

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Discovering, characterizing and exploiting controllable-copy objects for kernel data-only attacks with CopyKat

Jakob Koschel, Andrea Mambretti, Alessandro Sorniotti, Pietro Moretto, and Claudio Migliorelli, *IBM Research Europe;* Andrea Di Dio and Cristiano Giuffrida, *Vrije Universiteit Amsterdam;* Anil Kurmus, *IBM Research Europe*

https://www.usenix.org/conference/usenixsecurity26/presentation/koschel

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

Discovering, characterizing and exploiting controllable-copy objects for kernel data-only attacks with CopyKat

Jakob Koschel[†*] Andrea Mambretti[†*] Alessandro Sorniotti[†*] Pietro Moretto[†]
Claudio Migliorelli[†] Andrea Di Dio[‡] Cristiano Giuffrida[‡] Anil Kurmus[†]

[†]IBM Research Europe – Zurich [‡]Vrije Universiteit Amsterdam

[*]These authors contributed equally

Abstract

Modern kernel hardening mechanisms considerably reduce the viability of control-flow hijacking attacks, prompting attackers to shift their focus to data-only attacks. These attacks depend on corrupting non-control data to amplify limited memory-corruption primitives into more powerful capabilities such as arbitrary writes. Prior work has identified small sets of such objects in specific circumstances. However, no existing approach discovers and characterizes them at scale. In this paper, we introduce CopyKat, a fully automated pipeline for identifying and characterizing Controllable-Copy Objects (CCOs): heap-allocated kernel structures whose fields can be corrupted to redirect a store or memory-transfer operation involving attacker-controlled data. We propose a multi-stage methodology with progressive filtering of objects. Through lightweight taint-analysis applied to a fuzzing campaign, we uncover candidate CCOs. Then, we confirm such taint by using a precise taint engine, thereby enhancing our precision. Finally, we perform the corruption of the CCO using a concolic execution engine to verify the controllability of the object. We also collect the necessary path and value constraints for exploitation. We find 122 verified CCOs and build 8 end-to-end exploits involving different exploitation patterns, showcasing the correctness of our characterization and the flexibility of the discovered objects.

1 Introduction

The development of a kernel exploit typically begins with the identification of a bug causing the violation of some property (temporal or spatial memory safety, for instance). The community has devised effective solutions to find bugs in the kernel (e.g., syzkaller [13] and extensions [34]). After finding an initial bug, a target on which to exercise the violation must be identified for privilege escalation. Historically, the target has been control-flow metadata, whose compromise could lead to arbitrary code execution. However, with the design and adoption of kernel-level control flow integrity (CFI) [12, 36], the community started looking at data-only attacks.

This class of attacks targets non-control kernel objects whose fields, when corrupted, can amplify a limited primitive into a more powerful capability, and is becoming a central technique in modern exploitation. In practice, attackers often reuse a small set of well-known amplifier objects (e.g., `msg_msg`) to steer writes, but this creates a race between attackers and kernel developers: once a class of objects is repeatedly abused, kernel developers respond by segregating or otherwise hardening them (e.g., cache segregation of `msg_msg`), which pushes attackers toward more complex cross-cache layouts and techniques, and in turn motivates further mitigations. Consequently, automated discovery and characterization of such amplifier objects is crucial. It offers early visibility into emerging attacker strategies, enables maintainers to preemptively close exploitation avenues (both in-cache and cross-cache), and addresses a key subproblem in Automated Exploit Generation (AEG).

Although prior work [2, 5, 22, 23, 37, 40, 43] has explored the discovery of kernel objects useful for exploitation, these efforts generally focus on specific patterns and surface only a small number of targets. Moreover, once an object is identified, existing systems rarely investigate what is actually required to create an exploit with it in practice. However, identifying a promising object is only the beginning. As detailed in Section 3, an attacker must determine the precise syscall paths that allocate and trigger it, as well as the format and provenance of the attacker-controlled data reaching the sink. Additionally, the attacker must reason about the corruption window between pointer initialization and use, slab-cache constraints affecting reachability, collateral corruption effects, and semantic constraints on the target memory region. Without characterizing these aspects, a significant gap remains between object discovery and object usability.

In this paper, we address this gap with CopyKat, a generic approach for automatically discovering and characterizing data-only amplifier objects at scale. Rather than encoding a known exploitation pattern and searching for matching objects, our approach starts from a more basic attacker capability. Specifically, we assume the ability to corrupt a pointer field

inside a heap object. From there, it follows data flows from corrupted heap-resident pointers to generic write sinks whose source data is attacker-controlled. This allows us to identify *controllable-copy objects* (CCOs): heap-allocated kernel structures that can amplify a basic pointer corruption into a write at an attacker-chosen location, even when the resulting primitive is constrained rather than a fully unconstrained arbitrary write. This formulation directly addresses the narrow discovery scope of prior pattern-specific approaches.

A representative example of such pattern-specific approaches is BridgeRouter [40]: it defines a two-stage capability-upgrading workflow in which a *bridge* object first transforms an uncontrolled overwrite into a controlled one, and a *router* object subsequently converts this into an arbitrary memory write. Its analysis and fuzzing pipeline are then designed to discover kernel objects that can realize and be composed along the very specific bridge-to-router flow. While effective for instantiating this workflow, the approach inherently focuses the search on objects that fit these predefined roles, ultimately discovering only a handful of controlled router objects matching this pattern. In contrast, our pattern-agnostic analysis identifies and verifies 122 CCOs across the kernel. Crucially, we also automate the characterization of these objects—that is, their requirements and constraints—at scale, and demonstrate its practical value through 8 end-to-end privilege-escalation exploits.

This paper makes the following **contributions**:

- Characterization of CCO data-only amplifier objects. These objects allow attackers to enhance the primitive given by a base memory-safety vulnerability into a write at an attacker-chosen location, with constrained or fully attacker-controlled contents. The memory surrounding the target location of the write may also need to satisfy constraints.
- An automated tool, CopyKat, to find and extract characteristics of a CCO.
- Identification of 122 CCOs, with 8 exploits demonstrating their practical use. In particular, we demonstrate that such objects do not need to provide arbitrary write capability to attackers, and instead demonstrate that constrained write capabilities are sufficient to achieve privilege escalation.

Attacker model and goal Our attacker model is in line with existing work. We assume the attacker has a limited memory corruption primitive on the kernel heap, such as an out-of-bounds (OOB) write or use-after-free (UAF) write due to a vulnerability in the Linux kernel. These primitives are limited in their capability in the sense that the attacker can only write to memory in an adjacent area (in case of OOB), or at the same address once another object is allocated (UAF). The number of bytes written, and the written values, may also be constrained. The attacker's goal is to amplify this capability by leveraging CCOs. The attacker's ultimate goal is to obtain privilege escalation with this amplified primitive. While this may be achieved via an arbitrary controlled write, also known as a write-what-where capability, a weaker primitive may also be sufficient as we demonstrate.

Generally, we do not assume the attacker has access to additional vulnerabilities. However, depending on the underlying type of memory corruption and chosen exploitation strategy, exploitation using CCOs may also require defeating kernel address space layout randomization (KASLR)—trivial nowadays with well-known side-channel attack frameworks [3]—or even having an arbitrary read primitive. Obtaining such an information-leak capability is generally orthogonal to CCOs, and can be achieved with other existing approaches [5].

2 Background and Related Work

Classical kernel exploitation. Classical kernel exploitation traditionally corrupts control data (e.g., function pointers, return addresses) to hijack execution, first via code injection [32] and later through code-reuse techniques like ROP/JOP [33]. Mitigations such as NX/XD, stack canaries [10], SMAP/SMEP, and KASLR [8] pushed attackers to reuse existing kernel code [38, 42]. In a typical Control-Flow Hijacking (CFH) attack, a memory-corruption bug overwrites a control-flow object (e.g., a function pointer). An indirect call then redirects execution into attacker-chosen kernel gadgets. Although such attacks often require defeating KASLR, several derandomization techniques exist [3]. Consequently, CFH now relies entirely on reusing legitimate kernel instruction sequences, bypassing defenses without code injection or stack modification. As a result, kernel hardening has increasingly focused on Control-Flow Integrity (CFI) [1, 35], restricting indirect control transfers. In Linux, kCFI [9] provides the main software mechanism, while hardware-assisted approaches such as Intel IBT [44] and FineIBT [14, 45] combine coarse-grained checks with fine-grained type validation. Backward-edge CFI relies mainly on shadow stacks in hardware, including Intel CET [28] and ARM GCS [29].

Data-only kernel exploitation. With the broader deployment of kernel CFI variants [26, 27], attackers have turned to data-only attacks, which bypass CFI by corrupting non-control data. These attacks target security-critical structures to either augment an attacker's primitive (e.g., upgrading a constrained write to an arbitrary one) or directly achieve privilege escalation (e.g., via corrupted credentials and file objects) [17]. DirtyCred swaps unprivileged and privileged credential- or file-related objects via heap reuse, avoiding direct field corruption [18, 23]. DirtyFree generalizes privilege escalation via *arbitrary-free* primitives to replace security-critical objects such as `struct cred` [22]. Recent work also expands the set

```
1  struct tcp_md5sig_key {
2    struct hlist_node {
3      struct hlist_node *next, **pprev;
4    } node;
5    ...
6    u8 family;
7    int l3index;
8    u8 key[80];
9    ...
10 }
```

1: Definition of the CCO used as running example.

```
1  static struct tcp_md5sig_key *
2  tcp_md5_do_lookup_exact(const struct sock *sk,
3     const union tcp_md5_addr *addr, int family,
4     u8 prefixlen, int l3index, u8 flags) {
5     struct tcp_md5sig_key *key;
6     ...
7     hlist_for_each_entry_rcu(key, ...) {
8     if (key->family != family)
9       continue;
10    ...
11    if (key->l3index != l3index)
12      continue;
13    ...
14    }
15    return NULL;
16 }
```

2: The `tcp_md5_do_lookup_exact` function validates the supplied `key`.

```
1  int tcp_md5_do_add(struct sock *sk,
2       const union tcp_md5_addr *addr, int family,
3       u8 prefixlen, int l3index, u8 flags,
4  OP3: const u8 *newkey, // User input
5       u8 newkeylen,gfp_t gfp) {
6       struct tcp_md5sig_key *key;
7       ...
8       key = tcp_md5_do_lookup_exact(sk, addr, family,
9            prefixlen, l3index, flags);
10      if (key) {
11         // Primitive Trigger
12 OP5:     memcpy(key->key, newkey, newkeylen);
13         ...
14         return 0;
15      }
16      ...
17      // CCO Allocation
18 OP1: key = sock_kmalloc(sk, sizeof(*key), ...
19
20      // CCO Initialization
21      memcpy(key->key, newkey, newkeylen);
22      ...
23      // Pointer Allocation
24 OP2: hlist_add_head_rcu(&key->node, &md5sig->head);
25
26      return 0;
27 }
```

3: The `tcp_md5_do_add` function performs the memory operation on the pointer stored inside the CCO object.

Listing 1: The heap-allocated object defined in Listing 1.1 controls the destination pointer of the memory operation at line 12 in Listing 1.3, whose source buffer contains userspace-supplied data. By corrupting the `*next` field (line 3 in Listing 1.1), the attacker influences the destination of the memory-transfer. In doing so the attacker obtains an arbitrary write primitive. That capability is constrained by the checks shown in Listing 1.2: in particular, line 8 and line 11 require the corrupted pointer to refer to a region in which the `family` field contains the value `0x2` (`AF_INET`), and the `l3index` field contains the value 0.

of kernel data-only targets (e.g., file subsystem objects) and studies their exploitability under modern mitigations [43].

Automating kernel exploitation. Prior work automates different parts of the exploitation pipeline. SLAKE, PSPRAY, SLUBStick, Cross-X and PCPLost analyze object and allocator properties and automate kernel heap manipulation to facilitate exploitation [6, 21, 24, 25, 31]. FUZE and KOOBE assist exploit generation by extracting and composing capabilities for UAF and OOB-write vulnerabilities [4, 39]. KEPLER focuses on *control-flow hijacking* by automatically synthesizing kernel ROP chains from a CFH primitive [38]. SCAVY systematically discovers privilege-escalation targets based on access-control-related effects (via differential testing) across general kernel data structures [2]. A recent SoK [19] contains additional details on the related work in automated kernel exploit generation. Finally, BridgeRouter [40] introduces a framework for transforming limited out-of-bounds write primitives into arbitrary memory writes by chaining two classes of kernel objects–*bridge* objects, which migrates an overflow in one cache into another cache, and *router* objects, which redirect memory-copy operations to attacker-chosen locations. Its analysis combines static identification of candidate objects with guided-fuzzing-based exploration to instantiate this two-stage bridge-to-router workflow in practice. BridgeRouter is organized around this fixed capability-upgrading pattern: both its object discovery and chaining logic are explicitly guided by the requirement that selected objects realize the predefined bridge-to-router data flow. Consequently, the search space is restricted, yielding a relatively small set of usable chains.

In contrast, our approach begins from a minimal attacker capability—pointer corruption—and systematically recovers all data flows that propagate this corruption to write sinks with attacker-controlled data. This requires a more involved analysis pipeline, combining lightweight taint-guided fuzzing, precise dynamic taint verification, and concolic execution to both validate exploitability and extract semantic constraints. While more complex, this machinery allows us to relax the assumptions on how amplification must occur and to capture a broader spectrum of objects, including those that do not fit a bridge/router decomposition or that yield constrained, rather than fully arbitrary, write primitives.

3 Data-only attacks via CCOs

This paper concentrates on data-only attacks targeting data pointers used in write operations (which we refer to as a CCO pointer), as they allow an attacker to amplify a limited write primitive given by a vulnerability. This amplification leads to overwriting a *target* address, typically a field within an object that allows further escalation, for example by overwriting access-control-relevant data. The corruption of the CCO pointer must also take into account any *constraints* that prevent the write to the target from occuring.

One class of pointers that fall into this category comprises those used as the destination of a memory-store operation. If the attacker can establish that the source data of such store is attacker-controlled, corrupting such a pointer yields a write of attacker-controlled data to an attacker-chosen location in the kernel. Such an amplified write primitive often implies a complete compromise of the kernel, depending on constraints on the CCO write which we further detail below.

Definition 3.1. A *controllable-copy object* (CCO) is a kernel-heap object `obj` that satisfies the following properties: (i) The object `obj` contains a pointer `ptr`; (ii) The kernel uses the pointer `ptr` as the destination of a store operation or of a memory-transfer function; (iii) The source data for the store is attacker-controlled.

In Listing 1.1, we provide a sample CCO discovered by our analysis. We will use it across the paper as running example. The CCO is of type `struct tcp_md5sig_key` and is used to store a shared secret for TCP MD5 authentication for BGP packets as per RFC2385. This kernel struct is used in a list of `tcp_md5sig_key` objects as indicated by the `hlist_node` pointers `*next` and `*pprev`. The `*next` pointer is the CCO pointer that is used as the target of a store, which an attacker would need to corrupt. The source buffer of that store is attacker-controlled input, copied from userspace via a `copy_from_user` call in the `tcp_md5_do_add` parent function.

Having identified a CCO, an attacker can exploit it as follows (for reference, see Figure 1):

OP1 **CCO Allocation:** the CCO `obj` is allocated.
Running example: the `setsockopt` system call allocates an object of type `struct tcp_md5sig_key` via the `tcp_md5_do_add` function (see line 18 of Listing 1.3).

OP2 **Pointer Allocation:** the pointer `ptr` stored within `obj` is initialized to refer to its legitimate target (`area`).
Running example: the newly allocated CCO is linked into the hash list, thereby initializing its internal list pointer (i.e., `node.next`) at line 24 of Listing 1.3

OP3 **Fetching userspace input:** attacker-controlled `input` is supplied to the kernel.
Running example: attacker-controlled input, copied from userspace via a `copy_from_user` call in the `tcp_md5_do_add` parent function, is used to initialize the fields of the CCO (see line 21 of Listing 1.3).

OP4 **Pointer Corruption:** the attacker overwrites the value of `ptr`, redirecting it to a malicious area `counterfeit`.
Running example: the attacker uses a vulnerability to overwrite the `next` pointer, forcing it to point to the area `counterfeit`.

OP5 **Primitive Trigger:** the attacker executes the store or memory-transfer operation involving `ptr`.
Running example: by calling the `setsockopt` system call again, the `tcp_md5_do_lookup_exact` function returns the corrupted pointer to the malicious `counterfeit` and the store using this pointer occurs at line 12 of Listing 1.3.

At the conclusion of this sequence, the attacker can cause `input` to be stored at the attacker-controlled `counterfeit` area.

Several factors might impact the success of building this sequence of operations. These include gaining control over the source data used in the memory operation and pairing a limited primitive with a suitable CCO while considering the slab-cache constraints. Additionally, the attacker must map exploitation-operations to concrete system calls, identify a viable corruption window between pointer initialization and use, handle collateral corruption, and select a target memory area that satisfies semantic constraints while still achieving an adversarial goal. In the following subsections we describe these in detail.

Discovering and characterizing CCOs automatically at scale is complex. It requires addressing several technical challenges:

CH1 *Identifying objects whose fields are used when determining the destination of memory-transfer operations involving attacker-controlled data.*

CH2 *Designing and implementing a lightweight taint analysis approach on top of a fuzzer.*

CH3 *Validating all relevant data-flows using a complete taint engine.*

CH4 *Verifying whether the object can be successfully used after corruption.*

3.1 CCO Analysis

3.1.1 Control over source data

In OP3, the attacker supplies `input` to the kernel; the kernel generally invokes `copy_from_user` to read data from userspace. Accordingly, the attacker must determine which system call and which argument can be used to pass `input`

to the kernel. Furthermore, while some system calls will accept user-provided buffer with arbitrary content, others expect the input to follow a specific format and will reject non-conforming data. The attacker therefore has to establish the correct format and verify that it is compatible with the values required to achieve successful exploitation.

3.1.2 Slab caches

The attacker's initial foothold is expected to be a limited primitive, for example an OOB or UAF write, or a double-free (DF) affecting objects in a SLAB cache. Such primitives are typically found in generic caches, whereas the objects whose fields the attacker wishes to corrupt in order to finalize the exploit (for instance process credentials) often reside in dedicated caches. Consequently, the attacker must locate CCOs amenable to corruption that are allocatable from generic caches. Exploitation is therefore facilitated by identifying a wide range of objects covering as many SLAB caches as possible, thereby allowing the attacker to pair any available limited primitive with a suitable object. As demonstrated by publicly available KernelCTF proof-of-concept exploits [15], initial adversarial capabilities (for example a memory-corruption primitive uncovered by Syzkaller) are more likely to involve objects allocated from generic caches, rather than dedicated ones, simplifying exploitation. In our running example, the CCO is allocated in a generic cache, i.e., `kmalloc-192`, and can be easily paired with a vulnerable object residing in the same cache, as described in Section 5.3.

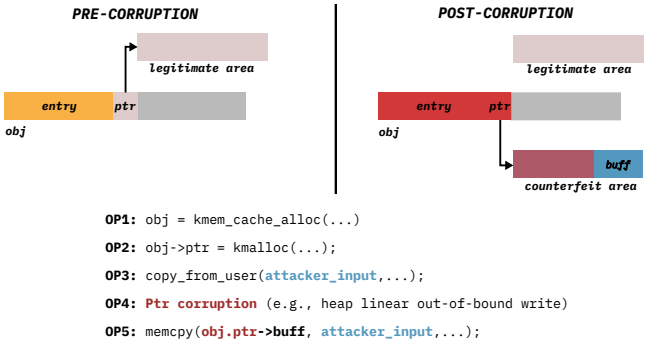


Figure 1: Layout of the CCO before and after corruption. The CCO is allocated in **OP1**. **OP2** is the last `store` setting the pointer field `ptr` that the attacker corrupts. **OP3** sets the `source` of the memory operation with attacker-controlled data. At **OP4** `ptr` is corrupted, redirecting the destination of the memory operation to an area selected by the attacker (`counterfeit`). **OP5** leads to the arbitrary write.

3.1.3 System call identification

Each of the operations required for exploitation are performed indirectly by the attacker, as they are mediated via system calls; consequently, the attacker must map each of the five conceptual exploitation operations onto one or more concrete system calls, providing the appropriate arguments so as to reach the requisite code. Furthermore, supplementary system calls may be required to establish the necessary kernel state before the calls that constitute the attack can be executed. In our running example, for each exploitation operation, our automated analysis reports the location in the kernel, along with a stack trace identifying the responsible userspace operation.

3.1.4 Handling collateral corruption

The most favorable situation for the attacker is one in which the initial primitive permits the precise corruption of pointer `ptr` without any extraneous corruption. In other cases the attacker may corrupt `ptr` together with other fields of object `obj`. Such collateral corruption can impair exploitability in two ways: (i) The corruption diverts the control flow so that the store or memory-transfer operation is never reached; (ii) The corruption causes the kernel to crash after the store or memory-transfer operation. Although the second outcome is not necessarily fatal (e.g., because the kernel is configured to continue execution after a panic) the first is unacceptable, as it prevents the arbitrary-write primitive from being exercised. Consequently the attacker must be aware of the necessary conditions to avoid, at a minimum, the first case and, ideally, both. Assuming that collateral corruption cannot be avoided and has been suitably minimized, these conditions amount to determining the range of permissible values for each variable affected by such collateral corruption. In the running example, the attacker overwrites the `*next` pointer, which is the first field of CCO (as shown in the structure definition in Listing 1.1). This constitutes a favorable case: when exploiting a linear OOB write, there is no risk of corrupting preceding fields.

3.1.5 Target memory area constraints

The objective of the attacker is to replace the intended destination address of the store or memory-transfer function to a legitimate area, which resides within pointer `ptr`, with an attacker-chosen target address to a counterfeit area. The latter is selected by the attacker to achieve an adversarial goal, for example overwriting fields of process credentials in order to elevate privileges. While the attacker is free to choose any target, the choice may incur additional exploitation constraints: for example, after the corruption, the surrounding code may still load other fields of the same structure (such as control flags or function pointers) before or after performing the targeted write. If these fields no longer contain valid or expected values, the control flow may abort or diverge before the attacker-controlled write occurs, or may fail to complete the operation safely. Consequently, the attacker must identify all semantic constraints that the code imposes on the target

memory region and select an alternate location that both satisfies all such semantic constraints and still yields meaningful adversarial capabilities once the memory-transfer occurs. It should be noted that the ideal CCO is one that imposes no additional constraints; such objects do exist and are identified automatically in our work. Nevertheless, we show that objects remain useful for exploitation even when they are subject to a nonzero set of constraints.

In the running example, the `*next` pointer is part of a hash-list node; the counterfeit area pointed to after corruption must therefore have a memory layout compatible with that of the CCO itself and must respect the constraints imposed by the `tcp_md5_do_lookup_exact` function (see Listing 1.2) in order to reach the `memcpy`.

3.2 CCO Exploitation

Corruption time window In this attack, the arbitrary-write primitive is obtained by corrupting a pointer stored on the heap that the kernel will later use as the destination of a store or memory-transfer operation. The attacker has a time window during which the pointer may be corrupted; this window opens immediately after the pointer has been written into the heap object and closes just before the pointer is read from that object to serve as the destination argument (possibly after intermediate stores). Corruption outside this interval will not produce the desired effect and may instead cause the kernel to panic. Identifying this time window is therefore essential to the attack. As will be shown, the most favorable situation for the attacker is one in which the store and the load occur in two distinct system calls: the attacker can invoke the first system call, trigger the corruption, and subsequently invoke the second. When both events occur within a single system call, the attacker must find ways of slowing or stalling the kernel at appropriate points so that the pointer can be corrupted.

In the running example, we invoke the `setsockopt` system call twice. During the first invocation, we exercise the path leading to the OP1 object allocation (line 18 in Listing 1.3). This also performs OP2 which initializes the pointer `*next` through the addition to the `hlist` line 24 in Listing 1.3. The second invocation instead reaches OP5, which resides in the key-update path at line 12 in Listing 1.3. To use this CCO correctly in an exploit, the attacker must be able to perform OP4 within the time window between the two `setsockopt` invocations.

CCO choice The attacker must select a suitable CCO for successful exploitation. The choice is influenced by several considerations: (i) The object must be corruptible using the initial adversarial capability; this may also entail co-location in particular caches. (ii) The attacker must be able to trigger the system calls that cause the object's allocation, the copying of attacker-controlled data and the ensuing store or memory-transfer operation

The starting points for the selection process are an accurate description of the adversarial capabilities conferred by the initial vulnerability, together with a database of potential CCOs.

Target choice To finalise the exploit, the attacker must ensure that the store or memory-transfer operation overwrites data, causing a security side-effect which is beneficial for the attacker, for example, zeroing out the `uid` field of the credential of one of the attacker's processes. The discovery of the address of such memory objects is a challenge which we consider out of scope since a variety of methods for locating them already exist. Even when the address is known, the attacker must still ensure that the size and the value of the data to be stored are compatible with the requirements for a successful exploit. For example, if only a single byte needs to be overwritten but the operation writes a minimum of four bytes, the attacker must ensure that the surplus bytes do not corrupt adjacent data or otherwise interfere with the intended behavior.

4 Automatically finding CCOs

Our design of CopyKat, shown in Figure 2, takes an instrumented kernel as input and returns a set of CCOs, accompanied by the necessary metadata to use it for exploitation. This metadata includes reproducers (i.e., C programs that contain all the operations shown in Figure 1 for a particular CCO), kernel and userspace backtraces identifying the salient operations (allocation, pointer storage, kernel data-transfer and target memory-transfer operation) as well as symbolic constraints that describe the requirements that need to be satisfied by an attacker to successfully use such object in an attack.

As with any automated analysis that must scale to the level of the Linux kernel, we are confronted with the fundamental trade-off between analysis time and accuracy. We resolve this tension by first applying a fast, lightweight taint analysis on top of syzkaller. This produces a large number of candidate CCO instances, although it over-approximates and introduces false positives. We then refine and filter the results using precise PANDA-based taint verification and S2E concolic execution to validate the information flows, extract the symbolic constraints and discard non-viable cases, thereby keeping the fuzzing stage fast while regaining precision later. The system is organized as a cascade of analyses. Early stages are fast but imprecise, while later stages are slower but more accurate. Each layer filters false positives from the previous stage.

4.1 Design

The first stage must be efficient at scale and should result in as large a set of objects as possible. In CopyKat, we address and overcome the challenges introduced in Section 3. We

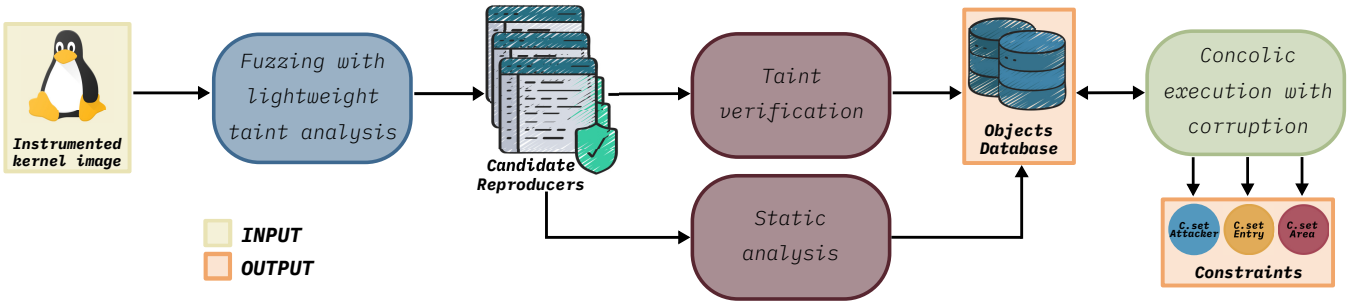


Figure 2: Illustration of our CopyKat pipeline for discovering CCOs. Starting from an instrumented kernel, CopyKat produces reproducers and three sets of constraints by cascading three analyses: a lightweight taint-enhanced fuzzing stage to surface candidates, a precise taint-verification stage to remove false positives, and a concolic-execution stage that simulates pointer corruption to confirm controllability and extract the path constraints required to reach the memory operation.

address CH1 by combining `llvm` analyses and passes with syzkaller fuzzing: fuzzing enables us to automatically exercise substantial portions of the kernel, while the `llvm` passes allow us to flag the pertinent events. We consider three main events: (i) a pointer is stored in a heap object; (ii) attacker-controlled data is stored in the kernel; (iii) a memory-transfer operation moves attacker-controlled data from event ii to a target location determined using the data stored by event i. All three events can be tracked using a dynamic data flow or taint engine. However, fuzzing is notoriously sensitive to performance degradation: a naïve integration of a taint engine into syzkaller is unlikely to scale adequately.

We address CH2 by leveraging KASAN shadow memory and instrumentation infrastructure. Specifically, we repurpose KASAN into a lightweight taint propagation and verification engine. This version does not include memory-safety functionality. The extended KASAN layer tags memory regions with labels denoting relevant properties such as "stores attacker-controlled data" or "a pointer to this region is stored in a heap object". Moreover, we construct the requisite instrumentation to propagate these labels in a manner analogous to KASAN's shadow memory management. With this extension, syzkaller generates reports whenever attacker-controlled data is transferred to a location whose address is influenced by a field stored in a heap object. Under appropriate assumptions, this behavior may be exploitable to obtain an arbitrary-write or similarly powerful primitive. We describe this in detail in Section 4.2.

We demonstrate empirically that this lightweight tainting is both efficient and effective, but we recognize that it may yield false positives due to overtaint: for example, a pointer held in a register is aliased into a heap field and later used directly; overtaint assumes a load from the alias on the heap and erroneously flags the heap field as participating in the determination of the destination pointer.

For CH3, we select the PANDA analysis framework for this purpose, and devise a suite of analyses to validate each report and to extract further additional data such as offsets, cache structures, the responsible system calls and the precise origin of the taint, etc. We describe this in detail in Section 4.3.

While our analysis with PANDA can verify the correctness of the taint, it does not guarantee that the discovered objects are actually corruptible and usable. To provide such guarantee we need to address CH4 and find how that can be done by the attacker. As part of this challenge, we need to answer the following questions: (i) Should any value within the CCO be restored during corruption if overwritten? (ii) Does the attacker-chosen target memory area need to preserve any specific values to keep the kernel's control flow on the path to the write? (iii) What are the characteristics of the resulting write primitive the attacker has obtained? To answer these questions, we employ concolic execution on the candidate reproducers via the S2E framework, described in Section 4.4 to validate and fully characterize each object.

Results from each analysis are added to a database. An extract of the database entry for our running example is provided in Listing 2 up to the PANDA analysis, and in Listing 3 for the concolic analysis. Entries are uniquely identified by the ID used by Syzkaller to flag the report.

4.2 Fuzzing with Lightweight tainting

To discover an initial set of CCOs, we extend syzkaller and run it against an instrumented kernel. The instrumentation–implemented via `llvm` passes–creates and propagates labels for memory regions. We define three memory labels to capture three properties: if the region belongs to the kernel heap, if it stores data supplied by an attacker, and if a pointer to the region has been stored on the heap (recall that being stored on the heap is a pre-requisite for being overwritten by the attacker). Labels are created by instrumenting allocation sites for the first property, by instrumenting calls to functions obtaining data from userspace such as `copy_from_user` for the second property, and by instrumenting store or memory-transfer operations for the third property. The labels are stored in unused bits of KASAN

shadow-memory bytes and are propagated by further instrumenting stores and memory-transfer operations. In particular, for the label that records that a pointer to a region was stored on the heap, we attach the label to the shadow memory of the pointee rather than to the pointer itself; thus the property is associated with every alias of any pointer to the region. Moreover, the label is extended to the shadow memory of the whole allocation (from `kasan_mem_to_shadow(ptr)` to `kasan_mem_to_shadow(ptr + sz)` for an object located at address `ptr` of size `sz`). This approach ensures that a pointer to any byte within a heap object can be identified as having been stored on the heap, provided that at least one valid pointer to that object has been stored on the heap. Creation and propagation of the taint are packaged as `kcov` coverage signal and fed back to syzkaller to steer the fuzzing. KASAN guarantees that the labels are automatically cleared whenever an object is deallocated.

The instrumentation also incorporates logic to detect any store or memory-transfer operation whose destination pointer has been stored on the heap, and whose source data is attacker-controlled. Syzkaller is further extended to selectively trigger a panic each time such a store or memory-transfer is executed, one per reproducer. The panic then causes Syzkaller to commence the reproducer generation and minimization phases.

The output of this stage is a collection of reproducers that identify a store or memory-transfer operation with the required characteristics of source data and destination pointer provenance. The analysis creates a database entry for each reproducer, initializing its identifier to match the one generated by Syzkaller. At this stage the entry records the responsible store or memory-transfer operation, together with its location in the binary and in the source code. We also apply static analysis using Joern [41] to determine the destination operand of the operation, its expression (pointer dereference, array access, ...) and its type, including a complete list of its fields.

4.3 Taint verification

This lightweight taint engine is similar to KASAN and consequently incurs a performance degradation that is acceptable for a fuzzing kernel. Unfortunately, it is also imprecise, as it generates both false positives and false negatives. False negatives may arise, for instance, when taint is propagated via operations other than stores or memory-transfers (e.g., any arithmetic or logical instructions). We are not concerned about this, as our approach already yields a sufficient number of candidates. False positives may arise in a number of situations. For example, on the source-data side, data that has been imported with `copy_from_user` (and therefore correctly marked as attacker-controlled) may be subsequently zeroed out or otherwise overwritten; likewise, on the destination-pointer side, a pointer stored on the heap may be retrieved via a local alias, preventing the attacker from controlling the destination address by corrupting the heap copy.

To address these challenges, we must verify that the requisite data flows exist for both the source data and the destination pointer. To this end, we execute each reproducer under PANDA (Platform for Architecture-Neutral Dynamic Analysis) [11]. PANDA is an open-source, QEMU-based whole-system dynamic-analysis platform offering a flexible plugin architecture and record-and-replay capability. We have constructed a pipeline of analyses that progressively uncover further information about the reproducer, or, where appropriate, dismiss it as a false positive. Initially we validate the source data involved in the store or memory-transfer identified by the syzkaller reproducer by creating a taint label on all buffers populated via `copy_from_user` (or equivalent functions), and checking whether any such label is present at the moment of the transfer. If a label is detected, we output kernel and userspace backtraces for the responsible `copy_from_user`, the system-call arguments and allocation information for the source-data object, including both kernel and userspace backtraces for the allocation site, and the cache identifier.

We then analyze the destination pointer of the flagged memory-transfer operation to determine whether it was loaded from a heap object. If that is the case, we identify the most recent store that set it. PANDA's deterministic record-and-replay capability makes this possible: knowing the destination pointer in advance enables us to taint every memory location targeted by a store whose value matches the future destination pointer of the flagged transfer.

One challenge with this approach is that the stored pointer and the destination pointer may not match exactly–for example, the heap stores the base address of an object whilst the destination pointer includes an offset. To handle this, we avoid setting PANDA callbacks on the store that sets the heap pointer. Instead, we hook into the taint-propagation instrumentation described earlier, which marks the whole heap allocation with the "pointer stored on the heap" label. This propagation proceeds in 8-byte steps, in accordance with KASAN's granule size (one shadow byte per eight bytes of kernel virtual memory). We can then check whether the destination pointer of the flagged memory-transfer operation lies within eight bytes of any taint-propagation callback, thereby ensuring accurate detection even when offsets are involved. We can therefore assign a taint label to every store of a pointer that may be associated with the destination pointer of the flagged memory-transfer operation, and subsequently determine whether a label is present at the instant the transfer is performed. If a label is found, we report both userspace and kernel backtraces for the store that influences the destination pointer, together with allocation details for the object that holds the stored pointer (userspace and kernel backtraces for the allocation, cache identifier, field name and in-struct offset). This store determines the start of the window during which the attacker may corrupt the stored value, thereby controlling the destination of the memory-transfer operation.

Each stage of the analysis adds further information to the database. The recording stage registers the values of the destination and source operands for the flagged memory-transfer. Because replay is deterministic, knowledge of these addresses permits later analyses to refer to those exact values a priori. Referring to the entry shown in Listing 2, the analysis fully characterizes five events: (i) the flagged store or memory-transfer (recorded as `sink` in the database). In the running example, the analysis confirms that it is the `memcpy` call at line 12 of Listing 1.3 which updates the `key` field of the CCO. (ii) the pointer assignment that sets the heap field which will be used to determine the destination pointer (`ptr_ass` in the DB). In the running example, the internal list pointer `next` within the `node` field is initialized at line 24 of Listing 1.3. This step in the analysis validates the reproducer as a true positive by observing a data flow between the pointer assignment and the memory-transfer. (iii) allocation of the target CCO, in which the destination pointer is stored (`target_obj_alloc` in the DB). In the running example, the CCO is of type `struct tcp_md5sig_key` and is allocated in the `kmalloc-192` cache at line 18 of Listing 1.3. (iv) allocation of the legitimate pointee for the destination pointer (`dst_ptr_alloc` in the DB). In the running example, since the CCO destination pointer is part of a hashlist node, its pointee is of the same type as the CCO and is also allocated from the same cache at line 18 of Listing 1.3. (v) userspace-kernel data-transfer that supplies the content of the source buffer (`taint_src` in the DB). In the running example, the `copy_from_user` call in the `tcp_md5_do_add` parent function supplies data from userspace to set the value of the `newkey` field. Each event carries an internal identifier that uniquely distinguishes operations within the instrumented kernel, together with a counter that records the occurrence of that particular operation. It also contains a pointer to a backtrace identifier for the operation (denoted `k_bt` for kernel backtraces and `u_bt` for userspace backtraces) which references a backtrace stored in a separate database for deduplication. The system call that triggers the operation is characterized, including the values of all relevant registers. Allocation events additionally record the name of the SLAB cache, the size of the allocated object and its base address. The base address aids in determining the runtime offset of the heap pointer within the object, which is required by the S2E analysis. Finally, each operation reports the instruction counter at the time of its execution: this is useful to estimate the size of the time window between pointer storage and memory-transfer.

4.4 Corruption verification and Constraint extraction

As shown in Figure 2, the PANDA analysis brings forward a set of reproducers for which the necessary data flows on source data and destination pointer could be verified. It also provides a database containing information related to the discovered primitive (i.e., which memory operation does it use, which `copy_from_user` influences its source buffer and which `store` operation influences its destination pointer). This database facilitates the initialization of the concolic analysis, which is implemented as a plugin in S2E [7], a framework for kernel-wide concolic execution.

This analysis has two main objectives: first, to verify whether corrupting the heap pointer set by the store flagged by the PANDA analysis really influences the value of the destination pointer. Second, to determine any further effects (if any) that this corruption may have on the control and data flows. We achieve these objectives by overwriting the aforementioned heap pointer, causing it to point to a region of memory under our control (denoted as `counterfeit` in Figure 1) which we initialize via a shallow copy of the original object and subsequently symbolise. The symbolised shallow copy forces concolic execution to follow as closely as possible the original path leading to the memory operation.

Based on the location of two operations–the target memory-transfer operation flagged in the reproducer and the store that sets its destination pointer–we distinguish two categories of reproducers: those in which the operations occur in separate system calls, and those in which both occur within the same system call. This distinction determines the attacker's corruption strategy, and consequently shapes how our analysis must model the corruption of the heap pointer. In the case where the operations occur in distinct system calls, the attacker is free to decide when to trigger the corruption. Consequently, we corrupt the pointer that controls the destination of the memory operation immediately before invoking the system call that triggers it. In the case where the operations occur within the same system call, the attacker must trigger the corruption in the interval between the execution of the store that sets the heap pointer and the load that fetches it as the destination pointer for the memory-transfer operation. We overwrite the heap pointer immediately after the store that sets its value. This allows us to obtain an exhaustive set of constraints that an attacker would need to satisfy in order to achieve successful exploitation within the interval between the store and the memory transfer operation. In practice, fewer constraints might apply if the corruption can be delayed.

The output of this analysis is a report (see Listing 3), which extends our database, presenting the number of constraints generated for each of the memory areas highlighted in Figure 1 as `entry`, `counterfeit`, and `attacker buffer`. For each of these areas we also compute the total number of bytes that these constraints influence in the overall symbolised buffer, as shown at line 15, line 11, and line 13 of Listing 3 respectively. The report includes the information described above for both: (i) the moment the reproducer reaches the memory-transfer (line 7 of Listing 3), and (ii) the moment the reproducer returns to userspace after the system call has terminated (line 17 of Listing 3). We record instances in which neither point is reached because of a panic (line 6 of Listing 3), as well as

instances where the memory-transfer is reached but the kernel thread panics before returning to userspace. The latter case reveals whether the primitive may be reused repeatedly within the same kernel thread, or whether several kernel threads must be employed to mount an attack.

In addition to the summary report we supply the complete set of constraints in KQuery format. These constraints can be used by the user to query a SAT solver and obtain valid values that satisfy the constraints, and use those during exploitation. The user may subsequently enrich these results by applying range-analysis techniques [20, 30] to the generated constraints so as to evaluate how strict or permissive they are in a given exploitation scenario. However, this is not required for their direct applicability in exploit construction given that the constraints CopyKat emits are readily usable through SAT solver queries, as we showcase in our evaluation.

4.5 Using CCOs

Once a CCO has been fully characterized, using it in an exploit follows a simple, repeatable workflow. The attacker first pairs the initial vulnerability with a compatible CCO, checking that the corruption primitive can reach the CCO's pointer field. Our database provides the cache, allocation site, and exact field offset of the pointer, allowing quick filtering for in-cache UAF/OOB cases or, in cross-cache settings, enabling allocator-manipulation strategies to force adjacency. The database also exposes the system calls that allocate the CCO and trigger the memory-operation sink, making it easy to reproduce all required operations in practice.

Next, the attacker satisfies the symbolic constraints obtained during concolic execution. These constraints describe (i) values that must appear in the target area, if any; (ii) requirements on attacker-controlled input; (iii) fields within the CCO whose values must be preserved. Because our analysis symbolises only the bytes relevant to reach the memory operation, solving these constraints with an SMT solver yields concrete corruption payloads with minimal effort. The attacker then chooses the intended final write location (e.g., a credential structure) and ensures that this location, or an adjacent counterfeit object, satisfies the counterfeit-area constraints and alignment requirements indicated by the database entry.

Finally, exploitation reduces to orchestrating the corruption window: the attacker allocates the CCO, triggers the pointer corruption at the window identified by our analysis (either between two system calls or via blocking mechanisms when both operations occur within one), and triggers the memory operation, both steps using solver-generated input. In doing so, attacker-controlled bytes are copied to an attacker-chosen address while respecting all semantic constraints –turning the original constrained primitive into a powerful data-only write capability.

In some cases the constraints within the CCO might be too difficult to solve and to use in a practical exploit, as some may involve runtime-dependent addresses (e.g., pointers to heap-allocated objects or function pointers). In such situations, an attacker cannot rely solely on solver-generated input and must instead leverage a UAF primitive to leak the original CCO content, then construct a counterfeit object that preserves the leaked field values while substituting the target pointer to redirect the write.

5 Evaluation

In our evaluation, we seek to understand various aspects concerning the discovery, use and flexibility of CCOs, by addressing the following research questions:

RQ1: **Discovery**. What is the distribution of the CCO across the system caches?

RQ2: **Use**. How precise is the CCO characterization?

RQ3: **Flexibility**. How easy is the use of CCOs in the most used exploitation strategies?

With respect to the `discovery` of CCOs we focus on answering RQ1. RQ1 is important for in-cache exploitation. The more generic and dedicated caches we cover, the easier it is for an attacker to find a suitable object for in-cache attacks. For dedicated caches, however, if the system runs with `CONFIG_SLAB_MERGE_DEFAULT` enabled–as is the case for most Linux distributions–the dedicated cache may be merged into a generic cache, allowing us also to leverage the object discovered for that generic cache. In rare instances, certain objects may enable an attacker to choose the cache into which to allocate by controlling the object's size. In our analysis we consider only the caches observed via the original reproducer. By answering RQ2, we demonstrate how full object characterization improves the attacker's ability to satisfy constraints–either manually or via SAT-solver queries–of the CCOs, and thus employ CCOs during exploitation. Finally, RQ3 investigates the flexibility of CCOs. Specifically, we demonstrate, by means of several end-to-end exploits, their practical usability in several realistic exploitation scenarios. Our analysis targets Linux v6.4.1.

5.1 CCO discovery

We carry out a 32-day fuzzing campaign during which syzkaller flagged 434 operations with the required characteristics; in particular, the overwhelming majority (386) were calls to `memcpy`, followed by 24 calls to `memmove` and 24 simple store instructions. The prevalence of calls to `memcpy` stems from the compiler translating simple memory assignments into calls to memory-copy intrinsics for optimisation purposes. We subsequently generate syzkaller reproducers that can be executed under PANDA for 410 cases; from these we

obtain 341 PANDA recordings. After obtaining the replays we execute the PANDA analyses described in Section 4.3 and successfully obtain metadata for 222 cases.

					# No Constraints		
	Cache Name	**# Objs**	**US**	**Diff**	**CF**	**AB**	**E**
Generic	kmalloc-16	1	1	1	0	1	1
	kmalloc-32	1	1	1	0	0	1
	kmalloc-64	4	4	2	0	0	4
	kmalloc-96	3	3	0	1	0	3
	kmalloc-128	5	5	2	0	0	2
	kmalloc-192	3	3	3	2	0	0
	kmalloc-256	1	1	0	1	0	0
	kmalloc-512	3	3	2	1	0	1
	kmalloc-1k	8	7	1	6	3	5
	kmalloc-2k	6 (1)	6	4	4	1	3
	kmalloc-4k	2	2	0	1	0	2
	kmalloc-8k	2	2	2	0	0	0
CG	kmalloc-cg-32	1	1	0	1	0	1
	kmalloc-cg-256	1	1	0	1	1	0
	kmalloc-cg-512	1	1	0	0	0	0
	kmalloc-cg-2k	2	2	1	1	1	1
	kmalloc-cg-4k	5 (2)	5	2	0	1	3
Specialized	PINGv6	1	1	1	1	1	0
	btrfs_inode	1	1	1	0	0	1
	buffer_head	4	4	4	3	2	1
	dentry	2	2	1	0	0	0
	filp	2	2	2	0	0	0
	hfsplus_icache	1	1	1	1	0	1
	jfs_mp	1	1	0	1	0	0
	names_cache	1	1	0	1	0	1
	ntfs_inode_cache	6	6	4	2	1	0
	p9_req_t	1	1	0	1	0	1
	proc_dir_entry	1	1	0	1	0	1
	skbuff_fclone_cache	1	1	0	1	1	0
	skbuff_head_cache	31	26	0	31	6	1
	sock_inode_cache	18	2	18	2	0	9
	udf_inode_cache	1	1	1	1	1	0
	xfs_inode	1	1	0	1	0	1
	TOTAL	122	100	54	66	20	44

Table 1: Distribution per cache of the verified objects. With US, we report the total number of objects reaching back to userspace, Diff the subset of objects in which the primitive is across different system calls, CF and AB the number of objects with NO constraints at the destination and source (respectively) of the memory-transfer, E the number of objects with NO constraints in the entry portion of the heap object from which the destination pointer is retrieved. Values in parentheses indicate objects identified by both CopyKat and BridgeRouter.

5.2 CCO Verification and Characterization

After collecting metadata for 222 cases, we proceeded to verify and characterize them. Verification is performed with the S2E framework by overwriting the heap pointer that controls the destination of the memory operation, and by observing how the system responds. By performing the overwrite we force the pointer to point to a valid memory region under our control, containing a shallow copy of the original memory area. The analysis completes successfully (i.e., no hangs or timeouts are observed) and the memory-transfer operation successfully targets the overwritten address in 122 cases.

Table 1 summarizes results for the 122 successful cases. It shows the distribution across slab caches, highlighting how the most important slab caches in the kernel contain at least one object. It also shows how many objects had *no* constraints for each of the three memory regions relevant to exploitation, namely, the source of the memory-transfer (AB in the table), the destination of the memory-transfer (CF in the table), and the entry portion of the heap object from which the destination pointer is retrieved (E in the table); in particular, we report the constraints on the bytes that precede the pointer field in the heap object. This is especially relevant in case a linear OOB is used to overwrite the pointer. We also count for how many objects the control flow does not reach a panic after the memory operation (US in the table), and how many objects involve different system calls (Diff), which affect the exploitation requirements as discussed in Section 4.4. Table 1 also includes a comparison with the 5 router objects identified by BridgeRouter [40] with a controlled source buffer (as per CCO definition). In parentheses, we denote the objects that also our CopyKat discovers, namely `msg_msg`, `msg_msgseg`, and `tty_struct`. It misses `seq_file` and `urb` because our fuzzing campaign did not reach them. However, after manually providing syzkaller-like reproducers, our pipeline successfully characterized both objects. Rather than assuming specific bridge-to-router data-flows, our dynamic analysis can capture complex data flows, thereby identifying a substantially larger set of objects (122 vs. 5), and more cache types. Table 3 summarises the distribution of the discovered objects across kernel subsystems, highlighting that the bulk of objects are concentrated in the network and file system code, while also showing that at least one object is present in most subsystems.

RQ1 Takeaway

CopyKat was able to discover, verifying and characterize at least 1 CCO in 84% of the available generic caches and 40% of the CG caches. It has also highlighted a higher concentration of CCOs within the two specialized caches `sock_inode_cache` and `skbuff_head_cache`.

Table 1 reports about how many objects have no constraints. Meanwhile in Table 4, Table 5, and Table 6, we report the detailed results for each of the 122 objects that we successfully verified and characterized highlighting which cache they belong to, and how many constraints per memory region we collect.

Hereafter, we provide a sample of the constraints reported for our running example:

```
(Eq 0x0
 (Extract w32 0 (Add w64 0xfffffffffffffffe
  (ZExt w64 (Read w8 0x11 v0_cco_counterfeit_buf_0)))))
```

The KQuery expression requires that the byte at offset `0x11` of the counterfeit buffer take the value `0x2`, reflecting the check on the `family` field in Listing 1.2. Here, this is enforced by zero-extending the byte, adding `0xfffffffffffffffe` (i.e., –2), and requiring the result to be zero. The attacker must hence be able to ensure that after corruption, the 17th byte of the new memory area is equal to 2.

RQ2 Takeaway

Our recorded constraints relieve the attacker from the tedious process of understanding and correctly crafting inputs and memory areas to build an attack. Through the use of a SAT solver, it is possible to generate valid inputs automatically.

5.2.1 CopyKat Filtering

CopyKat is designed to become progressively more precise and to filter out false positives that were, by design, accepted at earlier stages. Simultaneously, each stage may incorrectly discard true positive candidates, either as a result of deliberate engineering decisions or due to errors or bugs. In this section, we present a numerical breakdown of the filtering within our pipeline and characterize the underlying reasons.

As described in Section 5.1, our fuzzing campaign flags 434 cases in which the CCO pattern is observed, but in 24 cases, syzkaller fails to generate a C reproducer. We then successfully record execution for 341 out of those 410 reproducers under PANDA: in 17 cases we timeout (an engineering decision to keep PANDA record files small), whereas in 52 further cases execution does not reach the panic acting as sink signal: this is likely owing to the different OS images employed (BusyBox for PANDA vs. debootstrap for syzkaller), causing identical syscalls to observe different environments and behave differently. In 13 further cases the PANDA analysis fails (crashes in 8 cases and fails to show signal at memcpy in 5), leaving 328 analyzable cases. For 106 cases, the PANDA analysis fails to detect a data flow from the heap location where the pointer is stored to the memory-transfer operation. These 106 cases are the real false positives: they occur because the code appears to store the destination pointer on the heap and read it back for the memory-transfer, but the compiler avoids that reload entirely by computing the pointer once, storing it to memory for correctness, and keeps using the value directly from a register.

During the verification and characterization process under S2E, our analysis records failures for some of the 222 tested cases. Some of the failures are attributable to imprecision in the information collected during the PANDA analyses which trickle down into S2E decision making. This occurs when the PANDA taint engine ends up in an overtaint scenario–i.e., when a large number of taint labels is observed at the memory-transfer operation due to incorrect, or overly lax, taint propagation policies. When several `copy_from_user` sites are marked, the S2E analysis is forced to symbolize large volumes of data that might run deep into the kernel code, which incurs a considerable slowdown due to the overhead of managing a large symbolic state during concolic execution. In 19 cases, this specific problem causes the system to panic due to stuck CPU. In other 58 cases instead, the failure falls under one of the following categories: imprecision in `store` information; imprecision in the identification of the sink; S2E spurious crashes due to a bug in the symbolic execution engine; or actual false positives–i.e., we successfully reach the memory operation of interest, but the corruption did not affect the destination of the memory-transfer. To provide a more precise breakdown among these 58 cases and pinpoint for each the exact cause of failure requires an in-depth manual analysis that we leave for future work. Instead, we attempt to mitigate the overtaint problem for `copy_from_user` and `store` cases. By disabling sets of identifiers for both `copy_from_user` and `store` when more than one identifier is present in the metadata, we then recursively search for a minimal combination that still preserves valid data flows.

Another issue we encounter is related to the decision to employ only a shallow copy rather than a deep copy of the original memory region. A shallow copy aliases pointers and locks, potentially introducing inconsistencies because the kernel may subsequently reference either the original object or the alias. In 23 cases, we observe an immediate crash after pointer corruption, precluding the execution from reaching the intended operation. While performing verification with a deep copy may resolve the problem, we leave this as future work.

We conduct the verification and characterization analysis for each case with a timeout of one hour. The majority of successful reproducer programs terminate within five minutes, including S2E initialization.

5.3 Exploitation with CCOs

To demonstrate the practical usability of the discovered CCOs, we construct eight end-to-end exploits for real-world CVEs. Table 2 provides details for each. As shown in the table, a limited primitive such as a linear out-of-bounds write (OOB), a use-after-free (UAF) or a double-free (DF) is amplified by a CCO that converts it into an arbitrary-write capability. Arbitrary-write capability in kernel space is–save for rare exceptions– effectively synonymous with full system compromise. We choose however to complete the exploits and obtain `root` privileges on the target system by overwriting pre-selected memory regions that attackers have traditionally targeted to achieve privilege escalation. Typical targets include credential structures (for example their UID/GID

fields) and user-mode helper paths such as `modprobe_path` or `core_pattern`. The exploits demonstrate seven cases in which the store operation that sets the heap pointer, and the memory-transfer that dereferences that pointer are issued in distinct system calls, and a further case in which both operations are performed within the same system call. In the latter situation, an attacker may exploit mechanisms such as FUSE [16] as synchronisation primitives to align the corruption within the narrow window between the final store of the pointer and the memory operation that uses it. Exploits also encompass both in-cache and cross-cache scenarios. Although end-to-end exploitation still involves several non-trivial manual operations, our CCO characterization substantially reduces this burden by guiding the attacker through object selection, corruption, and constraint satisfaction, as we shall show below.

RQ3 Takeaway

Our exploits cover common vulnerability classes, include both in-cache and cross-cache cases, and adopt a heterogeneous set of known final write locations that lead to privilege escalation. This demonstrates the flexibility of CCOs in adapting to various exploitation scenarios.

5.3.1 Use case: CVE-2023-5345

We demonstrate the exploitation of CVE-2023-5345, a double-free vulnerability that permits an object residing in the same cache to be freed twice, employing the CCO described in our running example. As shown in Listing 1.1, the object is allocated from the `kmalloc-192` cache. Exploitation occurs in-cache, because both the vulnerable object and the CCO reside in the same cache. An attacker first allocates and frees the vulnerable object, then reallocates a CCO so that it overlaps the former allocation, and finally frees the CCO using the free primitive associated with the vulnerability. To reclaim the freed slot we allocate an `xattr` object so that it lands in the same `kmalloc-192` hole left by the freed CCO. Because `xattr` objects are var-sized and drawn from generic caches, this gives us precise control over which freelist entry we occupy. We craft the `xattr` payload to serve as a counterfeit CCO whose in-memory layout mirrors the genuine object in Listing 1.1. In particular, we set the `next` pointer to an attacker-chosen target address, and we populate the fields that are consulted by the validation logic in `tcp_md5_do_lookup_exact` with solver-derived values that satisfy the constraints recorded in our database, ensuring the counterfeit passes the lookup and reaches the update path. A subsequent `setsockopt` invocation then triggers the `memcpy` that copies attacker-controlled bytes via the forged `next` pointer into the chosen destination. The only remaining requirement is that the target memory area respect the target-region constraints uncovered by our analysis. The attacker can either choose a location that natively satisfies these constraints or create a correctly shaped counterfeit area around an otherwise desirable location. We adopt the latter and corrupt a `struct cred`: by spraying `pipe_buffer` pages we place well-formed bytes at the offsets demanded by the constraints while arranging adjacency (using PCPLost [25]) so that the `memcpy` successfully overwrites the credential fields.

5.3.2 Use case: CVE-2022-0185

We next exploit CVE-2022-0185, a linear OOB write in `kmalloc-4k`, to corrupt the `absinfo` pointer of a `struct input_dev` CCO allocated in `kmalloc-2k`. Because the objects reside in different caches, we use PCPLost [25] to obtain cross-cache adjacency and position sprayed `input_dev` instances immediately after the vulnerable allocation. In this case, both the pointer-initializing store and the dereferencing load occur within a single system call, leaving only a narrow time window for corruption. We create multiple threads that allocate `input_dev` objects and stall at the load site using a FUSE-based blocking primitive. The first thread reaching this point signals the parent, which immediately triggers the OOB write from the CVE to overwrite the CCO's `absinfo` pointer with the address of the final write target (`core_pattern`), using solver-derived values to satisfy the required constraints. All threads are then released, and the instance with the corrupted pointer reaches the update path and performs the redirected `memcpy`. A monitoring process polls `core_pattern` and, once modified, triggers a fault that executes the attacker-controlled helper binary as `root`, completing privilege escalation. This exploit highlights how CCO-specific constraint information and precise timing windows enable reliable intra-syscall pointer-corruption attacks even in cross-cache settings.

Mitigations The exploits bypass existing hardenings, including `HARDENED_USERCOPY` enabled, `SLAB_MERGE_DEFAULT` disabled, `CGROUPS` and `MEMCG` enabled. Cross-cache mitigations such as `SLAB_VIRTUAL` become ineffective when the attack is performed entirely in-cache. Moreover, the exploits can be adapted to work when more advanced mitigations such as `SLAB_FREELIST_RANDOM` are enabled. Rather than providing a systematic evaluation of mitigation bypasses or exploit reliability, this work focuses on characterising CCOs and their role in data-only attacks.

6 Conclusion

In this paper, we present CopyKat, a practical, fully automated tool to discover controllable-copy objects (CCOs) for data-only attacks. Our pipeline–combining lightweight taint analysis, precise taint verification, and concolic execution–identifies, verifies, and characterizes 122 CCOs across diverse slab caches. We build eight privilege escalation exploits from

CCO ID	CCO Cache	CCO Type	Basic Vulnerability	Vulnerable Cache	Final Write Location
408bf2	**kmalloc-192**	**`struct tcp_md5sig_key`**	**double free (CVE-2023-5345)**	**kmalloc-192**	**`struct cred`**
1a4d23	kmalloc-2k	`struct snd_pcm_plugin_channel`	linear OOB write (CVE-2022-0185)	kmalloc-4k	`modprobe_path`
3c4998	kmalloc-16	`struct aead_tfm`	linear OOB write (CVE-2024-3141)	kmalloc-cg-192	`struct cred`
f69897	kmalloc-512	`struct ucma_context`	use-after-free (CVE-2023-0461)	kmalloc-512	`core_pattern`
6071be	kmalloc-512	`struct snd_seq_queue`	use-after-free (CVE-2023-0461)	kmalloc-512	`core_pattern`
653543	kmalloc-2k	`struct input_dev`	linear OOB write (CVE-2022-0185)	kmalloc-4k	`core_pattern`
c002f2	kmalloc-192	`struct v4l2_subdev_state`	double free (CVE-2023-5345)	kmalloc-192	`struct cred`
895ddc	kmalloc-192	`struct v4l2_subdev_state`	double free (CVE-2023-5345)	kmalloc-192	`struct cred`

Table 2: Summary of our end-to-end exploits. The first row (in bold) indicates our running example.

real-world CVEs, demonstrating that CCO-based amplification is viable despite contemporary defenses. These results show that CCOs form powerful, previously unknown primitives for upgrading constrained memory-corruption bugs, and provide actionable insights for both offensive research and defensive efforts in mitigation design and vulnerability prioritization.

7 Acknowledgments

We would like to thank Damian Pfammatter and the anonymous reviewers for their feedback. This work was supported by: the Swiss National Science Foundation (SNSF Grant no. 10004669), armasuisse Science and Technology, and the Dutch Research Council (NWO) through project "INTERSECT" and "Theseus".

Ethical Considerations

This work inscribes itself in a line of research focusing on finding alternative ways to exploit vulnerabilities in complex systems such as the Linux kernel, under state-of-the-art mitigations. Such work does not find new vulnerabilities, but instead helps in assessing the exploitability of existing and future vulnerabilities, as well as evaluating the effectiveness of modern exploit mitigations. We perform a stakeholder-based analysis and follow the Menlo principles to evaluate the potential harms and benefits of our work, and explain why we decided to pursue this line of research.

Stakeholder analysis

We consider both direct and indirect stakeholders who may be impacted by publication and artifact release.

Research community: researchers studying exploitability, mitigations, and automated analysis, who benefit from reproducible methodology and clear threat models.

Linux kernel maintainers and defenders: Linux kernel developers, distro security teams, and incident responders who benefit from understanding new exploitation pathways and from hardening guidance.

Linux kernel users and deployers: end users, enterprises, cloud/container operators whose systems could be harmed if amplifier objects enable new reliable privilege escalation pathways.

Downstream vendors: downstream distributions and device vendors who may need to triage risk and deploy mitigations across configurations.

Respect for Persons

In our setting, the primary risks are not interpersonal interaction but *downstream exposure* (e.g., enabling misuse of exploitation know-how). To respect persons potentially affected by this exposure, we avoid non-consensual interaction with

third-party systems: experiments and measurements are constrained to systems and environments under the researchers' authorization and control (local testbeds and VMs). In addition, we have contacted one of the main kernel maintainers (Greg Kroah-Hartman), to notify the community. Their answer agreed with our point of view that this work can be published without embargo as it does not put systems directly at risk, given that it affects exploitation techniques and mitigations.

Beneficence

Our work offers clear defensive benefits (understanding amplifier-objects, enabling mitigation design) but also introduces risks (accelerating exploitation). We mitigate these risks through:

- Risk-benefit assessment: we explicitly articulate the attacker model enabled by CCO, including constraints (e.g., constrained writes and surrounding-memory constraints) so defenders can reason about realistic exploitability without overstating "arbitrary write" outcomes.
- Minimization of harm in evaluation: all exploitation demonstrations (8) are performed in controlled environments and are scoped to demonstrate scientific claims (e.g., privilege escalation feasibility under constrained writes), avoiding collateral impact.
- Careful artifact release: to reduce misuse, we do not release weaponized exploits: the released artifacts are structured to support reproducibility without directly enabling opportunistic exploitation. Should researchers in the community require our exploits for legitimate reasons, we will carefully vet requests and answer them.

Justice

The Menlo report frames Justice as fairness and equity in the distribution of research burdens and benefits. Kernel exploitation research can disproportionately burden less-resourced operators (e.g., small organizations and open-source maintainers) if it triggers patch churn or public exploitation. However, this work does not put systems at risk nor require immediate patches, as evidenced by the Linux kernel maintainers' reply to our notification.

Respect for Law and Public Interest

We comply with legal and policy constraints: experiments are conducted under authorization on controlled systems. For the public interest, we practice transparency and document assumptions, evaluation environment constraints, and the scope of artifacts released so that readers can assess residual risk and replicate results ethically.

Summary

We reached the decision to proceed with our research based on the fact that the benefits (improved understanding of kernel exploitation and mitigation) vastly outweighed the risks (accelerating exploitation). This aligns the project with the Menlo principles and their recommended operationalization for ICT security research.

Open Science

Along with this submission we publish our artifacts. We publish our changes to llvm, syzkaller, panda and s2e in patch format and additional C code format. Furthermore, we provide our objects database in json format along with our characterization of the constraints in KQuery format. The repository is organized in folders concerning every stage of our pipeline:

`kernel_instrumentation`: anything concerning building our modified kernel used as base of all the analysis.

`fuzzing`: all the changes made to Syzkaller to support lightweight taint analysis.

`taint_analysis`: all the changes and script to run PANDA.

`symb_ex`: all our changes to S2E, our S2E plugin and the scripts required.

`results`: contains our DB and our constraints file that are the output of the pipeline.

Everything was released and can be found at: https://zenodo.org/records/20427287

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), November 2009. https://doi.org/10.1145/1609956.1609960.

[2] Erin Avllazagaj, Yonghwi Kwon, and Tudor Dumitras. SCAVY: automated discovery of memory corruption targets in linux kernel for privilege escalation. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[3] bcoles. Kasld, n.d. https://github.com/bcoles/kasld.

[4] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1093–1110. USENIX Association, 2020.

[5] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A systematic study of elastic objects in kernel exploitation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1165–1184, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Yueqi Chen and Xinyu Xing. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*, pages 1707–1722. ACM, 2019.

[7] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 265–278, New York, NY, USA, 2011. Association for Computing Machinery. https://doi.org/10.1145/1950365.1950396.

[8] Kees Cook. Kernel space layout randomization, 2013. https://outflux.net/slides/2013/lss/kaslr.pdf.

[9] Kees Cook. [rfc patch 0/7] introduce kernel control flow integrity abi [pr107048], 2025. https://lore.kernel.org/linux-hardening/20250821064202.work.893-kees@kernel.org/.

[10] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, page 5, USA, 1998. USENIX Association.

[11] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Jake Edge. Control-flow integrity for the kernel, 2020. https://lwn.net/Articles/810077.

[13] Dmitry Vyukov et al. Syzkaller repository, 2016. https://github.com/google/syzkaller.

[14] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. Fineibt: Fine-grain control-flow enforcement with indirect branch tracking. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '23, page 527–546, New York, NY, USA, 2023. Association for Computing Machinery.

[15] Google. Kernelctf writeups, n.d. https://github.com/google/security-research/tree/master/pocs/linux/kernelctf.

[16] Alejandro Guerrero. Linux kernel slab out-of-bounds write: exploit and writeup, 2022. https://www.openwall.com/lists/oss-security/2022/01/25/14.

[17] David Howells. Credentials in linux. The Linux Kernel Documentation, 2024.

[18] Max Kellermann. The dirty pipe vulnerability, 2022. https://dirtypipe.cm4all.com/.

[19] Anil Kurmus, Andrea Mambretti, Alessandro Sorniotti, Vincent Lenders, Damian Pfammatter, and Bernhard Tellenbach. SoK: Automating kernel vulnerability discovery and exploit generation. In *19th USENIX WOOT Conference on Offensive Technologies (WOOT 25)*, pages 283–302, Seattle, WA, August 2025. USENIX Association.

[20] Guilhem Lacombe and Sébastien Bardin. Attacker control and bug prioritization. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, USA, 2025. USENIX Association.

[21] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. Pspray: Timing {Side-Channel} based linux kernel heap exploitation technique. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6825–6842, 2023.

[22] Yoochan Lee, Hyuk Kwon, and Thorsten Holz. Dirtyfree: Simplified data-oriented programming in the linux kernel. In ISOC, editor, *NDSS 2026, Network and Distributed System Security Symposium, 23-27 February 2026, San Diego, CA, USA*, San Diego, 2026.

[23] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1963–1976, New York, NY, USA, 2022. Association for Computing Machinery.

[24] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary memory writes through practical software

cross-cache attacks within the linux kernel. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 2024.

[25] Claudio Migliorelli, Andrea Mambretti, Alessandro Sorniotti, Vittorio Zaccaria, and Anil Kurmus. Cross-cache attacks for the linux kernel via pcp massaging. In ISOC, editor, *NDSS 2026, Network and Distributed System Security Symposium, 23-27 February 2026, San Diego, CA, USA*, San Diego, 2026.

[26] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. DROP THE ROP Fine-grained Control-flow Integrity for the Linux Kernel. BlackHat Asia'17, 2017.

[27] Joao Moreira, Mark Rutland, Peter Zijlstra, and Sami Tolvanen. Linux Kernel Control-Flow Integrity Support. Linux Plumbers Conference, 2022.

[28] n.d. Control-flow enforcement technology (cet) shadow stack - the linux kernel documentation, n.d. https://www.kernel.org/doc/html/next/x86/shstk.html.

[29] n.d. Guarded control stack support for aarch64 linux - the linux kernel documentation, n.d. https://docs.kernel.org/arch/arm64/gcs.html.

[30] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, page 73–85, New York, NY, USA, 2009. Association for Computing Machinery.

[31] Dong ok Kim, Juhyun Song, and Insu Yun. CROSS-X: Generalized and stable cross-cache attack on the linux kernel. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25)*. ACM, 2025.

[32] Aleph One. Smashing the stack for fun and profit. https://phrack.org/issues/49/14.

[33] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 552–561, New York, NY, USA, 2007. Association for Computing Machinery. https://doi.org/10.1145/1315245.1315313.

[34] syzkaller team. Research work based on syzkaller, 2024. https://github.com/google/syzkaller/blob/master/docs/research.md.

[35] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[36] Android team. Kernel control flow integrity, 2024. https://source.android.com/docs/security/test/kcfi.

[37] Ruipeng Wang, Kaixiang Chen, Chao Zhang, Zulie Pan, Qianyu Li, Siliang Qin, Shenglin Xu, Min Zhang, and Yang Li. Alphaexp: An expert system for identifying security-sensitive kernel objects. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 4229–4246. USENIX Association, 2023.

[38] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1187–1204, Santa Clara, CA, aug 2019. USENIX Association. https://www.usenix.org/conference/usenixsecurity19/presentation/wu-wei.

[39] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[40] Dongchen Xie, Dongnan He, Wei You, Jianjun Huang, Bin Liang, Shuitao Gan, and Wenchang Shi. BridgeRouter: Automated capability upgrading of out-of-bounds write vulnerabilities to arbitrary memory write primitives in the linux kernel. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 772–790, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[41] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 590–604. IEEE Computer Society, 2014.

[42] Kyle Zeng, Zhenpeng Lin, Kangjie Lu, Xinyu Xing, Ruoyu Wang, Adam Doupé, Yan Shoshitaishvili, and Tiffany 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*, CCS '23, page 3093–3107, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3576915.3623220.

[43] Jinmeng Zhou, Jiayi Hu, Ziyue Pan, Jiaxun Zhu, Wenbo Shen, Guoren Li, and Zhiyun Qian. Beyond control: Exploring novel file system objects for data-only attacks on linux systems. *CoRR*, abs/2401.17618, 2024.

[44] Peter Zijlstra. [rfc][patch 0/6] x86: Kernel ibt beginnings [lwn.net], 2021. https://lwn.net/ml/linux-kernel/20211122170301.764232470@infradead.org/.

[45] Peter Zijlstra. x86/ibt: Implement fineibt, 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=931ab63664f0.

Kernel Subsystem	# Objects
crypto	7
drivers	12
fs	31
ipc	1
kernel	3
lib	10
mm	1
net	52
security	2
sound	3
TOTAL	**122**

Table 3: Object count per kernel subsystem.

Additional CCO Exploits

CCO 1a4d23 CVE-2022-0185 enables a linear OOB write in kmalloc-4k, and we target a `struct snd_pcm_plugin_channel` CCO in kmalloc-2k to corrupt its `addr` field and obtain a 1-byte controlled write primitive. Since the exploit is cross-cache, we use PCPLost to obtain object adjacency. Then, we repeat the OOB write in order to redirect multiple `memcpy` operations to incrementally overwrite the final write location (i.e., `modprobe_path`). Finally, we trigger `call_modprobe()` to execute an attacker-controlled binary with elevated privileges.

CCO 3c4998 CVE-2024-53141 allows repeated OOB writes of two unsigned long values in kmalloc-cg-192, which we use to corrupt the `aead` pointer of a `struct aead_tfm` CCO in kmalloc-16 and redirect the `memcpy` to an attacker-controlled counterfeit area. Cross-cache adjacency between the vulnerable and the CCO is achieved via PCPLost. Similarly to the exploit described in Section 5.3, `pipe_buffer` is placed next to a credential to satisfy a constraint involving a fixed address. The redirected write corrupts the UID field to zero, and multiple forked children are checked to identify the process that gained root privileges.

CCO 6071be CVE-2023-0461 is a use-after-free vulnerability that allows an object in the kmalloc-512 cache to be freed twice. We leverage this primitive to leak a `struct snd_seq_queue` CCO in the same cache via `user_key_payload`, enabling us to craft a counterfeit object with a corrupted `timer` pointer field that redirects the write to `core_pattern`. To obtain the leak, we first overlap the `user_key_payload` with the freed vulnerable object, free it, and subsequently allocate the CCO in its place. Because the CCO contains an attacker-controllable field that aligns precisely with the `datalen` field of `user_key_payload`, we set this field to a value large enough to capture the entire CCO content. We then trigger the `user_key_payload` read primitive to extract the leaked data and allocate an `xattr` object to construct the counterfeit CCO with the modified `timer` field pointing to the final write location. Finally, we trigger the CCO write primitive. A monitoring process polls `core_pattern` and, upon detecting modification, triggers a fault that executes the attacker-controlled helper binary with elevated privileges.

CCO f69897 CVE-2023-0461 is a use-after-free vulnerability that allows an object in the kmalloc-512 cache to be freed twice. We leverage this primitive to leak a `struct ucma_context` CCO in the same cache via `user_key_payload`, enabling us to craft a counterfeit object with a corrupted `cm_id` pointer field that redirects the write to `core_pattern`. To obtain the leak, we first overlap a `user_key_payload` with the freed vulnerable object and we allocate the CCO in the next slot by performing spraying. We then overlap a `cmsg` object with the freed `user_key_payload` and set attacker-controlled data at the offset corresponding to the `datalen` field, causing the subsequent read primitive to disclose the contents of the adjacent CCO. This strategy is required because directly overlapping the CCO would overwrite this field with uncontrollable data. Triggering the `user_key_payload` read primitive yields the contents of the CCO. Next, we reallocate a CCO over the freed `cmsg`, we free it via the `user_key_payload` primitive, and we overlap an `xattr` object to construct the counterfeit CCO with the modified `cm_id` field pointing to the final write location. Finally, we trigger the CCO write primitive. A monitoring process polls `core_pattern` and, upon detecting modification, triggers a fault that executes the attacker-controlled helper binary with elevated privileges.

CCO c002f2 and CCO 895ddc CVE-2023-5345 enables a double-free in kmalloc-192, which we use to leak and counterfeit a `struct v4l2_subdev_state` CCO in the same cache, corrupting its `pads` pointer to redirect the `memcpy`. We discovered two variants of the same CCO, with different code paths to reach the memory-transfer operation. To obtain the leak, we first overlap a `user_key_payload` with the freed vulnerable object and we allocate the CCO in the next slot by performing spraying. We then overlap a `cmsg` object with the freed `user_key_payload` and set attacker-controlled data at the offset corresponding to the `datalen` field, causing the subsequent read primitive to disclose the contents of the adjacent

```
{
    "id": "408bf2a23593a83e9ab66cc2982c30d87e52b223",
    "fname": "__tcp_md5_do_add",
    "off": 104,
    "sl": "data_race(memcpy(key->key, newkey, newkeylen));",
    "call_id": "12464",
    "faddr2line": {
        "main": {
            "line": "net/ipv4/tcp_ipv4.c",
            "num": "1198"
        }
    },
    "joern_analysis": {
        "operators": [ ... ],
        "type": "tcp_md5sig_key",
        "typedef": [...]
    },
    "replay": {
        "memcpy_ctr": 1,
        "sink": {
            "call_id": 12464,
            "instr_ctr": 10718751805,
            "k_bt": 471,
            "len": 27,
            "syscall": {
                "u_bt": 813
                ...
            }
        },
        "ptr_ass": {
            "call_id": 5066, ... //Similar to sink entry
        },
        "target_obj_alloc": {
            "cache": "b'kmalloc-192'", ... //Similar to sink
            ↪ entry
        },
        "dst_ptr_alloc": {
            "cache": "_kmalloc_", ... //Similar to sink entry
        },
        "taint_src": [
            {
                "cfu_id": 8171, ... //Similar to sink entry
            },
        ],
    },
}
```


Listing 2: Excerpt from the database entry for the running example, built by the fuzzing and dynamic analysis steps, used as input for the concolic execution phase.

CCO. This strategy is required because directly overlapping the CCO would overwrite this field with uncontrollable data. Triggering the `user_key_payload` read primitive yields the contents of the CCO. Next, we reallocate a CCO over the freed `cmsg`, we free it via the `user_key_payload` primitive, and we overlap an `xattr` object to construct the counterfeit CCO with the modified `pads` field pointing to the final write location. Finally, we trigger the CCO write primitive. We choose `struct cred` as the final write location because the CCO write primitive allows control over a limited number of bytes, making it impractical to corrupt targets such as `core_pattern` or `modprobe_path`.

DB Entries

Listing 2 shows an extract of the database entry for our running example. This database is built in the first steps of the analysis and used as input for the concolic execution phase.


```
{
 "id": "408bf2a23593a83e9ab66cc2982c30d87e52b223",
 "type_syscall": "different",
 "memory_operation": "memcpy",
 "memop_signal": true,
 "panic": false,
 "memop": {
   "reached": true,
   "tot_constraints": 11,
   "counterfeit": 8,
   "counterfeit_bytes": 11,
   "attacker_buf": 8,
   "attacker_buf_bytes": 9,
   "cco_entry": 0,
   "cco_entry_bytes": 0
 },
 "return": {
    // Same as above
 }
}
```


Listing 3: Excerpt from the database entry for the running example, built by the concolic execution step. The record shows that the object controls the sink of a `memcpy`, that pointer setting and memory operation occur in different system calls, along with the total constraints and how they split across the relevant memory regions.

Listing 3 shows an extract of the database entry, built for the same running example but now highlighting the results of the concolic execution step.

Detailed CopyKat results

In this section, we report our successfully verified objects and their characterization–i.e., the number of constraints collected in the three memory area of interest, the cache where the object is allocated, if the object return to userspace after the memory operation, and if the primitive is across multiple system calls. Table 4 reports the results for both different and same type for all the objects that do NOT return to userspace. Table 5 and Table 6 instead report the characterized objects that successfully return to userspace after the memory operation.

			Memory Operation		
	CCO ID	CACHE	CF	AB	E
Different	128479	sock_inode_cache	20322	1	4
	64a5f2	sock_inode_cache	40	90	15
	6e675f	sock_inode_cache	159	2	0
	70dc58	sock_inode_cache	20162	4	4
	78cb32	sock_inode_cache	20303	2	0
	8154d3	sock_inode_cache	20196	49	0
	8c3bdc	sock_inode_cache	20207	33	0
	9432b4	sock_inode_cache	20325	1	0
	9ada95	sock_inode_cache	20155	7	4
	9f911d	sock_inode_cache	22098	10	0
	ac162c	sock_inode_cache	173	106	20
	ac5857	sock_inode_cache	20162	6	0
	b0c5fe	sock_inode_cache	82	119	19
	b56187	sock_inode_cache	20211	1	4
	c9f21b	sock_inode_cache	20188	8	4
	e9702c	sock_inode_cache	27	2	7
Same	2e1d68	kmalloc-1k	0	0	0
	4f5d80	skbuff_head_cache	0	8948	2
	67397c	skbuff_head_cache	0	0	3
	8dc4ae	skbuff_head_cache	0	0	1
	93ae6b	skbuff_head_cache	0	0	4
	ec3c23	skbuff_head_cache	0	273	1

Table 4: List of successfully verified reproducers that panic after the verified memory operation. This list includes both cases different or same system call. We report in which cache the object is allocated and the total number of constraints collected for the destination of the memory-transfer (CF), for the source of the memory-transfer (AB) and entry portion of the heap object from which the destination pointer is retrieved (E).

		Memory Operation			Return		
CCO ID	CACHE	CF	AB	E	CF	AB	E
0bb9de	kmalloc-cg-2k	513	313	108460	2804	388	108566
0c9d9e	sock_inode_cache	0	136	0	0	138	0
1099f4	sock_inode_cache	0	592	0	0	592	0
181a9c	kmalloc-2k	45	234	36	46	1053	48
1a4d23	kmalloc-2k	0	0	0	0	64	0
1f7c83	kmalloc-1k	0	66804	46	0	66804	452
3328a2	kmalloc-cg-4k	2	79	0	36514	79	0
372012	kmalloc-2k	0	6	27	1	7	39
3c4998	kmalloc-16	12	0	0	12	0	0
3f0ef2	buffer_head	0	0	1	0	0	1695
408bf2	kmalloc-192	8	8	0	8	8	0
472de5	kmalloc-192	4	301	974	4	301	985
53302f	ntfs_inode_cache	1716	91296	44	3397	91545	289
605844	kmalloc-cg-4k	17	1694	0	21	1694	0
6071be	kmalloc-512	408	23	0	459	145	6
63f5a4	ntfs_inode_cache	993	216634	8	2502	234299	16
895ddc	kmalloc-192	0	262	84	0	262	95
898f8f	dentry	2523	4140	237	3634	4141	238
8a4e6e	udf_inode_cache	0	0	8	0	721	295
94f213	hfsplus_icache	0	29558	0	0	29584	0
a3d8e6	ntfs_inode_cache	382	177742	25	45284	197054	25248
a8e47f	ntfs_inode_cache	2290	0	44	3179	0	72
abf63e	btrfs_inode	2041	5	0	722276	19843	0
b04d9d	kmalloc-128	5	1756	0	77	2028	0
b2f9fe	kmalloc-8k	28	86260	7398	23867	92224	8666
be6586	kmalloc-64	44	1530	0	92	1554	0
c002f2	kmalloc-192	0	18	84	0	18	95
c1a850	PINGv6	0	0	16309	0	0	16331
c21e24	buffer_head	0	0	0	0	0	0
cf2cba	buffer_head	24	11748	1474	38	14092	3410
d3e0ae	filp	7	172	85	10	172	85
e60afb	kmalloc-64	110	22	0	312	46	0
eba5d8	kmalloc-2k	85	51	123	113	54	263
eece01	kmalloc-128	5	2	0	5	2	0
f69897	kmalloc-512	17	16	4	26	26	7
fa417a	filp	5	256	47	16	1576	85
fc433a	kmalloc-8k	1	61340	54	737	68800	152
fedb71	buffer_head	0	47489	48	0	48042	659

Table 5: List of successfully verified reproducers that return to userspace where the primitives reside across different system calls. We report in which cache the object is allocated and the total number of constraints collected for the destination of the memory-transfer (CF), for the source of the memory-transfer (AB) and entry portion of the heap object from which the destination pointer is retrieved (E).

		Memory Operation			Return		
CCO ID	CACHE	CF	AB	E	CF	AB	E
01a9c8	skbuff_head_cache	0	0	1	0	0	515
074193	skbuff_head_cache	0	0	0	0	0	0
0dd6b6	skbuff_head_cache	0	729	4	0	731	7
1f6570	kmalloc-96	0	439	0	0	439	0
284c2d	kmalloc-1k	845	0	0	2795	4149	6
28a508	p9_req_t	0	8362	0	0	8492	0
291aa1	skbuff_head_cache	0	31931	3	0	33520	78
2d7c47	kmalloc-64	106022	45024	0	106023	56075	0
38c2cd	skbuff_head_cache	0	0	4	0	0	4
3b24a8	kmalloc-1k	0	13864	0	0	13894	0
3cbec9	kmalloc-512	0	16229	1	0	16288	80
3d6373	kmalloc-cg-512	45	5	1	343	7	7
3ef901	kmalloc-128	68	1549	4	88	2550	8
4548cb	kmalloc-128	24	3738	4	64	4719	4
462028	kmalloc-cg-4k	44	165	26	45	191	3019
48188e	ntfs_inode_cache	0	2429	36	0	39276	36
50bb48	kmalloc-cg-32	0	460	0	0	460	0
52607f	skbuff_head_cache	0	170	4	0	173	4
55c46a	skbuff_head_cache	0	231	2	0	231	86
562661	skbuff_head_cache	0	795	6	0	824	22
58b3b1	skbuff_head_cache	0	147	1	0	148	15
653543	kmalloc-2k	0	7	0	0	7	0
6bb114	kmalloc-4k	456	4094	0	25987	607463	0
6e395f	skbuff_head_cache	0	280	2	0	566	14
7059d7	skbuff_head_cache	0	459	10	0	461	13
723989	dentry	5	1071	4	13	1141	14
744da6	skbuff_head_cache	0	400	1	0	585	398
75fb70	kmalloc-1k	20	0	0	112	3469	6
785722	kmalloc-cg-4k	1	0	0	23	0	0
7ac169	xfs_inode	0	113	0	1	117	39
7ca268	skbuff_head_cache	0	44	1	0	106	88
7cb653	kmalloc-cg-4k	44	3294	39	44	3633	3922
8dc0f0	proc_dir_entry	0	594	0	0	2287	1
92dec8	kmalloc-1k	0	75147	1392	0	96380	23066
9429b5	skbuff_head_cache	0	207	2	0	313	12
96aabe	skbuff_head_cache	0	22	1	0	22	15
9d7709	jfs_mp	0	11341	1	0	11341	5
a46dc0	skbuff_head_cache	0	92	21	0	92	34
ac1cc6	ntfs_inode_cache	0	81771	44	0	84639	60
b169cf	kmalloc-64	1	29769	0	2612	30322	0
bebbfc	kmalloc-96	218	195302	0	346	197579	1793
bf4732	kmalloc-256	0	16232	13	0	16249	13
c07d0a	kmalloc-2k	0	185	0	0	244	0
c194a9	kmalloc-128	64	435	4	606	741	40
c477e8	skbuff_head_cache	0	12	1	0	34	7
c49f7a	kmalloc-1k	0	981	0	0	4532	163
c52a95	kmalloc-cg-2k	0	0	0	0	0	1155
c6a2a8	kmalloc-96	33	201568	0	739	209206	1859
c7ac6e	skbuff_head_cache	0	367	3	0	424	101
d066d0	kmalloc-4k	0	1048	0	1422	1111	0
d55bac	names_cache	0	4308	0	29	4389	0
d675da	skbuff_head_cache	0	3558	2	0	3697	13
d6be19	kmalloc-cg-256	0	0	4	0	0	10
e10805	skbuff_fclone_cache	0	0	2	0	0	7
e19230	skbuff_head_cache	0	1516	186	0	1516	210
e1b086	kmalloc-1k	0	183	19	0	183	257
e5f3b2	skbuff_head_cache	0	44	2	0	44	6
ed78e9	skbuff_head_cache	0	610	3	0	654	16
f082a4	skbuff_head_cache	0	17	1	0	18	17
f16a16	skbuff_head_cache	0	3879	15	0	5432	554
f1b2f6	skbuff_head_cache	0	3458	6	0	3596	27
f3244a	skbuff_head_cache	0	490	2	0	646	14

Table 6: List of successfully verified reproducers that return to userspace where the primitives reside in the same system call. We report in which cache the object is allocated and the total number of constraints collected for the destination of the memory-transfer (CF), for the source of the memory-transfer (AB) and entry portion of the heap object from which the destination pointer is retrieved (E).