
# SCAVY: Automated Discovery of Memory Corruption Targets in Linux Kernel for Privilege Escalation

Erin Avllazagaj, Yonghwi Kwon, and Tudor Dumitraș, *University of Maryland*

https://www.usenix.org/conference/usenixsecurity24/presentation/avllazagaj

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

# SCAVY: Automated Discovery of Memory Corruption Targets in Linux Kernel for Privilege Escalation

Erin Avllazagaj, Yonghwi Kwon, Tudor Dumitraș
*University of Maryland, College Park*

## Abstract

Kernel privilege-escalation exploits typically leverage memory-corruption vulnerabilities to overwrite particular target locations. These memory corruption targets play a critical role in the exploits, as they determine which privileged resources (e.g. files, memory, and operations) the adversary may access and what privileges (e.g. read, write, and unrestricted) they may gain. While prior research has made important advances in discovering vulnerabilities and achieving privilege escalation, in practice the exploits rely on the few memory corruption targets that have been discovered manually so far.

We propose SCAVY, a framework that automatically discovers memory corruption targets for privilege escalation in the Linux kernel. SCAVY's key insight lies in broadening the search scope beyond the kernel data structures explored in prior work, which focused on function pointers or pointers to structures that include them, to encompass the remaining 90% of Linux kernel structures. Additionally, the search is bug-type agnostic, as it considers any memory corruption capability. To this end, we develop novel and scalable techniques that combine fuzzing and differential analysis to automatically explore and detect privilege escalation by comparing the accessibility of resources between executions with and without corruption. This allows SCAVY to determine that corrupting a certain field puts the system in an exploitable state, independently of the vulnerability exploited. SCAVY found 955 PoC, from which we identify 17 new fields in 12 structures that can enable privilege escalation. We utilize these targets to develop 6 exploits for 5 CVE vulnerabilities. Our findings show that new memory corruption targets can change the security implications of vulnerabilities, urging researchers to proactively discover memory corruption targets.

## 1 Introduction

Kernel exploitations typically *start* by triggering vulnerabilities that enable memory corruption and *end* by overwriting particular memory locations with specific values, to put the system in an exploitable state where the attacker may change the system's behavior. These targeted memory locations for corruption, which we call *memory targets* hereafter, are *orthogonal* to the exploited vulnerability: attackers may use them again when exploiting other vulnerabilities, and defenders often focus on the memory targets to prevent the exploitation of unknown vulnerabilities. For example, Figure 1 shows an exploit created by combining a vulnerability and a memory target (A), resulting in multiple combinations (or exploits) such as B and C. While prior research has focused on discovering and exploring vulnerabilities, comparatively less attention was given to the discovery of memory targets that are needed in the last step of the exploitation process.

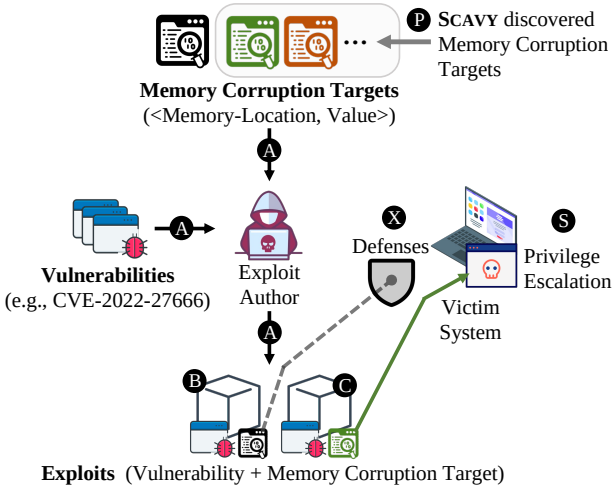


Figure 1: Memory Targets in Exploit

Assume that a privilege escalation exploit (B in Figure 1) seeks to corrupt *a function pointer* in a kernel data structure to execute a malicious payload that escalates the privilege. Unfortunately, the attacker in this exploit utilizes an *unnecessarily strong* capability, leading to a rather *obvious* exploitation method (i.e., control flow hijacking) for privilege escalation which is prevented by popular defenses such as CFI (X). A *new memory target* can enable the attacker to escalate the privilege (C) without being detected (S). For

example, corrupting other memory targets such as `username`, `inode` numbers, or the `task_struct::addr_limit` field, can accomplish privilege escalation. In particular, the `addr_limit` field [1] was famously employed in 2019 by the NSO group to install the Pegasus spyware on Android devices [2]. The vulnerability had been found by Syzkaller in 2017, but it was not patched in many released devices for many years, as its security implications were unknown [2]. As a result, in 2020, the `addr_limit` was used in over 40% of the Android kernel exploits in the wild [3]. This results in the removal of the field from the Linux kernel to prevent such attacks [4]. As such, the discovery of new memory targets enables the exploitation of many vulnerabilities including those that are *previously considered unexploitable or not critical* [5]. Moreover, new types of memory targets can enable exploits to evade existing defenses focusing on the popular memory targets [6–10]. Discovering memory targets is also beneficial in a defensive context, as several techniques (e.g., freelist pointer obfuscation [11]) can secure these kernel objects [12].

Despite its importance, finding memory targets has been challenging and typically done *manually*, relying on expertise in the Linux kernel code base. The prior research on automating kernel exploit generation [13, 14] has focused on exploring specific vulnerabilities with limited types of memory targets (e.g., function pointers or reference-count fields). Consequently, the fields of Linux kernel data structures that privilege-escalation exploits can target have not been systematically researched. For instance, among the 6,582 structures in the Linux kernel version 5.15.80, only 746 (11.3%) of them contain either function pointers (142; 2.1%) or pointers to structures containing function pointers (604; 9.2%) that are considered security-sensitive and have been actively addressed by previous work. The remaining 88.7% represent an *unexplored attack surface*. Moreover, even for the previously addressed 11.3%, an additional thorough search is desirable as [13, 14] only examined reachable memory areas from existing proof-of-concept (PoC) code and [15] focuses on a few kernel objects that induced crashes during the analysis.

We propose SCAVY[1], a framework for systematically discovering *memory targets* in the Linux kernel for privilege escalation (Ⓟ in Figure 1). SCAVY searches for *broader types* of memory targets, beyond the pointer-type memory targets that existing techniques focus on [16–19]. In particular, SCAVY aims to discover new diverse types of targets that can impact (or enable) many existing and unknown vulnerabilities.

During the analysis of the kernel structures and memory areas for the memory targets, SCAVY encounters two challenges: (1) examining a large number of potential memory targets and (2) lack of oracles that can detect diverse forms of potential privilege escalations. SCAVY handles the two challenges by leveraging a differential analysis-based multi-execution reasoning, that are more efficient than existing techniques rely on symbolic and taint analysis. Specifically, it runs multiple executions with and without memory corruption and checks the program states of the executions in terms of the accessibility of security-sensitive resources. The results are analyzed to guide the search process and detect privilege escalations.

SCAVY generated 955 PoC exploits that can potentially escalate privileges, by corrupting 275 unique fields in 86 kernel structures. From the PoC exploits from `CVE-2022-27666`, we create *two* fully functional privilege escalation exploits with new SCAVY identified memory targets (Section 3.1 and Section A.2). Our exploits do not require bypassing popular kernel defenses (e.g., KASLR, SMEP/SMAP, and CFI), unlike the original exploits that had to handle them.

In summary, we make the following contributions:

- We revisit the definitions of memory targets and post corruption program states to identify broader types of memory targets in the Linux kernel.
- We design and implement SCAVY, a framework that systematically discovers memory targets, along with the values needed to achieve privilege escalation. We discuss the design choices for each step of the search and the differential analysis-based execution reasoning.
- We evaluate the effectiveness of SCAVY by finding 20 memory targets in 12 kernel structures, where, notably, *17 of the targets have not been used in publicly available privilege escalation exploits*. We demonstrate that the memory targets found by SCAVY are vulnerability agnostic by developing 6 exploits for 5 different CVEs.
- We open-source our technique for future research [20].

## 2 Problem Statement

Despite important advances in *discovering* software vulnerabilities, incorrect assessments of their security implications remain common and can have a pernicious impact. For example, expert recommendations for prioritizing patches [21, 22] initially omitted `CVE-2017-0144`, the vulnerability later exploited by WannaCry and NotPetya; `CVE-2019-2215` was discovered in 2017 but was not patched in many Android devices because its security implications were unknown, before it was exploited by the NSO group [2]. *Exploitability* assessments are challenging because they require reasoning about state machines with an unknown state space and emergent instruction semantics [23], known as "weird machines". For privilege escalation, in particular, exploitability assessments hinge on the ability to overwrite specific memory targets.

### 2.1 Problem Definition

SCAVY aims to solve a problem of finding memory targets that can lead to privilege escalation. In our context, a memory target is a field of kernel data structure and privilege escalation is defined as a change of a privilege that allows access to an

[1] SCAVY is an abbreviation for 'Scavenger.'

unauthorized resource without using legitimate methods (e.g., permission changing APIs). To facilitate the discussion, we divide a program's execution state into three different states based on how an exploit progresses to achieve its ultimate goal (e.g., taking over the system's control).

1. Before an exploit starts, a process's execution is in an **unexploited state**.
2. After an exploit triggers a vulnerability to corrupt system states to escalate privilege, granting access to resources the exploit wants to compromise, the state becomes an **exploitable state**. This state has access to the exploit's target resources, meaning that the exploit has *achieved all accesses needed* for its ultimate objective but *has yet to achieve* it (i.e., has not compromised the system yet).
3. After the exploit achieves its objective, the execution state becomes an **exploited state**.

**Threat Model.** We assume a local unprivileged adversary seeking to use a vulnerability in the Linux kernel with a *memory target by* SCAVY to escalate privileges. This threat model is relevant in various settings, such as cloud computing (e.g., Docker), Android [24], and malware exploiting kernel vulnerabilities [25]. As SCAVY finds memory targets for any given exploits, our threat model includes a wide range of diverse memory corruption capabilities of exploits. Note that it also means that SCAVY found memory targets may require vulnerabilities with certain capabilities. As described in Section 4 and Section 3, creating an end-to-end functional exploit from the SCAVY identified memory targets requires manual efforts. Automating the exploit generation process is out of the scope.

### 2.1.1 Unexploited, Exploitable, and Exploited States

Figure 2 illustrates the two critical states for our problem definition (i.e., unexploited and exploited states), including privileges of various resources under the states. We first define five different types of privileged resources in Figure 2:

- **Read-only Resources**: These are the files that are configured to be read-only (e.g., Apache [26]'s configuration file are read-only to prevent unauthorized modifications [27]). `setuid` files are also read-only.
- **Inaccessible Resources**: Sensitive files or root owned files/directories (e.g., `/etc/shadow` or `/root`) are not readable and writable by an unprivileged process.
- **Privileged System calls**: There are system calls specific privileges (e.g., `mount` and `chroot`). If an unprivileged process calls them, the request will be denied and failed.
- **Kernel/Other Processes' Memory**: An unprivileged user process is not allowed to read/write kernel memory. Other processes' memory is also not accessible (i.e., cannot read/write) due to the memory space isolation.

We define the three states with privileged resources.
**Unexploited State.** An execution under this state follows the permission configured for each resource. For example,

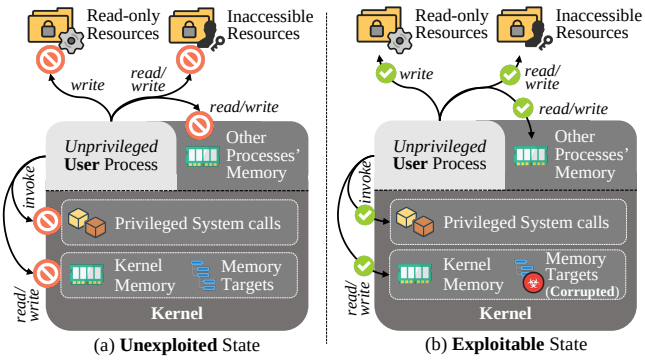


Figure 2: Unexploited and Exploited States

as shown in Figure 2-(a), it cannot read or write inaccessible files/folders (e.g., `/etc/shadow` and `/root`) and cannot write read-only files such as `/etc/passwd`, without calling permission/privilege changing APIs. Kernel memory, privileged system calls, and memory of other processes are inaccessible.
**Exploitable State.** Suppose a prohibited operation (e.g., read-/write) on any of the privileged resources in the unexploited state becomes available *after corrupting a memory target*. In that case, we consider that the execution's state has changed to an *exploitable state*. Note that there exist multiple instances of different exploitable states depending on which resource's privilege is escalated. For example, an exploitable state can have escalated privilege on `/etc/passwd` while another exploitable state provides access to a prohibited memory area. Hence, depending on the exploit's ultimate goal and its environment, a particular exploitable state would be needed. To this end, describing the exact *escalated privilege* and *prerequisite conditions* of an exploitable state is important to determine whether it can be used to achieve the exploit's goal.
**Exploited State.** An exploitable state becomes the *exploited* state, if the exploit achieves its ultimate goal with the escalated privileges (e.g., adding a root user with the escalated `/etc/password` and `/etc/shadow`). Note that the definition of the exploited state is *specific to each exploit's goal*. For example, to change configurations of Apache, an exploitable state execution with write access to `httpd.conf` is required.

## 2.2 Goal and non-goals

**Goal.** SCAVY aims to find a memory target that can change an unexploited state to an *exploitable state* when the target is corrupted. To clearly define the memory target's applicability, it is important to identify (1) the prerequisite of corrupting the memory targets and (2) the post-conditions (or escalated privileges) of the corrupted memory targets. In particular, the prerequisites and post-conditions are defined as follows.

- **Prerequisites**: (1) Required privilege and method for allocating the memory target, (2) Privilege required for corrupting the memory target (e.g., memory write permission if the target is read-only), and (3) Memory corruption capability (i.e., corruptible memory location, size,

and possible values) for the memory target.
- **Post-conditions**: (1) Escalated privilege/permission, (2) Name or path of the escalated resource, and (3) Area (i.e., offset and the range) of the escalated resource.

**Non-goals.** SCAVY does not focus on identifying new vulnerabilities. SCAVY's new memory targets may allow us, in some cases, to repair exploits that have been rendered inoperable by system-level defenses (e.g., SMAP), but they do not improve the exploits' *capability* or *reliability* (as defined in Section 4). While we demonstrate the ability to transit from an *exploitable* to an *exploited* state by developing a few functional exploits (see Section 3), automating this step is out of our scope. Finally, our goal is not to construct an automated *end-to-end* exploitation tool.

## 3 Motivating Examples

We describe two exploits using memory targets identified by SCAVY, to motivate its impact and practicality.

### 3.1 Corrupting `vm_area_struct::vm_file`

**Target Kernel Structure.** The `mmap` system call creates a memory-mapped file with an opened file descriptor. It will create a memory buffer containing the file content, and if it is created with the write permission, changes to the buffer will be written back to the original file, when the file is closed.

The Linux kernel uses the `vm_area_struct` structure, shown in Listing 1, for the memory-mapped files. The structure contains the address range of the mapped memory (`vm_start` and `vm_end`), its permission (`vm_page_prot`), and a pointer to the file (`vm_file`).

```
struct vm_area_struct {
    unsigned long vm_start;     /* mmap() retval. */
    unsigned long vm_end;
    ...
    pgprot_t      vm_page_prot; /* pg. permissions */
    ...
    struct file*  vm_file;      /* victim field */
    ...
};
```

Listing 1: Declaration of `vm_area_struct`.

**Discovery.** SCAVY automatically discovers that corrupting `vm_file` can impact the content of the corresponding memory-mapped file, potentially causing privilege escalation. Specifically, SCAVY first creates two executions access the same file (so that they will access `vm_area_struct::vm_file`). Then, it corrupts the `vm_file` with a random value in one of the executions. SCAVY compares the two executions (i.e., with and without the corruption) to discover the two executions obtain different contents of the file.

Next, SCAVY creates the third execution to check whether it can achieve privilege escalation. This time, instead of a random value, SCAVY uses other instances of valid values of `vm_area_struct::vm_file`. Specifically, SCAVY opens a set of both privileged and unprivileged files to obtain the values of `vm_file` instances. SCAVY copies the values to corrupt the `vm_file` and checks whether the third execution can access contents of any files that their `vm_file` values were copied.

This checks whether copying the content of `vm_file` from a privileged file to adversary-owned file would allow the adversary to *access the privileged file's content* using the adversary-owned file's permission. For example, if an adversary copies the `vm_area_struct::vm_file` of the password file (i.e., `/etc/passwd`) to an unprivileged temporary file's `vm_area_struct::vm_file` (e.g., `/tmp/file`), the adversary can read and write the password file. To this end, SCAVY identifies `vm_area_struct::vm_file` as a memory target.

**Completing the Exploit.** We use `CVE-2022-27666`, which provides an out-of-bounds write capability [28], to corrupt the memory target. The exploit first opens two files: (1) a dummy file with a read/write permission and (2) the password file (`/etc/passwd`) with a read permission. Then, it creates a few thousand memory maps of the files using `mmap` (e.g., 3,000 times in this example). Note that each `mmap` call results in allocating an instance of `vm_area_struct` in kernel. We then leverage the vulnerability to leak *neighboring pages* of the created structures to read the `vm_file` that maps `/etc/passwd`. Next, we use the vulnerability again to copy the content of `vm_file` of the `/etc/passwd` into the dummy file's `vm_file`.

Finally, the exploit calls `msync` with the corrupted `vm_area_struct::vm_start`. This makes the kernel synchronize the mapped page with the content of `/etc/passwd`, using the permissions of the dummy file which is the read and write permissions. To this end, the attacker can add a new root-level account by modifying the `/etc/passwd`.

Appendix A.2 presents more details of this exploit, along with the option of using `file::f_mapping`, another memory target discovered by SCAVY, to exploit `CVE-2022-27666`. In Section 7.3.3, we discuss the exploitation of two additional vulnerabilities corrupting `vm_area_struct::vm_file`.

### 3.2 Corrupting `key::description`

**Target Kernel Structure.** The `keyctl_instantiate` system call allocates the `key` structure in the kernel, shown in Listing 2. It contains fields to store permissions (`perm`), the owner's identifiers (`uid` and `gid`), and a text description of the key (`description`). This time, we target `description`, which is a string pointer where its text is used by the `keyctl_search` system call which allows a user to search a keyring.

```
struct key {
    refcount_t       usage;
    ...
    kuid_t           uid;
    kgid_t           gid;
    key_perm_t       perm;
    ...
```

```
    unsigned long    len_desc;
    char*            description;
    ...
};
```

Listing 2: Declaration of `key`.

**Discovery.** SCAVY discovers that once the `description` field is corrupted, an attacker can call `keyctl_describe` to read a string value from the corrupted address, allowing an arbitrary memory read. Specifically, SCAVY creates two processes which call (1) `add_key()` with a crafted description including payload to create a key and (2) `keyctl_read()` to access the created key. In one process, SCAVY injects a corruption that overwrites `key::description` with a value from another instance of `key`. The other process runs without any memory corruption. At the end of both executions, SCAVY compares the return buffers of `keyctl_read()` to detect the deviation, discovering a privilege escalation on the kernel memory, which is inaccessible to unprivileged user processes.

**Completing the Exploit.** We modified an existing exploit for `CVE-2016-0728`, that overwrites `key::key_type::revoke`. The field is a function pointer; thus it allows an adversary to hijack the control flow. However, in practice, this made the exploit unreliable in Android devices, either due to lack of kernel symbols [29] or SMEP/SMAP [30].

Instead of corrupting the function pointer, we focus on `key::description`. As with the original, our exploit first triggers the vulnerability by overflowing `key::usage` to '0', which is a reference count. This frees the structure prematurely and gives us a use-after-free primitive. We use this primitive by allocating a '`msg_msg`' object, which includes a buffer where we can insert arbitrary data (a message). The buffer overlaps with the fields of the `key` object and allows us to overwrite the length (`len_desc`) and the description pointer (`description`), which is the target. If done properly, this allows an adversary to read from arbitrary kernel addresses. With this capability, the adversary can bypass KASLR, leak kernel memory, and read secret keys from other Android apps' keyrings, potentially leaking session cookies.

## 4 SCAVY in the Kernel Exploitation Development Pipeline

In this section, we contextualize our work with respect to the prior research on the Linux kernel exploitation development. As shown in Figure 3, a functional kernel exploit is created through *four stages*: ❶ Identifying a vulnerability causing a crash, ❷ Discovering the vulnerability's other capabilities that can corrupt various memory targets, ❸ Combining the capabilities to escalate privilege, and ❹ Creating a reliable exploit escalating privilege while bypassing defenses.

– **Stage ❶**. Vulnerability Discovery.

**Fuzz Testing Approaches.** To find a vulnerability that can change a system's behavior, various testing techniques, such as fuzzing, have been proposed. Syzkaller [31] is a popular fuzzer that is specialized for finding Linux kernel vulnerabilities. Recently, various advanced fuzzers have been proposed, including those leveraging hybrid fuzzing [32], symbolic execution [33], and state-based exploration [34–36] to improve the effectiveness of testing for vulnerability discovery.

**Exploiting Violation Detectors.** Violation detectors [37–41] are runtime techniques that raise exceptions when they detect operations violating desired properties (e.g., out-of-bounds reads/writes [37], use-after-free [38], and race conditions [39,40]). When they are applied to the target system, they essentially turn the violations into crashes, helping fuzzers identify the violations that can be a strong indicator of potential vulnerabilities. Recently, leveraging such detectors has become a typical tactic in the Linux kernel fuzzing [41].

**Relevance to SCAVY.** An exploit requires a vulnerability that can corrupt a specific memory, which can lead to privilege escalation. A vulnerability in this stage typically causes a crash, often because it corrupts a critical memory. However, it is unclear whether it can corrupt a specific memory target with a desired value. SCAVY focuses on discovering memory targets, but not finding the vulnerabilities.

> In Section 3.1, this step is equivalent to choosing a known vulnerability (i.e., `CVE-2022-27666`).

– **Stage ❷**. Capability Discovery.

**Capability of a Vulnerability.** A vulnerability is often released with a *single memory corruption target*, which typically causes a kernel crash when triggered. However, such an initial capability may not be sufficiently powerful and versatile enough to achieve privilege escalations. As a result, investigating a vulnerability's other capabilities has become a critical step to see if the vulnerability can be exploitable. A line of research exists that searches a vulnerability's full capabilities [13–15]. Specifically, [13] introduced capability-guided fuzzing to investigate out-of-bounds write vulnerabilities. [14] leverages fuzzing and symbolic execution to explore various contexts of a use-after-free (UAF) vulnerability and determine if the attacker can control the system to reach an exploitable state. [15] explores multiple crashes of the same bug in an effort to observe more exploitable crashes. AlphaExp [42] discovers fields of kernel structures that can be exploited to achieve arbitrary code execution (ACE) or arbitrary address writing (AAW) capabilities but not privilege escalation. It focuses on a different set of structure fields, missing structures such as `vm_area_struct` that SCAVY found.

**Relevance to SCAVY.** This stage explores whether the vulnerability can corrupt a more diverse memory range. As SCAVY discovers memory targets which are essentially kernel data structures' fields, capabilities of are vulnerability is critical to see whether it can be used to corrupt the SCAVY's memory targets. To use a SCAVY's memory target, a vulnerability should have a capability that can corrupt the memory target.

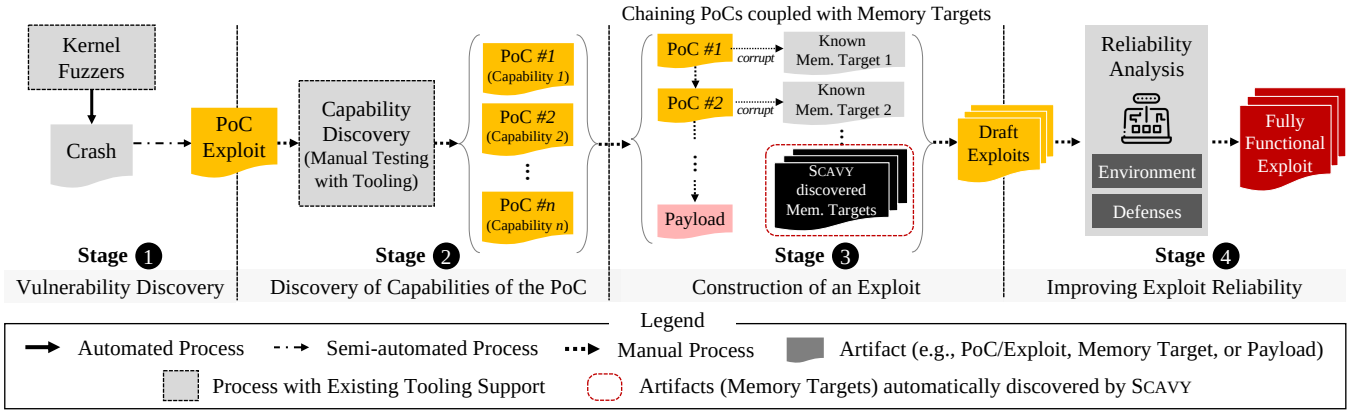


Figure 3: SCAVY in the Linux kernel exploit development pipeline

In Section 3.1, the vulnerability's exploit *already has the capability* of corrupting the memory target.

– **Stage ❸**. Construction an Exploit.

**Escalating Privilege.** Privilege escalation is typically achieved either (1) by gaining *arbitrary code execution* or (2) by *elevating privilege of the current user/resource* to root.

First, existing approaches [13, 14, 43] automatically test whether a vulnerability can corrupt a known field of data structures that can hijack the control flow (e.g., function/data pointers). Second, existing scripts [44] search for fields of kernel structures related to known critical system configurations (e.g., `uid`, `gid`, or credentials [5]) that can lead to privilege escalation if corrupted. However, they focus on *known fields* or *limited types of fields* (e.g., code or data pointers [13,14,43] and *reference counter* [45–47]). While approaches searching for other types of memory targets [45–49] exist, they are either manually done or focusing on certain types. For example, [48] focuses on structures handling variable-length data, which can allow out-of-bound reads if corrupted and [49] looks for structures with data pointers, which allows adversaries to control the referenced data if corrupted.

Some exploits require *chaining multiple vulnerabilities (or PoCs)* to corrupt multiple memory targets (e.g., leaking a value first and corrupting a field with the value), achieving privilege escalation. Finally, a payload is followed to achieve the exploit's ultimate goal (e.g., taking over the system).

**Relevance to SCAVY.** Typically, only the vulnerabilities capable of corrupting *a few known memory targets* are used to construct an exploit. Vulnerabilities that can corrupt other memory targets but not those known ones were considered usable. SCAVY discovers new memory targets, enabling more vulnerabilities to be usable for an exploit. SCAVY found memory targets allow an exploit to achieve privilege escalation in more diverse and subtle ways.

In Section 3.1, the vulnerability is used to corrupt a new SCAVY found memory target, `vm_area_struct::vm_file`, achieving the privilege escalation of a single file, instead of escalating an entire process's privilege [50]. Note that we chain two capabilities to first *read* the value of `vm_file` and then *write* the leaked value to another `vm_file` field.

– **Stage ❹**. Improving Exploit Reliability.

**Consistently Corrupting Memory Targets.** While an exploit is created in the previous stage, it might not reliably achieve privilege escalation, due to the randomness of memory layout and timing during the execution of the exploit. Hence, exploit authors leverage various techniques [43, 51, 52] to achieve an environment for reliable memory corruption. Specifically, [52] shows a method called *heap feng-shui* that extends to all slab caches and thus can be recycled for multiple kernel exploits. [43] automatically finds system calls to allocate objects of interest in a desired heap memory layout. [51] tests commonly known heap layout manipulation and exploit stabilization methods to aid the exploit development.

**Bypassing Defenses.** A reliable exploit should also evade existing defenses [10, 53, 54]. To this end, researchers have proposed various techniques to bypass the defenses. Specifically, [55] chains kernel-side ROP gadgets to bypass modern control flow integrity-based protections (e.g., CFI [10]). [56, 57] present methods to bypass the Kernel Address Space Layout Randomization (KASLR) [53]. [54] and [58] have proposed a method to bypass Supervisor Mode Execution Prevention (SMEP) [59] and SMAP (Supervisor Mode Access Prevention), respectively. [50] presents practical tricks to improve the reliability of an exploit.

**Relevance to SCAVY.** While SCAVY is not directly relevant to this stage, using some SCAVY found memory targets makes it easier to achieve reliability. For example, an analysis of the published PoC exploit for `CVE-2016-0728` mentions that existing defenses such as SMEP/SMAP [9, 59] can throttle the success rate of the exploit [30], where some SCAVY found memory targets can avoid the detection of those defenses.

In Section 3.1, SCAVY found memory target can achieve privilege escalation without violating the integrity of code pointers, without requiring a defense bypass.

# 5 Design

Figure 4 shows an overall procedure of SCAVY, consisting of three phases: (1) Instrumentation and Analysis (Section 5.1), (2) Discovery of Potential Memory Corruption Targets (Section 5.2), and (3) Detection of Memory Corruption Targets for Privilege Escalation (Section 5.3).

## 5.1 Instrumentation and Analysis

**Type Casting Instrumentation.** To identify memory corruption targets, SCAVY needs to identify types of allocated memory (e.g., types of structures) in the kernel and corrupt them. While previous techniques such as GREBE [15] rely on expensive taint analysis, they cannot be used for the sheer number of potential memory corruption targets SCAVY deals with. To this end, SCAVY instruments type casting operations.

Note that the Linux kernel coding-style guideline document [60] indicates that a memory allocation should be type-casted. In LLVM, it uses `CastInst` [61] for the type casting operation. Hence, we instrument `CastInst` by inserting a `call` to a dummy function that takes two parameters: (1) the memory address of the variable that is being type-casted, and (2) its new data type, passed as a string at compile-time. Note that we only instrument `CastInst` when its source and destination types are different (e.g., '`void*`' is assigned to a structure). At runtime, we use Kprobe [62] to log the dummy function's parameters to user space along with the return value of memory allocators (e.g., `kmalloc()`). Later, the fuzzer associates allocated memory addresses with their data type.

**Analysis for Kernel Data Structures.** During the instrumentation and analysis process, SCAVY aims to extract memory layouts of kernel structures of interest, so that it can guide the subsequent analyses of SCAVY. While we can extract them from the source code via an LLVM pass, they may not be accurate if a compiler optimizes the memory layouts of the structures. Specifically, structures that are not packed (i.e., structures without the '`((packed))`' attribute) may have unused memory space between fields, making the offsets of structures' fields in a binary different from the structure's definition in the source code. For example, if a structure has a 6 bytes of character array followed by an integer (i.e., `struct { char s[6]; int n; }`), a compiler may insert two unused bytes between the two fields (i.e., after `s[6]` and before `n`, resulting in `struct { char s[6]; char unused[2]; int n; }`). Hence, we use `pahole` [63] to find the sizes and offsets of structures at the binary level. We use construct a lookup dictionary mapping between a structure's name and the structure's fields' offsets, sizes, and types. Note that structures inside a structure are resolved recursively.

**Memory Corruption Bridge.** Most of SCAVY's components run in user mode, which do not have direct access to the kernel memory. Hence, we implement a kernel module that allows SCAVY's user mode components to corrupt the kernel memory. Specifically, the kernel module exposes an interface via `ioctl()`, providing read and write capabilities of the kernel memory to the user mode programs. During the instrumentation process, we include the bridge module into the kernel.

## 5.2 Discovery of Potential Memory Targets

**Three Stage Operations of Exploits.** We observe that most exploits in practice exhibit a pattern of three distinctive operations: (1) *allocating* a resource associated with a memory target, (2) *corrupting* the memory target to obtain sufficient permissions for privilege escalations, and (3) *conducting privileged actions* or actions escalating privilege.

Figure 5-(a) shows the exploit described in Section 3. It first creates two files. The memory targets are associated with the files (❶). Then, it corrupts the memory target, which is a kernel structure storing information of the file via the vulnerability (❷). This allows an adversary to obtain the permission of the root-owned file '`/etc/passwd`.' It then adds a new user by writing the '`/etc/passwd`' file, escalating privilege (❸). Figure 5-(b) follows a similar procedure. It creates a process that allocates the memory target `task_struct` (❶), and then corrupts the '`task_struct::euid`' associated with the created process, changing the process's effective user to the root user (❷). Finally, it creates a privileged shell via `execve()`.

**Search Space for Each Stage.** As each stage conducts a different operation, candidate operations for each stage may exist in a different space, requiring a distinctive focus. Specifically, to search operations for the first stage, one may look for system calls that *allocate* memory buffers, regardless of whether they will access the buffers or not. For the second stage, one should focus on the contents of the memory buffers and the impact of the corruption on the system. For the last stage, we should focus on the system calls dependent on the corrupted memory (i.e., system calls using the corrupted data).

Due to, in part, the different focus on each stage, in practice, different tools are used. For example, Syzkaller [31] is effective for searching the first stage because it aims to achieve higher code coverage. Intuitively, covering more code would exercise more program paths that may allocate data structures. For the second stage, KOOBE [13] and FUZE [14] are effective as they implement heuristics for detecting memory corruptions, such as overflowing into other co-allocated objects. Unfortunately, there are no popular tools for the third stage as it is manually done in practice.

**SCAVY's Approach.** Unlike existing tools that are only effective for each stage, SCAVY proposes to apply different criteria and metrics for each stage dynamically. Specifically, for the first stage, we use code coverage of system calls as a metric. Intuitively, by covering most of the code, it might also

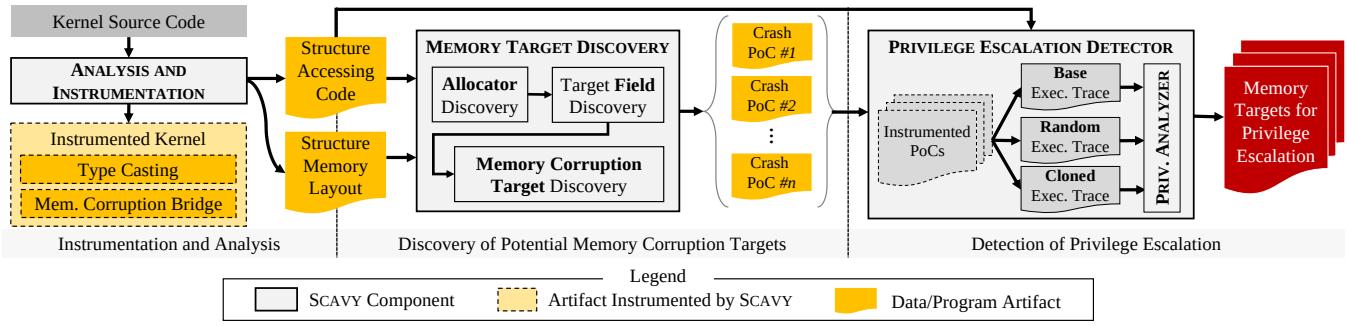


Figure 4: Overall SCAVY Design

```
1  for (int i = 0; i < 100000; i++)
2      open("/etc/passwd", O_RDONLY);
3  int wf = open("/tmp/writablefile.txt", O_RDWR);
4
5  CORRUPT_FILE_MAPPING(/* SPRAYED OBJ. MAPPING */);
6
7  write(wf, "root2:x:0:0:root2:/:/bin/bash\n", 32);
8  close(wf);
```

(a) Exploit that Corrupts File Mapping

```
9  int pid = fork();
10 if (pid == 0) {
11     CORRUPT_euid(0);
12     exit(0);
13 } else {
14     waitpid(pid);
15     setuid(0);
16     execve("/bin/sh");
17 }
```

(b) Exploit that Corrupts **euid**

Figure 5: Exploits Consisting of Three Steps

cover all the kernel structure allocation code of interest. For the second stage, we use coverage of instructions that *load* the corrupted memory to identify whether the corruption has an impact on the system or not. In the third stage, we focus on code coverage of the code dependent on the corrupted memory, to explore diverse consequences of the corruption.

### 5.2.1 Allocator Discovery

**Objectives.** This stage aims to identify a list of system calls that allocate kernel structures, which are the memory corruption targets. We aim to find system calls allocating as many unique kernel structures as possible because all the subsequent analyses are limited to the result of this stage. SCAVY runs every system call, tracks accesses to all allocated memory buffers, and detects types of allocated kernel structures.

**Tracking Allocated Structures.** We track memory allocations and memory accesses to identify allocated kernel structures as follows. We use Kprobe to set breakpoints on `kmalloc()`[2] and `kmem_cache_alloc()` to capture their return values (i.e., base addresses of allocated kernel structures).[3] We also set a breakpoint on the dummy function in Section 5.1 to capture its arguments. SCAVY then processes the Kprobe's output to associate addresses from typecasting operations with their allocation addresses. Note that analyzing type casting (and not using expensive taint/symbolic analysis [15]) is a key design choice that makes SCAVY scalable, allowing us to conduct the analysis on millions of generated sequences.

**Coverage Guided Searching.** SCAVY tries to cover more code allocating kernel structures, aiming to discover allocators for diverse memory targets. We use code coverage, *without changing the fuzzer and the coverage*, as, intuitively, covering more invocations of `kmalloc()` and `kmem_cache_alloc()` would likely find allocators for various memory targets.

### 5.2.2 Memory Target (Structure Field) Discovery

**Objectives.** Corrupting certain parts of the kernel structures change the system's behavior, while other parts may not have an observable impact on the system. Hence, SCAVY aims to find which fields of the structures can be memory targets causing an observable impact. Specifically, among all the allocated structures in the first stage, we search, one field at a time, which fields of structures can impact the system when corrupted (e.g., a crash). Similar to Section 5.2.1, we search conservatively, as the subsequent analyses depend on it.

**Conservative Target Searching.** We use a *conservative* definition that *if a corrupted field of a kernel structure is read at least once, it may impact the system*, meaning that it is a potential memory corruption target. Specifically, we corrupt each 4-8 byte field of the kernel structures with random bytes and use the `hardware watchpoint` [64] to monitor instructions that load the corrupted field. If we identify any such *load* instructions, the field is considered a potential memory target, and SCAVY moves on to the search for the next field. The `watchpoint` is lightweight and reports sufficient information on the memory access (i.e., whether the field was accessed).

### 5.2.3 Memory Target Discovery

**Objectives.** With the potential memory target for structures, SCAVY uses a fuzzer to execute various system calls that may access corrupted memory and cause a crash. In this stage, we *prioritize* code that *accesses the corrupted memory* (i.e., the

[2] While it is `__kmalloc()`, we simply call it `kmalloc()` hereafter.

[3] We track `kmalloc()` and `kmem_cache_alloc()` in the process of interest.

corrupted field of structures). However, during fuzzing, there can be crashes (i.e., kernel panic) on the first access of the corrupted memory, which we call *premature crashes*. Those crashes may prevent the fuzzer from exploring the full impact of the corruption, missing later corrupted memory accesses.

**Focused Fuzzing on Relevant Code.** To facilitate the search in this stage, we make two adjustments on our fuzzing process. First, we make it focus on the code that is accessing the kernel structures of interest. Specifically, we make our fuzzer only accept coverage from functions accessing structures.

Second, we prioritize the instructions loading the corrupted memory by detecting executions running the instructions and seed them. Specifically, recall that SCAVY collects information about the code related to the structures of interest in the instrumentation and analysis phase, as shown in Figure 4 (Structure Access Code). SCAVY uses hardware breakpoints[4] to detect those instructions and make them seed.

**Avoiding Premature Crashes.** We propose an approach that executes a generated program call twice to avoid premature crashes. In the first execution, we run it *without corrupting* the memory target. The fuzzer collects the code coverage accordingly and re-executes it by corrupting the memory target with a random value. If the execution crashes or we see a deviation in the code coverage (indicating potential privilege de-escalation), we log the execution to be analyzed later.

## 5.3 Detection of Privilege Escalation

The outcome of Section 5.2 is a list of programs causing crashes, some of which may cause privilege escalations. In this phase, we aim to detect which of them from the previous phase are causing privilege escalation. Specifically, SCAVY uses a differential-analysis-based approach to detect privilege escalation as follows. First, we run a given program thrice: one run without corruption, one with corruption using a random value, and another with corruption copying the value of the memory target from other valid structures created by a privileged process. Second, all three runs will execute privileged operations (e.g., operations requiring root permission) such as `setuid(0)` and `read`/`write` on privileged resources. If the first and the third runs have differences, we consider it may have escalated or de-escalated privilege.

### 5.3.1 Inserting Privilege Dependent Operations

**Read and Write System Calls.** For each PoC, we insert read and write system calls for all the resources (e.g., files and sockets) created during the PoC, as their behaviors will be different if privilege escalation happened. For example, as shown in Figure 5-(a), we add read and write system calls for all the files (i.e., '`/etc/passwd`' and '`/tmp/writablefile.txt`').

[4] Since hardware breakpoints require addresses of the instructions, we use `addr2line` [65] to obtain the corresponding instruction addresses.

**Privileged Operations.** As shown in Figure 5-(b), privileged operations behave differently based on the current privilege. We add privileged operations (i.e., root) such as `seteuid(0)` and `setegid(0)`. `sync()`, `msync()`, and `fsync()` are added for all the resources created before the memory corruption. We provide a list of all the added syscalls in Table 7.

### 5.3.2 Detecting Exploitable States

**Three Executions for Privilege Escalation Testing.** SCAVY runs three executions to detect a potential privilege escalation. First, we run the exploit on an unprivileged system without memory corruption. We expect all the privileged operations to fail and unprivileged operations to succeed. Second, we run the exploit on the same system with memory corruption using a random value. If privileged/unprivileged operations behave differently from the first execution, we consider it to have reached an exploitable state. Third, we run the exploit on the unprivileged system with memory corruption using the memory contents of another instance of the same type kernel structure. Specifically, we borrow a value of a structure's field from a root process with the same structure. Again, any deviations from the first run suggest privilege (de-)escalation. Note that the random and cloned values are complementary. Cloning works well for complex kernel data structures with certain formats as they are difficult to generate randomly. However, cloned values limited to what the system created. For example, cloning `uid` will never result in an invalid/unknown `uid`, which a randomly generated value can.

**Detecting Exploitable States.** For each run, we collect 4 indicators: (1) the return value of system calls, (2) the system call's return buffer's (or read buffer's) data, (3) the addresses of instructions loaded the corrupted memory, and (4) the code coverage. We consider two executions to be the same if the following 4 conditions are satisfied: (1) return values of the system calls are identical, (2) return buffers' contents of the system calls are identical, (3) executed `load` instructions on the memory target are identical, and (4) Jaccard similarity [66] of the code coverage is greater than 0.9 (accounting for noise in Kcov). Note that we have three runs to compare (2 pairs to compare). We compare the run without corruption with the two other runs with different corruption methods. If the run with corruption is different with any of the comparisons are not the same, we consider it to reach an exploitable state.

For example, in Figure 5-(a), SCAVY first runs it without a memory corruption (line 5). The `write()` at line 7 will fail due to the missing privilege. In the second run, it corrupts the memory target (i.e., file mapping pointer) with a random value. The execution will crash or fail due to the invalid file mapping pointer. In the third run, it corrupts the memory target by cloning a valid file mapping pointer. Now, `write()` at line 7 will succeed and SCAVY detects differences in code coverage and `write()`'s returns. Similarly, in Figure 5-(b), executions have different coverages and `setuid()` return values.

## 6 Implementation

SCAVY is written in Go (2068 LoC), C (331 LoC), C++ (827 LoC), and Python (2199 LoC). We customized the Syzkaller in Go. In C, we wrote the in-tree kernel driver used by the fuzzer to corrupt and set up hardware tracing. We also developed 3 LLVM passes, two of which produce the input to the fuzzer and the third to instrument the kernel typecast. Our privilege escalation detector module is written in Python.

**Modifying `Syzkaller`.** We modify `syz-executor` to execute the same program twice, as mentioned in Section 5.2.3. For each execution, we collect data from `KCOV`, `Kprobe`, and `dmesg`. The original Syzkaller collects the code coverage from `KCOV`. Our modification allows it to collect the memory allocation, deallocation (i.e., `kfree()`), and typecasts from `Kprobe` and memory hits of the corrupted memory from `dmesg`. To collect data from `Kprobe`, our Syzkaller instructs `Kprobe` to set up its probe points for `kmalloc`, `kmem_cache_alloc`, `kfree`, `instrument_typecast_instruction`, and the instrumented dummy functions added after each typecast instruction. The fuzzer instructs `Kprobe` to print all parameters of the instrumented functions and `kfree`, including the return values of the memory allocators. The fuzzer spawn two threads in parallel to the fuzzing executions to (1) parse the output logs of `Kprobe` and filter out logs that are not generated by `syz-executor` and (2) parse the output of `dmesg`. To implement the focused fuzzing for memory corruption relevant code (Section 5.2.3), our fuzzer generates a large number of fake instruction pointers for each new corruption, making the fuzzer think that hitting an instruction accessing the corrupted memory leads to a significant coverage increase. This results in the fuzzer prioritizing the code and using it as a seed. We also disable the fork server, which makes it terminate after executing the requested sequence of system calls.

**Privilege Escalation Detector.** Our privilege escalation detector needs to reproduce the executions observed during the fuzzing, including the memory corruptions. To achieve that, we use `syz-prog2c` that generates a C program reproducing the fuzzing execution, provided by `Syzkaller` and add C statements that corrupts the target memory to the C program. Then, we implement a module that executes the C program (i.e., instrumented PoCs) with different mutation methods. Specifically, we wrote a C library (*827* LoC C++) that spawns 2 threads along with the original running PoC code: (1) a thread to set up `Kprobe` and a custom driver interacting code we wrote into `syz-fuzzer` and (2) another root user thread that synthetically opens the privileged resources. The first thread tracks the structures created by all the three processes.

## 7 Evaluation

We evaluate SCAVY from three aspects. First, we measure the coverage of the typecast instructions with respect to all the Linux kernel structures to evaluate the effectiveness of our typecast instrumentation[5]. Second, we evaluate the effectiveness of the fuzzer in exploring crashes that can lead to exploitable states. Third, we analyze the detection results of the differential analysis module and evaluate their reachability on real world exploits. We also compare our exploitable state definition with FUZE's to show that our broader definition helps discover new memory targets. All experiments were done on a system running Ubuntu 20.04 LTS with an 3.70 GHz Intel Xeon E3-1245 v6 and 32 GB of RAM.

### 7.1 Typecast Coverage

SCAVY statically instrumented 2,313 source code files (85% of the 2,700). 23% (2,130 out of 9,169) of the kernel data structures are instrumented[6] to track their types after their allocation at runtime. While analyzing the low coverage result, we find many structures that are covered are irrelevant to us. First, there are 2,587 structures for debugging purposes such as `kprobe_insn_cache` and `trace_event_raw_kfree` are used by Kprobe and Ftrace and are mostly unavailable on production kernels. Second, there are 1,989 stack-allocated structures that are destroyed on function return (e.g., `msg_receiver` and `msg_sender` in `do_msgrcv()` and `do_msgsnd()`). Those short-lived structures typically do not hold critical system states and, hence, are not our target. Third, 1,931 structures are members of our instrumented structures, which we can consider instrumented. For example `signal_struct` is contained within `sigpending` – there is no code that directly allocates it. Fourth, many of the remaining structures are processor-related global variables (e.g., `intel_watermark_params`) and architecture-specific structures (i.e., defined in `arch`) that we do not instrument or helper structures for accessing their other structures (e.g., `xfrm_skb_cb` for accessing `sk_buff->cb`).

To this end, we prune out 4,576 from 9,169 and additionally consider 1,931 structures to be covered, resulting in a new coverage of 88.4% (4,061/4,593). Our coverage can be interpreted as a lower bound of what our fuzzer can corrupt.

### 7.2 Fuzzer Effectiveness

We evaluate the effectiveness of our fuzzer, compared to the stock version (commit: 61f86278) of Syzkaller. Specifically, we measure the coverage of the code that is relevant to structures (e.g., allocating and accessing the structures of interest).

For a fair comparison, we improve the stock version of Syzkaller by adding hardware tracking of memory reads, our object allocation implementation, and corrupted memory access tracking. We run both fuzzers for 3 days on the identical QEMU VM (with 1 core and 1 GB RAM).

[5] If a typecast instruction is not instrumented, SCAVY may not analyze the specific use of kernel structure, possibly missing a memory target.

[6] We noticed that instrumenting certain boot files caused the kernel to crash. Hence, we restrict the instrumentation to directories other than '`arch`', '`boot`', '`kernel/panic`', and '`signal`'.

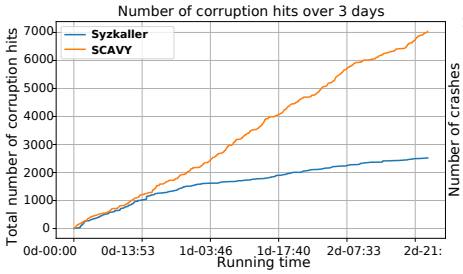


Figure 6: # of Observed Loading of the Corrupted Fields

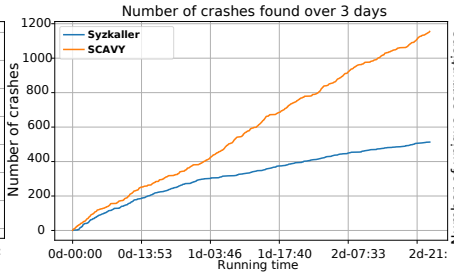


Figure 7: # of Crashes Observed

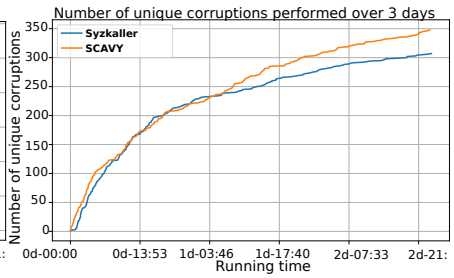


Figure 8: Comparison of the Number of Unique Fields Corrupted over Time

**Code Coverage.** Figure 6 shows the total number of read operations on any corrupted memory during the 3 days of fuzzing. This number indicates the number of *instructions loading the corrupted memory* executed during fuzzing. The graph shows that for the first 14 hours, the 2 fuzzers perform similarly, but the stock fuzzer seems to hit the corruption at a slower rate. When manually checking the corpus we find this happens because, in part, the stock fuzzer prioritizes exploring code paths, where some of them may not access the fields of structures of our interest. For example, the stock fuzzer may prefer to open a file and then immediately open a socket, instead of writing to the opened file, since opening a socket can lead to higher code coverage. In contrast, SCAVY prioritizes covering the code that is relevant to the data structures of interest (e.g., accessing the file in the example).

**# of Crashed Observed.** Figure 7 presents the number of crashes observed from the two fuzzers. While both fuzzers inject memory corruptions into the least represented structures, the stock Syzkaller does not prioritize hitting the memory it corrupts. This can be inferred by the lower number of crashes, suggesting the de-prioritization of corrupted memory accessing code while prioritizing the overall coverage. In contrast, our SCAVY fuzzer was able to generate more crashes as it prioritizes the instructions accessing corrupted memory. This is further evidence of the effectiveness of our approach, which prioritizes the exploration of the search space for exploitable states; focusing on the code coverage from source code that operates on the target structure type and adding synthetic code coverage based on code that loads corrupted memory.

**# of Corrupted Fields.** We measure the number of corrupted fields during the experiments to show that SCAVY covers diverse kernel data structures than state-of-the-art techniques. Specifically, we plot the number of unique fields of structures that are corrupted. Figure 8 shows that both fuzzers exhibit a similar rate of exploration in discovering new fields to corrupt.

While SCAVY focuses on a set of structures of interest, this graph suggests that our fuzzer does not become stuck corrupting limited types of structures. As described in Section 5.2.1, while SCAVY leverages redefined coverage that may focus on specific structures, our fuzzer still operates with some guidance from unfiltered code coverage, and as a result, it works well for both exploration and exploitation. Note that we repeat the 24-hours experiment 10 times, following the best evaluation practices for fuzzers [67]. We present the extended results in Section A.4. After proving distinction between the two distributions using the U-test [68], we conclude that the fuzzer without corruption-hit-guidance tends to cover more code, because, in part, they do not deal with crashes caused by the corruptions. However, the crashes caused by corruptions are potential privilege escalations detected by SCAVY, meaning that the original fuzzers' high code coverage is irrelevant to finding memory targets. In Section A.4 we show the effects of the modifications of the fuzzer on the code coverage.

## 7.3 Exploitable State Detection

To understand the impact of our new definition of exploitable state in finding memory targets, we conduct two evaluations. First, we compare the scope of our exploitable state definition to the commonly studied ones, such as write-what-where and control flow hijacking. Second, we demonstrate the efficacy of SCAVY by presenting its ability to discover both previously known and new memory targets. To conduct this experiment, we ran the fuzzer for 7 days. We identified 2,811 unique fields from 139 distinct kernel structures that could be exploited. The fuzzer generated a total of 3,863 PoCs whose corresponding object corruptions crashed the kernel.

### 7.3.1 Evaluation of the Exploitable State Definition

We run FUZE's symbolic executor and SCAVY to detect the exploitable states for all the PoC. To avoid the FUZE being stuck in an infinite loop, we disabled around 700K function call sites corresponding to KCOV, Ftrace, our custom driver, and part of KASAN. For a fair comparison, we used a 5-minute timeout (FUZE uses the same timeout). We then ran the same PoCs through our privilege escalation detector step.

FUZE found 354 PoCs that lead to at least one exploitable state, meaning that during the unconstrained symbolic execution, more than one value satisfies the branches taken to reach an exploitable state. The exploitable states FUZE found are mostly attacker-controlled write with a symbolic destination address. We show the number of PoCs that FUZE detected to reach at least 1 exploitable state in Table 3 (in Appendix).

| Structure::field | Prerequisites | Post. | CVE | Structure::field | Prerequisites | Post. | CVE |
|---|---|---|---|---|---|---|---|
| file::f_mapping | RW, $C_S$, 216, 8, p | RW, (𝔽) | ✷♠♣ | pipe_inode_info::bufs | W, $C_G$, 152, 8, p | R, ($\mathbb{M}_v$) | ♦ |
| ✪ task_struct::cred | RW, $C_S$, 1712, 8, p | X, (𝕊) | ♦ | kioctx::aio_ring_file | RW, $C_S$, 512, 8, p | R, ($\mathbb{M}_v$) | ✷ |
| task_struct::mm | RW, $C_S$, 1080, 8, p | R, ($\mathbb{M}_p$) | ♦ | kioctx::internal_pages | RW, $C_S$, 448, 8, p | R, ($\mathbb{M}_v$) | ♣ |
| task_struct::active_mm | RW, $C_S$, 1088, 8, p | R, ($\mathbb{M}_p$) | ♦ | aio_kiocb::ki_filp | RW, $C_S$, 0, 8, p | RW, ($\mathbb{M}_v$) | — |
| task_struct::vma_cache | RW, $C_S$, 1104, 8, p | R, ($\mathbb{M}_p$) | ♦ | key::user | RW, $C_S$, 72, 8, p | RW, ($\mathbb{M}_v$) | ✷ |
| address_space::i_pages | RW, $C_G$, 8, 8, p | W, (𝔽) | ✷ | key::description | W, $C_S$, 32, 8, p | R, ($\mathbb{M}_k$) | ✣✷ |
| vm_area_struct::vm_file | RW, $C_S$, 160, 8, p | RW, ($\mathbb{M}_v$) | ✷♠♥ | key::perm | W, $C_S$, 112, 8, 3«25 | R, ($\mathbb{M}_v$) | ✷ |
| inode::i_uid | W, $C_S$, 4, 4, 0 | W, (𝔽) | ♣♠ | shmem_inode_info::i_mapping | RW, $C_S$, 168, 8, p | R, ($\mathbb{M}_v$) | ✷ |
| inode::i_mapping | RW, $C_S$, 4, 4, p | RW, (𝔽) | ♣♠ | ✪ cred::cap_bset | W, $C_S$, 64, 8, $2^{21}$ | R, ($\mathbb{M}_v$) | ♦ |
| inode::i_pipe | RW, $C_S$, 568, 8, p | R, (𝕄) | ♣♠ | ✪ cred::euid | W, $C_S$, 20, 4, 0 | X, (𝕊) | ♦ |

Table 1: **Corruptions Found to Lead to Exploitable States.** Prerequisites are of the form: <exploit capabilities, generic/special slab cache,offset,size, value (kernel pointer or specific value)>. Post-conditions are of the form <observed capability, resource>. Resources are: Files(𝔽), Syscall (𝕊) and Memories of Process($\mathbb{M}_p$), Kernel($\mathbb{M}_k$) and other Virtual($\mathbb{M}_v$). ✪ indicates known field. We evaluate the exploitability of the corruptions using real-world CVEs marked in the table as: CVE-2010-2959 (♣), CVE-2014-3153 (♦), CVE-2016-0728 (✣), CVE-2017-7184 (♥), CVE-2017-7308 (♠), CVE-2022-27666 (✷).

Meanwhile, the detection step of SCAVY flagged *955* PoCs for having a deviation that can lead to an exploitable state. While our definition of exploitable state does not include control flow hijacking and write-what-where (i.e., it is not a superset of the prior work), the additional PoCs SCAVY found may imply the effectiveness of SCAVY's broader definition.

Among the 955 PoCs, many are corrupting the same field. Therefore, we decided to prioritize our manual analysis of the fields with the most positive detections and the ones we found within our expertise to analyze. In Table 1, we show 20 fields that were manually verified to lead to privileged operations and the exploitable state that was detected based on our definition in Section 2. The rest of the fields are presented in Table 6 in the Appendix. It is worth noting that the corruptions we report in this paper are *a subset of PoCs* we detected (in total, we detect 68). In Appendix A.1, we present details of the manual analysis we performed on the deviations detected by SCAVY. In the second column of Table 1, we summarize the capabilities required for the attacker to correctly corrupt the field, such as requiring both read and write or just write capability. For example, to exploit using `key::description`, the attacker needs to have the capability to **W**rite **8** bytes into the `key` cache at offset **32** the object with the value **0**. In the third column, we summarize the post-conditions, essentially escalated privileges, such as reading a pipe or writing a file. For example, after corrupting `key::description` with a valid kernel pointer, the attacker can read arbitrary kernel memory.

**Evaluating with Real world Vulnerabilities.** In the last column, we evaluate the reachability of SCAVY found memory targets using real-world exploits. Specifically, we first obtain exploits from KHeaps [51], as the authors provide vulnerable kernel environment that can reproduce their exploits. For each exploit, we first identify the part of the exploit that creates (or sprays) the victim kernel structure. Then, we check the exploit's memory corruption capability with the SCAVY's memory target's requirements. If the exploit has sufficient capability, we massage the memory layout [50] for our victim kernel structure (i.e., SCAVY's memory target) to be within the corruption range. Next, we replace the victim kernel structure creation (or spraying) code with SCAVY's PoC's code to allocate SCAVY's memory target. Finally, we trigger the vulnerability to corrupt SCAVY's memory target with an arbitrary value. After that, we replace the part accessing the corrupted memory in the original exploit with SCAVY's PoC's privilege escalation checking code, which invokes privilege-assessing system calls that eventually access corrupted memory and raise crashes if corrupted. If we get a crash with the corrupted value in its panic dump, we mark column 4 of this field with the respective CVE ID. We publish the modified exploits that can corrupt SCAVY targets in our repository.

#### 7.3.2 Evaluation of the Privilege Escalation Detector

Among the 955 detected behavior deviations, we find that code coverage is the most common deviation. Other significant deviations include accessing random addresses (when modifying data pointer fields) and privilege de-escalations (when modifying non-pointer fields). Table 4 (in Appendix) summarizes our results. We also analyze the common system calls where the deviations occur. Table 5 (in Section A.3) shows the top 10 system calls exhibit deviations, helping detect reaching exploitable states. Among them, we observe the `seteuid` system call that we additionally instrument, helping 16 cases out of 955 (1.5%). Others are mostly I/O related system calls (e.g., `pread64`, `ioctl`, `socketpair`, and `sendmsg`).

#### 7.3.3 Exploiting Real Vulnerabilities

We evaluate SCAVY found memory targets in terms of writing exploits with real-world vulnerabilities to illustrate its real-world impact and practicality. Our result is summarized in Table 2. Note that the vulnerabilities are from different software weaknesses, as denoted by the CWE (Common Weakness Enumeration), showing that SCAVY's memory targets do not

| | CVE | CWE | Exploited Struct/Field |
|---|---|---|---|
| 1 | 2017-7308 | Type Conversion | vm_area_struct::vm_file |
| 2 | 2017-7184 | Input Validation | vm_area_struct::vm_file |
| 3 | 2010-2959 | Int Overflow | kioctx::internal_pages |
| 4 | 2016-0728 | Use-After-Free | key::description |
| 5 | 2022-27666* | OOB Write | vm_area_struct::vm_file |
| 6 | 2022-27666* | OOB Write | file::f_mapping |
| 7 | 2009-3547 | nullptr deref. | pipe_inode_info::bufs |
| 8 | 2014-3153 | Input Validation | cred::euid |
| 9 | 2017-11176 | Use-After-Free | task_struct::cred |
| 10 | 2004-1235 | Race condition | file_struct::fops |

Table 2: Modified Real World Exploits (* indicates constructed fully functional exploit).

have specific requirements on the type of vulnerabilities.

We obtain the exploits from various sources. The first three exploits (1∼3) are from the original KHeaps [51] paper. The fourth exploit is borrowed from a public PoC[7]. Other exploits are from publicly available PoCs [50, 69]. For all the exploits except for the 2016-0728's exploit, we follow the procedure described in Section 7.3.1 to modify the exploits to corrupt SCAVY memory targets. The exploit for 2016-0728 is shown in Section 3.2. The exploits 5 and 6 are end-to-end exploits using 2022-27666 with SCAVY found memory targets. The entire exploits are presented in Section A.2. The exploits 7∼9, we found that the original exploits corrupt kernel structures that contains SCAVY's memory targets (in Table 1), while they target a different field to corrupt.

Note that we were unable to reproduce the 2004-1235 exploit as we could not reproduce the vulnerable environment (e.g., the kernel and OS); however, based on the developer email chain [70], we deduced that the exploit could control the content of a vm_area_struct and it had the vm_file field in the target kernel version (2.6), implying that the exploit can corrupt a SCAVY memory target. We release the exploits and vulnerable systems' VMs on [20].

## 8 Discussion

**SCAVY Facilitating Defenses.** There exist defenses for kernel data structure fields, such as XOR'ing their values, redzoning the fields [71], and applying write-once memory [72], while they cause significant overhead if applied to every kernel structure [6], Applying them only on SCAVY found memory targets would result in lower overhead, making them deployable.

**Limitations.** There exist a few sources of false positive in SCAVY's analysis. First, since SCAVY considers *any* differences between the executions with and without memory corruption as a potential privilege escalation if a divergent is caused by other reasons, such as a corrupted semaphore or lock fields causing an execution to hang and timeout, it would lead to a false positive. Second, if the device-specific pointers are corrupted, an execution may divert and be detected by SCAVY without escalating privilege, meaning it is a false positive case. For example, `bio::bi_private` is dedicated to device-specific structs and is usually `null`. When releasing, the kernel frees the pointer if it is not null, causing a deviation in code coverage. While SCAVY detects it, as per our definition, it does not lead to an exploitable state. Hence, it is considered a false positive. However, it is still a valid exploitation (while not a privilege escalation) as an attacker can corrupt this field to get an arbitrary-free capability. The `refcount` fields are also considered false positives as they do not escalate privilege but they can lead to valid use-after-free capabilities. Third, if it corrupts a file head, which maintains the file's current position for read and write, the file read API will return a different file portion, leading to different buffer contents. However, it does not escalate privilege. At last, we rely on a differential analysis method to detect privilege changes, meaning that SCAVY can only detect when system calls have *visible changes from userland* via the system calls in Table 7. For example, corrupting `addr_limit` to a larger value would not have immediately observable execution divergence, which will be missed by SCAVY.

**Responsible Disclosure.** We have responsibly informed and shared all the findings and artifacts (e.g., exploits) with the Linux maintainers and discussed its potential security consequences. We have their approval to disclose.

**Future Work.** Our work focuses on the impact of single-field corruption, while, in reality, an attacker may corrupt multiple fields. Also, SCAVY relies on manual analysis of the capabilities of prior exploits to provide a suitable target. We leave automating such tasks as a future work.

## 9 Conclusions

We designed and implemented SCAVY, which efficiently searches for memory targets that can lead to privilege escalation in the Linux kernel. Our design includes a new definition of the exploitable state for the Linux kernel in the context of local unprivileged attacks and differential analysis-based multi-execution reasoning to effectively detect potential privilege escalation. SCAVY discovered 17 memory targets in 12 kernel structures that can lead the system into an exploitable state when corrupted. We show the impact of SCAVY-found memory targets by demonstrating real-world PoCs of 9 CVEs corrupting the 17 memory targets.

## Acknowledgments

We thank the anonymous referees for their constructive feedback. The authors gratefully acknowledge the support of DARPA (Young Faculty Award), NSF (2427783 and 2426653), and an Amazon Research Award. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

[7] https://gist.github.com/PerceptionPointTeam/18b1e86d1c0f8531ff8f

## References


[1] D. Rosenberg. (2010) Interesting kernel exploit posted. [Online]. Available: https://lwn.net/Articles/419141/

[2] M. Stone. (2020) Cve-2019-2215: Android use-after-free in binder. [Online]. Available: https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html

[3] M. Brand. (2021) In-the-wild series: Android exploits. [Online]. Available: https://googleprojectzero.blogspot.com/2021/01/in-wild-series-android-exploits.html

[4] M. Rutland and C. Marinas. (2020) arm64: uaccess: remove set_fs(). [Online]. Available: https://github.com/torvalds/linux/commit/3d2403fd10a1db

[5] Z. Lin, Y. Wu, and X. Xing, "Dirtycred: Escalating privilege in linux kernel," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1963–1976.

[6] T. Yamauchi, Y. Akao, R. Yoshitani, Y. Nakamura, and M. Hashimoto, "Additional kernel observer: Privilege escalation attack prevention mechanism focusing on system call privilege changes," *Int. J. Inf. Secur.*, vol. 20, no. 4, 2021.

[7] K. Cook. (2017, Nov) security things in linux v4.14. [Online]. Available: https://outflux.net/blog/archives/2017/11/14/security-things-in-linux-v4-14/

[8] T. Garnier. (2016, Apr) mm: Slab freelist randomization. [Online]. Available: https://lwn.net/Articles/685047/

[9] J. Corbet. (2012, Sep) Supervisor mode access prevention. [Online]. Available: https://lwn.net/Articles/517475/

[10] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 179–194.

[11] K. Cook. (2017) Add slub free list pointer obfuscation. [Online]. Available: https://lists.openwall.net/linux-kernel/2017/07/07/546

[12] R. GONG. (2023) Randomized slab caches for kmalloc. [Online]. Available: https://lwn.net/Articles/938246/

[13] W. Chen, X. Zou, G. Li, and Z. Qian, "Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities," in *USENIX Security*, 2020, pp. 1093–1110.

[14] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *USENIX Security*, 2018, pp. 781–797.

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

[16] (2023) Corjail: From null byte overflow to docker escape exploiting `poll_list` objects in the linux kernel. [Online]. Available: https://syst3mfailure.io/corjail/

[17] V. Nikolenko. (2023) Cve-2016-6187: Exploiting linux kernel heap off-by-one. [Online]. Available: https://duasynt.com/blog/cve-2016-6187-heap-off-by-one-exploit

[18] A. Nguyen. (2023) Cve-2021-22555: Turning `\x00\x00` into `10000$`. [Online]. Available: https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html

[19] A. Konovalov. (2023) Cve-2017-1000112: Exploiting an out-of-bounds bug in the linux kernel ufo packets. [Online]. Available: https://xairy.io/articles/cve-2017-1000112

[20] Scavy github repo. [Online]. Available: https://github.com/BadDataLab/SCAVY

[21] "Microsoft resumes security updates with 'largest' patch tuesday release," Redmont Mag, 30 March 2017, https://redmondmag.com/articles/2017/03/14/march-2017-security-updates.aspx.

[22] "Massive microsoft patch tuesday security update for march," Qualys, 30 March 2017, https://blog.qualys.com/laws-of-vulnerabilities/2017/03/14/massive-security-update-from-microsoft-for-march.

[23] T. F. Dullien, "Weird machines, exploitability, and provable unexploitability," *IEEE Transactions on Emerging Topics in Computing*, 2017.

[24] Linux. (2023) Android application sandbox. [Online]. Available: https://source.android.com/docs/security/app-sandbox

[25] E. Avllazagaj, Z. Zhu, L. Bilge, D. Balzarotti, and T. Dumitraș, "When malware changed its mind: An empirical study of variable program behaviors in the real world," in *USENIX Security*, 2021.

[26] (2023) Apache web server. [Online]. Available: https://httpd.apache.org/

[27] mitre. (2023) Cwe-548: Exposure of information through directory listing. [Online]. Available: https://cwe.mitre.org/data/definitions/548.html

[28] NVD. (2023) Cve-2022-27666 detail. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-27666

[29] C. Mulliner. (2023) Cve-2016-0728 vs android. [Online]. Available: http://www.mulliner.org/blog/blosxom.cgi/security/CVE-2016-0728_vs_android.html

[30] M. Mimoso. (2023) Serious linux kernel vulnerability patched. [Online]. Available: https://threatpost.com/serious-linux-kernel-vulnerability-patched/115923/

[31] D. Vyukov. (2023) Syzkaller. [Online]. Available: https://github.com/google/syzkaller

[32] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel." in *NDSS*, 2020.

[33] D. Jones. (2023) Trinity: Linux system call fuzzer. [Online]. Available: https://github.com/kernelslacker/trinity

[34] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC CCS*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2345–2358.

[35] FSecureLABS. (2023) Kernelfuzzer: Cross platform kernel fuzzer framework. [Online]. Available: https://github.com/FSecureLABS/KernelFuzzer

[36] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "Statefuzz: System call-based state-aware linux driver fuzzing," in *USENIX Security*, 2022.

[37] (2023) The kernel address sanitizer (kasan). [Online]. Available: https://www.kernel.org/doc/html/latest/dev-tools/kasan.html

[38] (2023) The kernel memory sanitizer (kmsan). [Online]. Available: https://docs.kernel.org/next/dev-tools/kmsan.html

[39] (2023) Kernel thread sanitizer (ktsan). [Online]. Available: https://google.github.io/kernel-sanitizers/KTSAN.html

[40] (2023) The kernel concurrency sanitizer (kcsan). [Online]. Available: https://docs.kernel.org/5.19/dev-tools/kcsan.html

[41] Google. (2023) Linux kernel sanitizers. [Online]. Available: https://github.com/google/kernel-sanitizers

[42] R. Wang, K. Chen, C. Zhang, Z. Pan, Q. Li, S. Qin, S. Xu, M. Zhang, and Y. Li, "Alphaexp: An expert system for identifying security-sensitive kernel objects," in *USENIX Security*, Aug. 2023.

[43] Y. Chen and X. 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*, 2019, pp. 1707–1722.

[44] C. Polop. (2023) Peass - privilege escalation awesome scripts suite. [Online]. Available: https://github.com/carlospolop/PEASS-ng

[45] X. Tan, Y. Zhang, X. Yang, K. Lu, and M. Yang, "Detecting kernel refcount bugs with two-dimensional consistency checking," in *USENIX Security*, Aug. 2021.

[46] J. Mao, Y. Chen, Q. Xiao, and Y. Shi, "Rid: Finding reference count bugs with inconsistent path pair checking," *SIGPLAN Not.*, vol. 51, no. 4, p. 531–544, mar 2016.

[47] J. Liu, L. Yi, W. Chen, C. Song, Z. Qian, and Q. Yi, "Linkrid: Vetting imbalance reference counting in linux kernel with symbolic execution," in *USENIX Security*, Boston, MA, aug 2022.

[48] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," ser. CCS '20. New York, NY, USA: ACM, 2020.

[49] D. Liu, P. Wang, X. Zhou, W. Xie, G. Zhang, Z. Luo, T. Yue, and B. Wang, "From release to rebirth: Exploiting thanos objects in linux kernel," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 533–548, 2023.

[50] X. Zou and Z. Qian. (2022) Cve-2022-27666: Exploit esp6 modules in linux kernel. [Online]. Available: https://etenal.me/archives/1825

[51] K. Zeng, Y. Chen, H. Cho, X. Xing, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Playing for k(h)eaps: Understanding and improving linux kernel exploit reliability," in *USENIX Security*, 2022.

[52] V. Nikolenko. (2018) Linux kernel universal heap spray. [Online]. Available: https://duasynt.com/blog/linux-kernel-heap-spray

[53] J. Edge. (2013, Oct) Kernel address space layout randomization. [Online]. Available: https://lwn.net/Articles/569635/

[54] V. Nikolenko. (2023) Practical smep bypass techniques on linux. [Online]. Available: http://bit.ly/3GVjvEn

[55] W. Wu, Y. Chen, X. Xing, and W. Zou, "Kepler: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *USENIX Security 19*, 2019.

[56] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with intel tsx," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 380–392.

[57] B. F. S. GmbH. (2020, Jun) Meltdown reloaded: Breaking windows kaslr by leaking kva shadow mappings. [Online]. Available: https://labs.bluefrostsecurity.de/blog/2020/06/30/meltdown-reloaded-breaking-windows-kaslr/

[58] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *USENIX Security*, 2014.

[59] linxz. (2023) Supervisor mode execution prevention. [Online]. Available: https://linxz.tech/post/architecture/2021-10-19-smep/

[60] Linux. (2022) Linux kernel coding style. [Online]. Available: https://www.kernel.org/doc/html/v6.1/process/coding-style.html#allocating-memory

[61] llvm. (2023) llvm::castinst class reference. [Online]. Available: https://llvm.org/doxygen/classllvm_1_1CastInst.html

[62] J. Keniston, P. S. Panchamukhi, and M. Hiramatsu. (2023) Kernel probes. [Online]. Available: https://www.kernel.org/doc/html/latest/trace/kprobes.html

[63] G. Rodrigues. (2009) Poke-a-hole and friends. [Online]. Available: https://lwn.net/Articles/335942/

[64] P. Krishnan, "Hardware breakpoint (or watchpoint) usage in linux kernel," in *Proceedings of the Linux Symposium*. Citeseer, 2009, pp. 149–158.

[65] (2023) addr2line(1) — linux manual page. [Online]. Available: https://www.man7.org/linux/man-pages/man1/addr2line.1.html

[66] scikit learn. (2023) Jaccard similarity. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.jaccard_score.html

[67] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," ser. CCS, New York, NY, USA, 2018.

[68] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[69] SecWiki. (2023) linux-kernel-exploits. [Online]. Available: https://github.com/SecWiki/linux-kernel-exploits

[70] P. Starzetz. (2005) Linux kernel `uselib()` privilege elevation. [Online]. Available: https://isec.pl/en/vulnerabilities/isec-0021-uselib.txt

[71] L. H. (2009) Linux kernel heap tampering detection. [Online]. Available: http://phrack.org/issues/66/15.html

[72] J. Corbet. (2018) The slab and protected-memory allocators. [Online]. Available: https://lwn.net/Articles/753154/

[73] R. Landley. (2007). [Online]. Available: https://www.kernel.org/doc/html/latest/core-api/rbtree.html

[74] Will. (2021). [Online]. Available: https://www.willsroot.io/2021/08/corctf-2021-fire-of-salvation-writeup.html

[75] E. Avllazagaj. (2023) Cve-2022-27666: My file your memory. [Online]. Available: https://albocoder.github.io/exploit/2023/03/13/KernelFileExploit.html

# A Appendix

## A.1 Analysis of Detected Exploitable States

We present our manual analysis for the proof-of-concept code detected by SCAVY, focusing on exploring concrete privilege escalation exploits from the detected exploitable states.

### A.1.1 Case 1: `file::f_mapping`

The `file::f_mapping` memory target can be leveraged to create exploits for two different scenarios.

**Scenario 1: Attacking Inaccessible File.** An exploit creates a root-owned process opening a privileged file, which is completely inaccessible in an unprivileged process. For example, it can spawn many '`passwd`' processes, spraying the slab memory with the '`/etc/shadow`' file's `file` kernel structures. Then, the exploit also creates an unprivileged file such as a temp file (e.g., `/tmp/file`). Next, the exploit leverages a vulnerability with a read capability to read the `file::f_mapping` of the '`/etc/shadow`,' which will be copied to the `file::f_mapping` of the temp file through a vulnerability with a write capability. In other words, the values of `file::f_mapping` of `/etc/shadow` and the temp file are swapped. After this, reading and writing the temp file is accessing the `/etc/shadow`'s content.

**Scenario 2: Attacking Unwritable (but readable) File.** Scenario 1 creates many root-owned processes (e.g., `passwd` processes), which might raise suspicion. If one wants to write an unwritable file (instead of read and write an inaccessible file, as we see in scenario 1), one can achieve it more subtly. Specifically, an exploit can open many '`/etc/passwd`' files

in read mode and a temp file in read/write mode. Swapping the `file::f_mapping` of the '`/etc/passwd`' and the temp file allows the exploit to write the '`/etc/passwd`', escalating the privilege. With the privilege, it can create a root-level account by editing the '`/etc/passwd`' file.

### A.1.2 Case 2: `vm_area_struct::vm_file`

This memory target can, in practice, be used to access privileged files' contents if they exist in shared memory-mapped files (i.e., mapped with `mmap()` with the MAP_SHARED flag). Specifically, an exploit first creates a shared memory of a temp file with read and write permissions, which will create a `vm_area_struct::vm_file` kernel structure. Then, it overwrites the `vm_area_struct::vm_file` of the temp file with the values of the `vm_area_struct::vm_file` of a root-owned file, using vulnerabilities with read and write capabilities. After the corruption, an exploit invokes `msync` to synchronize the mapping with the corrupted new `vm_area_struct::vm_file`.

Note that, in Section 3.1, we show that one can modify our exploit of `CVE-2022-27666` to write into any of the dummy file's `vm_area_struct` so that it would access the content of `/etc/passwd`.

```
#23:08:35 executing program 0 (corruption 'struct.
    anon_vma_chain*+40 (8)' at call 4):
r0 = semget(0x3, 0x3, 0x481)
getgroups(0x2, &(0x7f0000000180)=[<r1=>0xee01,<r2=>0xee01])
ioctl$NS_GET_OWNER_UID(0xffffffffffffffff, 0xb704,&(0
    x7f0000000200)=<r3=>0x0)
semctl$IPC_SET(r0, 0x0, 0x1, &(0x7f0000000240)={{0x2,0xee01,
    r2, r3, 0xee01, 0xa0,
0x5f3}, 0xffffffff,0x0, 0x0, 0x0, 0x0, 0x0, 0x7})
r4 = semget(0x3, 0x4, 0x40)
semctl$GETVAL(r4, 0x2, 0xc, &(0x7f0000000000)=""/248)
semget(0x0, 0x2, 0x4)
getgroups(0x1, &(0x7f0000000100)=[<r5=>r1])
r6 = getegid()
getgroups(0x5, &(0x7f0000000140)=[r5, r2, r2, r6, r5])
```

Listing 3: Syzkaller representation of the PoC.

### A.1.3 Case 3: `anon_vma_chain::rb`

SCAVY found this kernel data structure field as a potential memory target. However, we did not include it as a memory target as corrupting it does not escalate privilege.

SCAVY discovered this memory target by observing a crash during the execution of `semctl()` after corrupting the `anon_vma_chain::rb` field with a random value. Specifically, it crashes at `semctl$GETVAL` shown in line 8 in Listing 3. We further analyze the structure to understand the crash. In particular, the `anon_vma_chain` structure is used for anonymous virtual memory, which is essentially shared memory not backed by a file system. The `rb` field links the `anon_vma_chain` structure into a red-black tree [73] for search optimization. As a result, corrupting it with a random value, which is not a valid pointer, caused a memory dereference error. While we could not enhance the case to replace the `rb`'s value with another valid `rb`'s value, replacing a valid value may impact the search operations using the red-black tree.

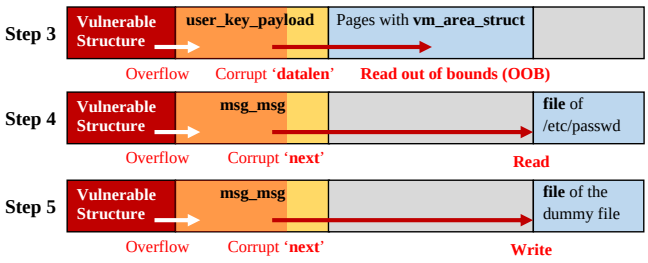


Figure 9: Visualizing Exploit of CVE-2022-27666.

## A.2 Exploiting CVE-2022-27666

The exploit requires memory massaging of Linux page allocator. Hence, we borrow the same noise-mitigation and massaging technique from an existing exploit [50]. The rest of the exploit is summarized in six steps as follows:

1. Allocate `user_key_payload` next to the vulnerable structure. Then, overflow and corrupt `user_key_payload::datalen` to obtain the read out of the bounds (OOB) capability.
2. Open two files: one dummy file with read/write permissions and another root-owned unwritable file that can only be opened read-only by non-root users (e.g., `/etc/passwd`).
3. Repeatedly call `open()`[8] to assure the `file::f_mapping` structures allocate next to user key payload. Leak the two `vm_area_struct::file` pointers of the opened files.
4. Rearrange the memory to now put the `msg_msg` structure next to the vulnerable structure and overflow into it such that the `msg_msg::next` points to the leaked `file` structure. With the corruption, reading `msg_msg` leaks the content of the `file` structure of the `/etc/passwd`.
5. Rearrange the kernel memory to use the `msg_msg` to achieve arbitrary write capability [74], and use it to overwrite the `file::f_mapping` of the dummy file with the leaked `f_mapping` value of the `/etc/passwd`.
6. Call `write()` with the dummy file's descriptor to append a new root-level account on `/etc/passwd`. As shown in the blog post [75], the root privilege can be obtained by simply running `su` with the appended account.

In Figure 9, we illustrate the memory layout for the steps 3 to 5. After step 5, as SCAVY detected, the `/etc/passwd` becomes writable through the dummy file's descriptor.

[8] There is a limit of 1,000 invocations of `open()`. However, most of the allocations would fall in the memory holes created in step 1, without exhausting the limit. If necessary, one can use the following method to overcome the 1,000 limit: open both files once and `mmap` the opened files until the `vm_area_struct` are allocated next to the corrupted `user_key_payload`.

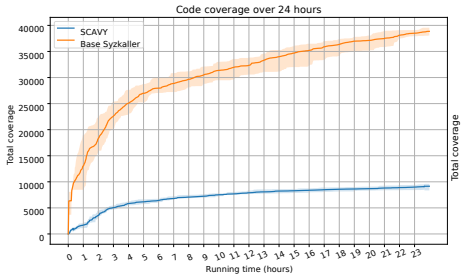


Figure 10: Code coverage of SCAVY VS the unmodified fuzzer.

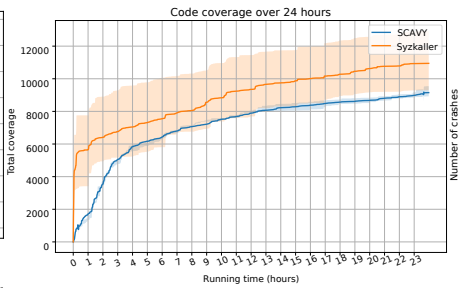


Figure 11: Code Coverage Over Time

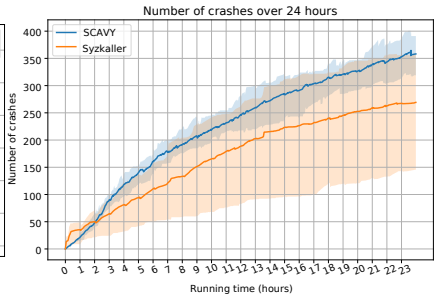


Figure 12: # of Crashes Observed

| Structure | Field | # PoCs |
|---|---|---|
| proc_inode | pid | 52 |
| anon_vma | root | 36 |
| anon_vma | parent | 41 |
| socket_alloc | vfs_inode.i_flags | 2 |
| socket_alloc | vfs_inode.xa_head | 6 |
| pde_opener | c | 20 |
| io_uring_probe | ops | 6 |
| file | f_ra | 5 |
| io_wq | worker_done.pprev | 1 |
| sk_security_struct | peer_sid | 7 |
| others | - | 178 |

Table 3: **Memory Targets Found by FUZE's Symbolic Execution.** Multiple PoCs for each memory target are detected.

## A.3 Evaluation of SCAVY's Detector Step

| Deviation | Count |
|---|---|
| Coverage | 658 |
| Corruption hits | 452 |
| Return values | 200 |
| Buffers | 176 |

Table 4: Count of deviation conditions.

| Deviating Syscall | Count |
|---|---|
| pread64 | 283 |
| io_submit | 35 |
| ioctl$sock_SIOCGIFINDEX_802154 | 27 |
| ioctl$ifreq_SIOCGIFINDEX_team | 26 |
| ioctl$sock_SIOCGIFINDEX_80211 | 23 |
| seteuid | 16 |
| ioctl$BTRFS_IOC_GET_SUBVOL_ROOTREF | 12 |
| ioctl$BTRFS_IOC_INO_LOOKUP_USER | 11 |
| socketpair$nbd | 10 |
| sendmsg$ETHTOOL_MSG_LINKINFO_GET | 9 |

Table 5: Count of syscalls demonstrating deviation

## A.4 Comparing with the unmodified Syzkaller

We compare SCAVY's fuzzer against an unmodified Syzkaller baseline using the same methodology described in Section 7.2. We disabled the reproducer and ran it on 1 VM with the debug flag. Figure 10 shows that SCAVY's fuzzer achieved a lower code coverage than the original Syzkaller. This does not mean our fuzzer is ineffective, but highlights that SCAVY's fuzzer is more focused on the memory target accessing code rather than exploring irrelevant kernel code.

| Structure::Field (offset) | Structure::Field (offset) |
|---|---|
| anon_vma::parent::rb_root (+0) | mount::mnt_mounts::prev (+0) |
| anon_vma::refcount (+0) | pde_opener::file (+0) |
| anon_vma::root::count (+0) | pid_namespace::ucounts (+0) |
| anon_vma::rwsem (+0) | pipe_buffer::ops (+0) |
| anon_vma::rwsem (+24) | proc_inode::pid (+0) |
| avc_node::ae::pprev (+16) | proc_inode::pid (+4) |
| bio::bi_private (+0) | proc_inode::sibling_inodes (+8) |
| ctl_table_header::(anon) (+16) | proc_inode::sysctl (+0) |
| dentry::d_lockref (+0) | proc_inode::vfs_inode::i_acl (+0) |
| dentry::d_op (+0) | proc_inode::vfs_inode::i_data (+0) |
| dnotify_mark::fsn_mark (+56) | proc_inode::vfs_inode::i_data (+8) |
| ext4_inode_info::vfs_inode (+280) | proc_maps_private::mm (+0) |
| file_lock::fl_link::prev (+0) | shmem_inode_info::swaplist (+0) |
| file_lock::fl_list::pprev (+0) | vfs_inode::i_data (+0) |
| file_lock::fl_wait (+8) | vfs_inode::i_lock (+0) |
| files_struct::fdtab (+64) | vfs_inode::i_mapping (+0) |
| fs_struct::pwd::dentry (+0) | sock::sk_callback_lock (+0) |
| fs_struct::seq (+0) | sock::sk_peer_cred (+0) |
| fs_struct::users (+0) | socket::i_acl (+0) |
| hugetlbfs_inode_info::vfs_inode (+272) | socket::i_default_acl (+0) |
| hugetlbfs_inode_info::vfs_inode (+60) | socket::i_op (+0) |
| inode::i_data::prev (+0) | vfs_inode::i_data::prev (+0) |
| inode::i_lock (+0) | vfs_inode::i_lru::next (+0) |
| journal_head::b_triggers (+0) | vfs_inode::i_sb_list::next (+0) |
| kioctx::users (+0) | user_struct::epoll_watches (+32) |

Table 6: Potential Privilege Escalations Detected by SCAVY but *Unverified*.

```
open("/etc/passwd",O_APPEND)
open("/etc/shadow",O_APPEND)
open("/etc/shadow",O_RDONLY)
setuid(0)
setgid(0)
open("/proc/self/mem",O_RDONLY)
open("/proc/1/mem",O_RDONLY)
opendir("/root")
socket(AF_INET, SOCK_RAW, IPPROTO_RAW)
mount("dev/sda1", "/mnt", "ext4", MS_RDONLY, NULL);
chmod("/etc/shadow", S_IRUSR | S_IWUSR)
kill(1, SIGKILL)
mknod("/dev/mydevice", S_IFCHR | 0600, makedev(10, 100)
unlink("/etc/shadow")
symlink("/etc/passwd", "/tmp/passwd")
```

Table 7: List of system calls that conduct operations dependent on privilege.