-thread cache for small chunks, accelerating allocation and deallocation by removing the need for global heap locks. This system effectively took over the role of Fastbins for high-speed memory transactions.

The structure of the Tcache is crucial for attackers: it is an array of singly-linked lists, indexed by chunk size. When a chunk is placed into the Tcache, it is treated as "in use" and, critically, it is never merged (coalesced) with adjacent free chunks.[10] The fixed limit on the number of chunks allowed in each Tcache bin (tcache_count) is a constraint that must be factored into the attacker’s heap grooming strategy.[17] Because the list is singly-linked, a UAF primitive allows the attacker to overwrite the pointer linking the current free chunk to the next one, providing direct

control over the allocation sequence when the chunk is later retrieved.[9]

The architectural evolution of glibc demonstrates that the shift to high-performance, lock-free structures like the Tcache fundamentally simplified the initial stages of exploitation. Prior to Tcache, exploiting Fastbins required handling arena locks and dealing with global synchronization. The Tcache's thread-local nature allows an attacker to reliably execute Tcache poisoning within a single thread, insulating the exploit from global heap contention and simplifying the required grooming phase.[9]

C. Auxiliary Bins: Fastbins and Unsorted Bin

While Tcache handles most common allocations, other bins remain relevant:

- **Fastbins:** Also singly-linked lists, these serve as a secondary cache, utilized if the Tcache is full or unavailable.[10]
- **Unsorted Bin:** This bin is a doubly-linked list used as a temporary holding area for newly freed chunks (typically large chunks, greater than the Tcache/Fastbin limits, such as 0x400 bytes) before they are sorted into small or large bins.[17] This temporary function makes the unsorted bin uniquely valuable for information leakage. When a chunk is inserted into this bin, its forward (fwd) and backward (bck) pointers are set to point to the list head, an address located within the static data section of the libc library.[11]

D. Concurrency and Stability Challenges

The reliability of a user-space heap exploit is deeply intertwined with the operating system's scheduling and memory management non-determinism. Modern multitasking kernels utilize context switching, introducing risk if a task is switched out of the CPU during a critical heap operation sequence (e.g., freeing the vulnerable object and immediately attempting to reoccupy the slot).[18] If an unexpected task allocates a chunk in that brief window, the exploit fails to reclaim the target slot, destroying the expected heap layout.[18]

Furthermore, the per-thread nature of the Tcache means that unwanted task migration between different CPUs can introduce inconsistency in the use of per-CPU freelists. If the vulnerable object is freed on CPU A, its slot returns to CPU A's freelist. If the exploit process migrates to CPU B to allocate the payload, it will draw memory from CPU B's freelist, leading to a failure to occupy the target slot.[18] To counteract this operating system non-determinism, specialized stabilization techniques are mandatory. This includes utilizing CPU Pinning, achieved via the sched_setaffinity system call, to attach the exploit threads to a specific CPU, ensuring the exploit and payload allocations interact with the same, consistent thread-local cache structures.[19] Reliability in heap exploitation requires actively managing and minimizing the non-determinism introduced by the multitasking kernel.

III. Stage 1: Heap Feng Shui and Reliability (Grooming)

The foundational requirement for a reliable UAF exploit is Heap Feng Shui, or grooming, which is the act of manipulating the heap's memory allocation patterns to place specific objects in predictable locations relative to the vulnerable chunk.[2]

A. The Goal of Heap Grooming

Heap grooming serves to bridge the temporal gap inherent in a UAF vulnerability. The goal is to ensure that immediately after the vulnerable object is freed, the allocator returns the same memory slot to an attacker-controlled replacement object.[2] This replacement must typically be the same size as the freed object to ensure it is drawn from the correct Tcache or Fastbin size bin.[3]

The successful execution of grooming transforms the temporal UAF flaw into a spatial corruption opportunity. By placing a replacement object of a different type at the original object's address, the attacker achieves type confusion. The dangling pointer, designed to access the metadata of the old, now-freed object, instead performs operations (reads or writes) on the data of the new, controlled object.[3]

B. Techniques for Controlling Memory Layout

Reliable control over the heap layout involves several coordinated techniques:

1. **Object Replacement Strategy:** Since the Tcache operates on a Last-In, First-Out (LIFO) basis, the fastest way to achieve slot reclamation is to free the vulnerable object and immediately call malloc for an object of the exact same size. The Tcache will return the most recently freed chunk.[19]
2. **Defragmentation:** To improve predictability, attackers often allocate and free a large number of objects in the same memory cache as the vulnerable object. This action fills up partially used memory pages (slabs), forcing the allocator to start new, fresh memory pages, thereby reducing memory fragmentation and increasing the chances that the desired memory allocation succeeds predictably.[19]
3. **Heap Spraying:** While historically used to place large amounts of executable shellcode, modern heap spraying is often used as a reliability technique. It involves allocating vast, contiguous blocks of memory and filling them with uniform data (such as specific gadget addresses or NOP-like sequences) to ensure that if a control flow hijack attempt lands near the intended target, it is still within controlled memory.[20]

The methodology demonstrates that the initial UAF primitive is not merely a trigger, but a mechanism refined through heap grooming to achieve persistence and spatial control. The ability to iterate and control these allocations negates the time-dependent nature of the UAF, resulting in a persistent, predictable vulnerability that can be leveraged at the attacker’s pace.[2]

IV. Stage 2: Information Leakage and ASLR Bypass

Contemporary exploitation relies on defeating randomization defenses before any critical control flow hijacking can occur. Address Space Layout Randomization (ASLR), complemented by PIE, randomizes the memory addresses of executable code (libraries, binaries) and data regions (stack, heap) upon process execution.[13] This necessitates that a UAF must first be leveraged as an information disclosure primitive to locate critical library addresses.

A. The Unsorted Bin Leak: Libc Base Disclosure

The standard and most reliable technique for obtaining the base address of the glibc library is through the manipulation and reading of the Unsorted Bin.[11]

The mechanism proceeds as follows:

1. **Placement:** The attacker must trigger a memory allocation and subsequent deallocation of a "big chunk"—typically larger than the maximum Tcache or Fastbin size (e.g., greater than 0x400 bytes)—to ensure the chunk is placed into the Unsorted Bin.[11]
2. **Metadata Write:** As the chunk is inserted into the doubly-linked Unsorted Bin, the ptmalloc allocator writes the address of the bin's list head into the chunk's backward pointer (bck).[11] Since the list head is a static structure within the memory image of libc, this pointer contains a randomized, yet structurally related, libc address.[11]
3. **Disclosure:** The UAF vulnerability is then triggered as a read operation on the dangling pointer, allowing the attacker to read the contents of the freed chunk's metadata fields, specifically capturing the bck pointer.[11]
4. **Calculation:** Since the offset of the list head within the libc binary is constant and known, the attacker calculates the absolute, randomized **Libc Base Address** by subtracting the static offset from the leaked pointer value, effectively neutralizing ASLR for the entire library.[11]

B. The Dual Role of the UAF Primitive

The requirement to perform this leakage step demonstrates a fundamental weakness in probabilistic defenses like ASLR; they merely force a two-stage attack model (Leak then Exploit) instead of mitigating the underlying memory corruption issue.[13] The UAF primitive is highly versatile, serving simultaneously as a reliability primitive during grooming and as an information primitive during the leakage phase. This is possible because the dangling pointer's temporal window [2] can be exploited, first, to manipulate the heap structure (writing control data) and, second, to read the contents of the memory after the allocator has populated it with sensitive internal metadata (reading libc addresses).[2]

Once the libc base address is known, the attacker has solved the memory location problem for all code and static data within the standard library, enabling the final phases of the attack.

V. Stage 3: Transforming UAF into an Arbitrary

Read/Write Primitive

The core objective of the middle exploitation stages is to elevate the UAF from an uncontrolled memory corruption event into a deliberate, targeted Write-What-Where primitive: the ability to write an arbitrary value (What) to an arbitrary memory location (Where).[1] This primitive is the critical bottleneck, as it grants the necessary power to hijack program execution.

A. Tcache Poisoning Fundamentals

Tcache Poisoning is the canonical technique for achieving arbitrary write using a UAF against ptmalloc.[9] It directly exploits the single-linked list nature of the Tcache.

The exploitation sequence is as follows:

1. **Freeing the Vulnerable Chunk:** The attacker triggers the UAF by freeing a chunk of a specific size, placing it at the head of the corresponding Tcache bin list.
2. **Corrupting the Pointer:** The UAF primitive (a controlled write via the dangling pointer) is executed to overwrite the chunk's **next pointer**. The attacker calculates and inserts the target memory address (e.g., the address of a critical function hook, minus the chunk header size) into the next field.[9]
3. **First Allocation:** The attacker triggers a malloc() call for a chunk of the same size. The allocator returns the original chunk.
4. **Second Allocation (The Poison):** A second, identical malloc() call is triggered. The allocator follows the newly corrupted next pointer and returns a chunk whose user-data section starts at the attacker-chosen target address.
5. **Arbitrary Write:** The attacker's subsequent data writing operation on this second allocated chunk is effectively an arbitrary memory write to the pre-calculated, critical target location.[9]

B. Leveraging Limited Primitives

The success of Tcache poisoning is not strictly dependent on a full arbitrary overwrite primitive. Analyses of real-world exploits show that even severely restricted memory access, such as a vulnerability that only allows decrementing a pointer field by one byte, can be escalated.[23] By repeatedly triggering the UAF bug and replacing the freed object, an attacker can iteratively decrement a target pointer field—potentially one that points to free list metadata—until the pointer's value is sufficiently modified to create a large memory overlap.[23] This overlap corrupts the internal free list structure, allowing the attacker to force a subsequent allocation into a known memory region, ultimately yielding a controlled arbitrary write. This demonstrates that allocator predictability and iteration can erode the safety margin of even the weakest memory corruption flaws.

The primary stages of the UAF exploit chain are formalized in the following table:

UAF Exploit Stages and Associated GLIBC Mechanism

Exploit Stage	Objective	Targeted GLIBC Mechanism	UAF Primitive Used
Reliability (Grooming)	Ensure target memory chunk is reallocated predictably.[19]	Tcache LIFO behavior; Allocator Size Matching.	Allocation/Free cycles (Heap Grooming/Spraying).[2]
Information Leak	Defeat ASLR by leaking libc/heap base address.[11]	Unsorted Bin pointers (bck pointer).[11]	Use (Read) of the dangling pointer post-free.[11]
Arbitrary Write Primitive	Gain control over a future allocation address.	Tcache next pointer (Tcache Poisoning).[9]	Controlled Write operation on the freed chunk.[1]
Code Execution	Hijack control flow to execute shellcode or ROP chain.[1]	Critical function hooks (e.g., __free_hook).[11]	Arbitrary Write to overwrite the hook address.[11]

VI. Stage 4: Advanced Mitigation Bypass (Safe Linking)

The evolution of glibc defenses has directly responded to the reliability of Tcache Poisoning. Starting in GLIBC 2.32, the Safe Linking feature was introduced to protect the integrity of the fastbin and tcache linked lists.[14]

A. The Safe Linking Countermeasure

Safe Linking implements pointer mangling. Instead of storing the raw memory address of the next free chunk in the next field, ptmalloc stores an encoded, XOR-masked value. This value is derived using a formula (Mangled_Pointer = Pointer \text{ XOR } (Key \text{ Right Shift } 12)), where the Key is typically the heap base address.[14] This mitigation aims to prevent two key attack vectors: first, using a UAF read to leak raw heap addresses (as the pointer is masked), and second, achieving a Tcache poisoning by simply overwriting the next pointer with a known target address (as the attacker would need to know the dynamic heap key to calculate the correctly mangled value).

The continuous refinement of glibc mitigations, moving from simple double-free checks to

robust pointer integrity defenses like Safe Linking, validates the high severity and enduring exploitability of UAF vulnerabilities.[14]

B. Bypassing Safe Linking with UAF Primitives

Safe Linking dramatically increases the complexity required for successful exploitation. Attackers must now either discover a reliable way to leak the heap base key, or engineer a complex sequence of heap operations to circumvent the pointer check logic entirely.

For older versions of glibc (e.g., 2.31), the allocator employed a simpler defense: it stored a heap base address pointer (tcache_key) in the freed chunk metadata to detect double-free attempts.[11] A UAF write primitive could easily bypass this by overwriting the stored tcache_key with a junk value after the first free operation, thus defeating the check when the chunk was freed a second time.[11]

Against modern versions implementing Safe Linking, exploit complexity shifts from simple pointer overwrites to multi-stage logical attacks. Advanced UAF techniques, such as those leveraging the Tcache stashing mechanism in a "House of Rust" style attack, demonstrate that it is possible to achieve pointer corruption by exploiting subtle flaws in the *sequencing* and *logic* of the allocation process itself, even when the key is unknown.[14] These complex maneuvers prove that strong cryptographic defenses on heap pointers only raise the knowledge and complexity barrier for attackers; they do not eliminate the underlying exploitability inherent in UAF flaws.

VII. Stage 5: Gaining Code Execution (Control Flow Hijacking)

With the arbitrary write primitive firmly established (often via Tcache poisoning), the final stage focuses on hijacking the program's control flow to execute malicious code.

A. Targeting Critical Function Hooks

The standard strategy for achieving execution hinges on overwriting static function pointers located within the known data segment of the libc library.[1] This methodology evolved as a direct response to compiler and OS mitigations (like Data Execution Prevention, DEP) that made injecting and running raw shellcode difficult.

The primary targets are execution hooks that the library guarantees to dereference:

- **__free_hook:** This is the most frequently targeted function pointer. It resides in the libc data section and is executed immediately before the free() function returns control to the calling program.[11]
- **Other Hooks:** Hooks such as __malloc_hook are also viable targets, executed before malloc() is fully performed.

By using the arbitrary write primitive, the attacker overwrites the chosen hook's address with the calculated address of a highly useful library function, most commonly system().[11]

B. The system("/bin/sh") Payload Execution

After overwriting the function hook, the attacker must prepare the shell command and trigger the corrupted execution flow:

1. **Payload Placement:** The attacker allocates a new chunk of memory and writes the desired shell command string, such as "/bin/sh\x00", into the chunk's user data section.
2. **Triggering the Hook:** The attacker then calls free() on this chunk.
3. **Control Flow Hijack:** The execution of free() triggers the overwritten __free_hook, which now calls the system() function. The free() routine automatically passes the address of the chunk's user data (which holds "/bin/sh") as the argument to the hook, resulting in a call equivalent to system("/bin/sh\x00"), thereby dropping a shell with the privileges of the compromised process.[11]

C. Leveraging Non-Control Data Attacks (NCDA)

In environments where Control Flow Integrity (CFI) is rigorously enforced, preventing the direct modification of function pointers, attackers may pivot to Non-Control Data Attacks (NCDA).[15] This strategy involves using the arbitrary write primitive to corrupt sensitive, non-control data structures, such as security flags, capability sets, or user IDs, to achieve privilege escalation without ever hijacking the instruction pointer.[15] While NCDA is often discussed in the context of kernel exploitation, the principle is fully applicable to user-space sensitive data structures, particularly when the user-space process runs as a high-privilege service.

The successful exploitation of a user-space UAF, providing reliable code execution, is often the critical first step in a larger attack chain. Recent vulnerabilities, such as CVE-2023-6246 in glibc, demonstrate that a local, unprivileged user-space UAF can be chained to gain full root privilege escalation, illustrating the strategic importance of neutralizing the initial user-space compromise.[16]

VIII. Conclusion and Mitigations

The analysis confirms that exploiting a user-space UAF vulnerability in modern Linux is a highly complex, multi-stage process that targets the intricate operational logic and performance optimizations of the GLIBC ptmalloc allocator. The exploitation chain mandates rigorous heap grooming, information leakage to bypass ASLR, complex techniques like Tcache Poisoning to gain the arbitrary write primitive, and sophisticated bypass mechanisms (such as those against Safe Linking) to circumvent allocator integrity checks.

The persistence of UAFs as a high-severity threat is highlighted by recent flaws, including those

in core libraries like glibc, which continue to be exploited for local privilege escalation to root.[16] These vulnerabilities are often triggered by intricate concurrency issues and race conditions, affirming that complex codebases are still prone to temporal memory management errors.[1]

To counteract this sophisticated threat model, defensive strategies must focus on architectural and structural hardening:

1. **Hardened Memory Allocators:** The most potent defense against UAF is the structural elimination of dangling pointers. Solutions employing One-Time Allocation (OTA), where memory locations are never reused, such as FFmalloc, prevent attackers from reclaiming freed objects entirely, thereby neutralizing the UAF vector regardless of the application bug.[26]
2. **Compiler Defenses:** The rigorous application of memory safety tools, such as Address Sanitizer (ASAN) and Memory Sanitizer (MSAN), during development is crucial for catching temporal errors before deployment.[26]
3. **System and Library Maintenance:** Maintaining contemporary versions of the Linux kernel and critical user-space libraries (like glibc) is essential, as these updates often contain the latest allocator integrity checks and defenses, such as Safe Linking.[2]
4. **Code Auditing:** Specialized static and dynamic analysis and rigorous code review are required to identify the complex, cross-entry control flows that often trigger concurrent UAFs in large codebases.[24]

Works cited

1. CWE-416: Use After Free, accessed October 29, 2025, https://cwe.mitre.org/data/definitions/416.html
2. What Is a Use After Free (UAF) Vulnerability in a Linux Server?, accessed October 29, 2025, https://linuxsecurity.com/features/what-is-a-use-after-free-uaf-vulnerability
3. Using freed memory - OWASP Foundation, accessed October 29, 2025, https://owasp.org/www-community/vulnerabilities/Using_freed_memory
4. Userspace vs Kernel Space: A Comprehensive Guide | by Neel Shah - Medium, accessed October 29, 2025, https://medium.com/@shahneel2409/userspace-vs-kernel-space-a-comprehensive-guide-8f9b96cd6426
5. Userspace vs Kernelspace: Understanding the Divide - Oracle Blogs, accessed October 29, 2025, https://blogs.oracle.com/linux/post/userspace-vs-kernelspace-understanding-the-divide
6. How do userspace programs pass memory back to the kernel after free()? - Stack Overflow, accessed October 29, 2025, https://stackoverflow.com/questions/36323389/how-do-userspace-programs-pass-memory-back-to-the-kernel-after-free
7. Security vulnerability: Linux kernel memory corruption vulnerabilities exploitable

through the SLUBStick technique | SUSE | Support Center, accessed October 29, 2025, https://support.scc.suse.com/s/kb/Security-vulnerability-Linux-kernel-memory-corruption-vulnerabilities-exploitable-through-the-SLUBStick-technique
8. SETTLERS OF NETLINK: Exploiting a limited UAF in nf_tables (CVE-2022-32250), accessed October 29, 2025, https://www.nccgroup.com/research-blog/settlers-of-netlink-exploiting-a-limited-uaf-in-nf_tables-cve-2022-32250/
9. TCACHE Heap Exploitation - SecQuest, accessed October 29, 2025, https://www.secquest.co.uk/white-papers/tcache-heap-exploitation
10. Heap Exploitation Part 2: Understanding the Glibc Heap ..., accessed October 29, 2025, https://azeria-labs.com/heap-exploitation-part-2-glibc-heap-free-bins/
11. Heap exploitation, glibc internals and nifty tricks. - Quarkslab's blog, accessed October 29, 2025, https://blog.quarkslab.com/heap-exploitation-glibc-internals-and-nifty-tricks.html
12. Reliable and Stable Kernel Exploits via Defense-Amplified TLB Side-Channel Leaks - USENIX, accessed October 29, 2025, https://www.usenix.org/system/files/usenixsecurity25-maar-kernel.pdf
13. Exploiting Linux and PaX ASLR's weaknesses on 32- and 64-bit systems - Black Hat, accessed October 29, 2025, https://www.blackhat.com/docs/asia-16/materials/asia-16-Marco-Gisbert-Exploiting-Linux-And-PaX-ASLRS-Weaknesses-On-32-And-64-Bit-Systems.pdf
14. Bypassing glibc Safe-Linking: CSAW 2021 Quals (word_games) - Margin Research, accessed October 29, 2025, https://margin.re/2021/09/bypassing-glibc-safe-linking/
15. Beyond Control: Exploring Novel File System Objects for Data-Only Attacks on Linux Systems - arXiv, accessed October 29, 2025, https://arxiv.org/html/2401.17618v2
16. CVE-2023-6246 Root Access Vulnerability in glibc - Open Source Security Foundation, accessed October 29, 2025, https://openssf.org/blog/2024/02/05/cve-2023-6246-root-access-vulnerability-in-glibc/
17. MallocInternals - glibc wiki, accessed October 29, 2025, https://sourceware.org/glibc/wiki/MallocInternals
18. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability - USENIX, accessed October 29, 2025, https://www.usenix.org/system/files/sec22fall_zeng.pdf
19. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability - Haehyun Cho, accessed October 29, 2025, https://haehyun.github.io/papers/playing-for-keaps-22-sec.pdf
20. Heap spraying - Wikipedia, accessed October 29, 2025, https://en.wikipedia.org/wiki/Heap_spraying
21. A Lightweight Method for Defending Against UAF Vulnerabilities - arXiv, accessed October 29, 2025, https://arxiv.org/html/2405.20697v1

22. Software defense: mitigating common exploitation techniques - Microsoft, accessed October 29, 2025, https://www.microsoft.com/en-us/msrc/blog/2013/12/software-defense-mitigating-common-exploitation-techniques
23. Exploiting a textbook use-after-free security vulnerability in Chrome ..., accessed October 29, 2025, https://github.blog/security/vulnerability-research/exploiting-a-textbook-use-after-free-security-vulnerability-in-chrome/
24. Statically Discover Cross-Entry Use-After-Free Vulnerabilities in the Linux Kernel, accessed October 29, 2025, https://www.ndss-symposium.org/wp-content/uploads/2025-559-paper.pdf
25. Bypassing GLIBC 2.32's Safe-Linking Without Leaks into Code Execution: The House of Rust - c4e, accessed October 29, 2025, https://c4ebt.github.io/2021/01/22/House-of-Rust.html
26. Preventing Use-After-Free Attacks with Fast Forward Allocation - USENIX, accessed October 29, 2025, https://www.usenix.org/system/files/sec21summer_wickman.pdf