SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel

Yueqi Chen
ychen@ist.psu.edu
The Pennsylvania State University

Xinyu Xing
xxing@ist.psu.edu
The Pennsylvania State University

ABSTRACT

To determine the exploitability for a kernel vulnerability, a security analyst usually has to manipulate slab and thus demonstrate the capability of obtaining the control over a program counter or performing privilege escalation. However, this is a lengthy process because (1) an analyst typically has no clue about what objects and system calls are useful for kernel exploitation and (2) he lacks the knowledge of manipulating a slab and obtaining the desired layout. In the past, researchers have proposed various techniques to facilitate exploit development. Unfortunately, none of them can be easily applied to address these challenges. On the one hand, this is because of the complexity of the Linux kernel. On the other hand, this is due to the dynamics and non-deterministic of slab variations.

In this work, we tackle the challenges above from two perspectives. First, we use static and dynamic analysis techniques to explore the kernel objects, and the corresponding system calls useful for exploitation. Second, we model commonly-adopted exploitation methods and develop a technical approach to facilitate the slab layout adjustment. By extending `LLVM` as well as `Syzkaller`, we implement our techniques and name their combination after `SLAKE`. We evaluate `SLAKE` by using 27 real-world kernel vulnerabilities, demonstrating that it could not only diversify the ways to perform kernel exploitation but also sometimes escalate the exploitability of kernel vulnerabilities.

CCS CONCEPTS

• **Security and privacy** → *Operating systems security*; *Software security engineering*;

KEYWORDS

OS Security; Vulnerability Exploitation;

ACM Reference Format:
Yueqi Chen and Xinyu Xing. 2019. SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel. In *Proceedings of 2019 ACM SIGSAC Conference on Computer and Communications Security, London, United Kingdom, November 11–15, 2019 (CCS'19),* 16 pages.
https://doi.org/10.1145/3319535.3363212

1 INTRODUCTION

Despite extensive code review, a Linux kernel, like all other software, still inevitably contains a large number of bugs and vulnerabilities [42]. Compared with vulnerabilities in user-level applications, a vulnerability in kernel code is generally more disconcerting because kernel code runs with a higher privilege and the successful exploitation of a kernel vulnerability could provide an attacker with full root access.

One straightforward solution to minimize the damage of Linux kernel defects is to have software developers and security analysts patch all the bugs and vulnerabilities immediately. However, even though allowing anyone to contribute to code development and fix, the Linux community still lacks the workforce to sift through each software bug timely. As such, the Linux community typically prioritizes kernel vulnerability remediation based on their exploitability [30] (i.e., assessing a software bug based on ease of its exploitation).

To determine the exploitability for kernel vulnerabilities, an analyst typically needs to manipulate slab (i.e., heap in kernel), manually craft working exploits and demonstrate the capability in obtaining control over a program counter or escalating privilege for a user process. In general, this is a time-consuming and labor-intensive process. On the one hand, this is because given a kernel vulnerability, a security analyst lacks the knowledge about what kernel objects and system calls are useful for vulnerability exploitation. On the other hand, this is because even if the analyst figures out the kernel objects, as well as the corresponding system calls, he may still have no clue about how to use them to obtain the desired slab layout accordingly.

In the past, there are many techniques developed to facilitate the exploit development (e.g., [3, 4, 6, 19, 20, 24, 35, 46, 48]), and a recent research work indicates an analyst can use various test cases to explore the desired memory layout for vulnerability exploitation [15]. For the two following reasons, none of these techniques, however, can be directly applied or tweaked to tackle the aforementioned challenges. First, a Linux kernel is a complex system, in which many kernel objects useful for exploitation cannot be allocated through test cases regularly used. As we will show in Section 6, even merely using `Syzkaller` (a kernel fuzzing tool) [13] to generate test cases, we are still not able to pinpoint sufficient kernel objects suitable for kernel exploitation. Second, a Linux kernel contains many routines, making the slab very dynamic and non-deterministic. Even if an analyst could observe the desired memory layout through one test case, he is highly unlikely to use the same test case to obtain that layout as he expects.

In this work, we propose a new approach, to facilitate the development of working exploits for various kinds of kernel vulnerabilities or, more precisely, a technique to facilitate slab manipulation with the goal of obtaining the capability in hijacking control flow[1]. We name our technique after SLAKE, standing for **SLA**b manipulation for **K**ernel **E**xploitation. Technically speaking, it tackles the challenges above from the two angles. First, SLAKE performs static and dynamic analysis to identify the objects useful for kernel exploitation and track down corresponding system calls. Second, SLAKE models kernel exploitation methods commonly adopted. Using the model, it then designs a technical approach to facilitate the capability of security analysts in adjusting slab layout and thus obtaining the control over the program counter.

Given the pioneering research works (e.g., [24, 45, 46]), we do not claim SLAKE is the first technique designed for kernel exploitation facilitation. However, we argue that it is the first work that can facilitate the manipulation of the slab and thus assist an analyst in hijacking the control over kernel execution. Besides, SLAKE is the first work that can facilitate kernel exploitation for various types of kernel vulnerabilities (e.g., UAF, Double Free, and OOB). Using 27 real-world kernel vulnerabilities, we show that SLAKE could not only identify the kernel objects and system calls commonly adopted by professional analysts for kernel exploitation but more importantly, pinpoint objects and system calls that have never been used in the public exploits. We argue this is a very beneficial characteristic for security analysts because, as we will show in Section 6, this could significantly diversify the working exploits and potentially escalate the exploitability for kernel vulnerabilities.

In summary, this paper makes the following contributions.

- We design a new technical approach that utilizes static/dynamic analysis to identify the kernel objects and system calls useful for kernel exploitation.
- We model commonly-adopted kernel exploitation methods and then design a manipulation method to adjust a slab and thus obtain the layout desired for kernel exploitation.
- By extending LLVM and Syzkaller, we implement SLAKE and demonstrate its utility in crafting working exploits by using 27 real-world vulnerabilities in the Linux kernel.

The rest of this paper is organized as follows. Section 2 describes the background and the key challenges of this work. Section 3 specifies how to utilize static and dynamic analysis to explore objects and system calls useful for kernel exploitation. Section 4 introduces how to adjust slab layout. Section 5 and 6 describe the implementation and evaluation of SLAKE. Section 7 discusses some related issues and future work, followed by the discussion of related work in Section 8. Finally, we conclude the work in Section 9.

2 BACKGROUND AND CHALLENGES

In this section, we first describe the problem scope, assumptions, and objectives of this work. Then, we introduce the technical background, followed by the discussion of kernel exploitation challenges.

2.1 Problem Scope, Assumptions and Goals

Problem scope. This work focuses only on developing exploitation techniques for those kernel vulnerabilities that result in the corruption of the memory managed by SLAB/SLUB allocator. We claim the problem in this scope is meaningful and non-trivial. This is because, after being triggered, most kernel vulnerabilities only demonstrate the capability in corrupting memory regions tied to the SLAB/SLUB and, more importantly, there has not yet been a generic, systematic approach that could facilitate the manipulation of kernel memory layout and thus benefit the exploitation of such vulnerabilities.

Assumptions. By definition, the capability of a vulnerability is a power, indicating at which memory addresses the vulnerability gives an adversary the ability to overwrite data freely. In this work, we consider the capability of a vulnerability through a PoC program, which could panic kernel execution but not perform actual exploitation. Under the assistance of address sanitizer KASAN [10] and other debugging tools (e.g., GDB [40]), a security researcher could manually learn the capability of a vulnerability. It should be noted we do not assume researchers could go beyond the capability manifested by a PoC and find more powerful capability for a target vulnerability. For example, if the PoC demonstrates the ability to overwrite only one byte, but the vulnerability could actually provide the capability of performing an arbitrary write, we conservatively assume a researcher could obtain only the one-byte overwriting capability.

In addition, we assume that a capability of controlling program counter directly implies the exploitability of a vulnerability. On the one hand, this is because many previous works have already demonstrated an adversary can easily bypass kernel mitigation and complete successful exploitation as long as he could hijack the control of kernel execution [2, 7, 9, 14, 21–23, 27, 28, 32, 45]. On the other hand, this is because, with the ability to hijack kernel execution, an adversary can always convert this capability into a way to overwrite critical kernel object and thus carry out privilege escalation or information leakage [17].

Goals. As is mentioned in the section above, the goals of this work are in two folds. First, it aims to provide security researchers with the ability to identify useful kernel objects and corresponding system calls. Second, it aims to facilitate security researchers' capability in obtaining a desired memory layout for kernel exploitation. As a result, different from the research in exploitation automation (e.g., [3, 6]), we focus on ❶ building an automated approach to help security analysts identify useful kernel objects and system calls, ❷ building a technical approach to facilitate researchers' capability in memory layout manipulation. In fact, the identification of object and system calls, as well as memory manipulation are just one key component of kernel exploitation. Therefore, we do not claim the work is an end-to-end automated approach for kernel exploitation.

2.2 Technical Background

Here, we briefly introduce how SLAB/SLUB allocator works in Linux kernel, followed by the kernel exploitation techniques commonly adopted in the real world.

2.2.1 *SLAB/SLUB Allocator.*

SLAB/SLUB allocator organizes physical memory in a unit of cache. Kernel objects in the same cache share the same type or have similar sizes. Inside each cache are a set of slabs which are contiguous pages. For each newly created slab, SLAB/SLUB allocator partitions it into multiple individual

[1]As we will clarify in Section 2.1, this work focuses on the ability to hijack control flow but not that to escalate privilege for a userland process.

slots. For SLUB allocator, each unoccupied slot contains a metadata header which stores the address of the next slot unoccupied. Through metadata, unoccupied slots are organized in the form of a singly linked list with a dummy head `freelist`. Slightly different from SLUB allocator, SLAB allocator does not utilize metadata headers to organize the slots unoccupied. Rather, it employs an index array also named `freelist` to implement the logic of the linked list. When other kernel components request a memory region for a new object, both SLUB and SLAB allocator retrieve and assign the first slot of the list to hold the new object and update the `freelist`. When an object is deallocated (freed), both SLUB and SLAB allocator reclaim the freed slot and add it back to the beginning of the linked list. As such, SLAB/SLUB allocator work in a fashion of LIFO (Last In, First Out).

2.2.2 Kernel Exploitation Approaches. The kernel exploitation can be viewed as a three-step procedure. ❶ an adversary summarizes the type of a target vulnerability as well as at which memory addresses he could manipulate data freely (i.e., the capability of corrupting memory regions). ❷ The adversary determines the specific exploitation approaches to obtaining the ability to hijack control flow. ❸ Using the primitive of control flow hijack, the adversary disables kernel mitigation and protection, and thus performs ultimate exploitation.

Generally speaking, four exploitation approaches could lead to a control over the program counter. In the following, we briefly introduce these exploitation approaches. It should be noted that this work does not discuss the techniques developed for vulnerability capability summary nor those for bypassing mitigation. As is described in Section 2.1, they are out of the scope of this research work.

I. Manipulation through OOB write. Given a vulnerability with the ability to perform an out-of-bounds (OOB) write, there are two common approaches to hijacking control flow. The first is to overwrite a function pointer in the adjacent object and then dereference that pointer for exploitation. For another approach, the exploitation overwrites a data pointer in the adjacent object and then dereference a function pointer through that data pointer. To illustrate this, Figure 1a depicts an example. In regular operations, we assume Linux kernel dereferences the function pointer `fptr` ↪ through the pointer `ptr` referencing the kernel object `A`. Using the OOB write, an attacker could first overwrite the data object pointer `ptr`, referencing it to a memory region under his control (e.g., physmap [20, 37] or userland memory). In the memory area under the attacker's control, he could then carefully craft a fake data object with the function pointer `fptr` referencing the target of the attacker's desire.

II. Manipulation through UAF. Different from the manipulation approaches tied to OOB vulnerabilities, use-after-free (UAF) vulnerabilities have unique exploitation approach. Given a UAF vulnerability, an adversary first selects a kernel object (i.e., a spray object), the content of which is completely under his control. Then, he overlays that object on top of a vulnerable object. In this way, the attacker could overwrite the critical data (e.g., function or data object pointer) in that vulnerable object. Similar to the aforementioned approach, with the ability to manipulate a function or data object pointer, an attacker could easily obtain control over the kernel execution.

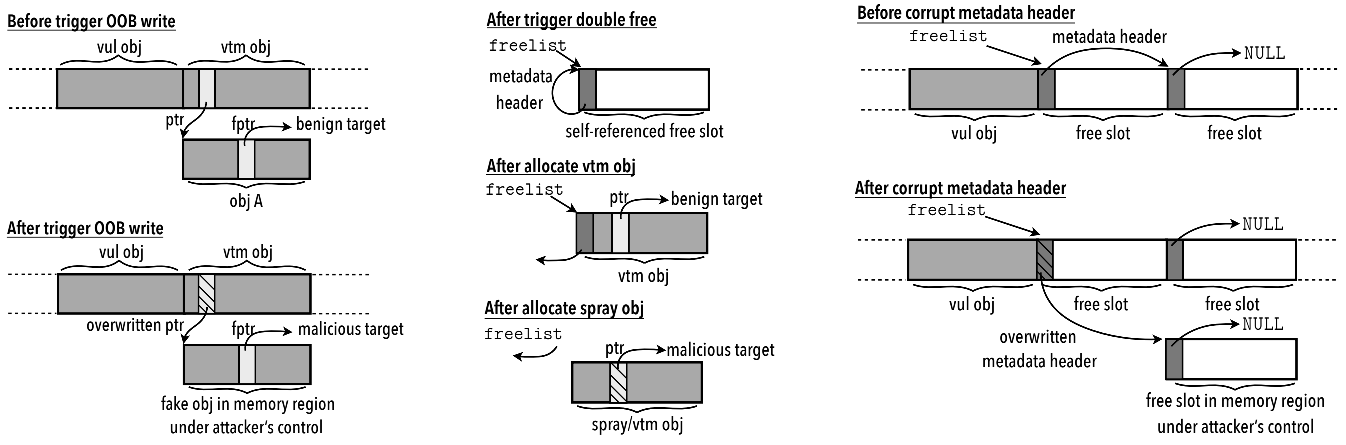
III. Manipulation through double free. With respect to the double free vulnerability, when the vulnerability is triggered, the metadata header of a vulnerable object refers to itself. As such, its exploitation could be achieved through the process below. First, an adversary carefully selects a victim object. Second, through heap spray, he uses that selected object to take over the freed slot pertaining to that self-referenced metadata. Third, he selects a spray object and allocates that object. As is shown in Figure 1b, after the allocation of the victim object, the `freelist` is updated with the value of metadata header, and the `freelist` references the victim object selected. As a result, the adversary can leverage the spray object to overwrite the function or object pointer residing in the victim object. Again, similar to the aforementioned exploitation approach tied to OOB, this allows the adversary to obtain the control over the program counter easily.

IV. Manipulation through metadata corruption. In addition to the manipulation of function and data object pointers, all the aforementioned vulnerabilities provide an attacker with the potential to tamper with the metadata header of free slots. As such, for all the aforementioned vulnerability, an alternative manipulation approach is to overwrite metadata header and trick SLUB allocator into allocating a victim object to a memory region under an attacker's control. To illustrate this, Figure 1c shows an example. Through the capability of a vulnerability, an attacker first overwrites the metadata header, referencing it to a region under the attacker's control. Since the metadata header indicates the next unoccupied region, the attacker could allocate a series of objects and force one victim object appearing at that desired memory region. He could easily manipulate the function pointer or data object pointer in the victim object, trigger the function pointer dereference and thus obtain the control over the program counter.

2.3 Key Challenges

Despite the commonly-adopted approaches mentioned above, it is still challenging for an adversary and even a professional security analyst to perform a successful exploitation. This is mainly because an adversary lacks the following knowledge.

❶ Which kernel objects are suitable for exploitation? In kernel exploitation, an adversary needs to select an exploitation approach and corresponding object(s). Take an OOB vulnerability for example. To hijack control flow through this vulnerability, an adversary typically overwrites the critical data in the adjacent object. However, it is common that the adjacent object may not contain critical data such as function or object pointers. Therefore, one common operation for that adversary is to allocate an object (with a function pointer enclosed) to the corresponding location prior to the trigger of that vulnerability. However, a Linux kernel encloses many objects. In order to find that appropriate object for his exploitation, the adversary generally has to seek through all the objects. In this process, he also has to take the vulnerable object into consideration. This is simply because SLUB/SLAB allocator manages and groups data objects based on their types and sizes, and only the data objects sharing the same type or similar sizes could be placed in the same slab.



(a) Exploitation through OOB write **(b) Exploitation through double free** **(c) Exploitation through corrupting metadata**

Figure 1: The illustration of some kernel exploitation approaches.

❷ How to (de)allocate objects and dereference corresponding pointers? An adversary typically utilizes a set of system calls to (de)allocate selected objects or dereference pointers through these objects. In a Linux kernel, there are hundreds of system calls with various arguments. Given a target object, there has not yet been a knowledge base indicating which group of system calls can be used for its (de)allocation, nor prior knowledge specifying which system calls could be applied to dereference a function pointer through that target object. Under this situation, when performing kernel exploitation, adversaries typically seek the kernel objects repeatedly adopted across vulnerabilities as well as their corresponding system calls. From the perspective of adversaries, this not only restricts the ways to perform exploitation but more importantly leads vulnerabilities unexploitable. From the viewpoint of defenders, repeatedly-used data objects provide security analysts with an opportunity to build intrusion signatures and thus ease their identification for the potential intrusion.

❸ How to systematically adjust system calls to obtain the desired slab layout? In fact, even if an adversary tracks down the system calls for a particular object, it is still difficult for him to obtain the desired memory layout. This is because, when allocating the target object, in addition to the object of interest, the system calls identified also allocate or deallocate other kernel objects, resulting in unexpected variation in memory layout. Take the aforementioned OOB case for example again. The OOB vulnerability provides an adversary with the ability to overwrite critical data in its neighbor object. Using a set of system calls, an adversary could allocate a victim object containing a function or object pointer on the target slab. However, along with the allocation of the victim object, the system calls at the first place might allocate irrelevant kernel objects, which makes the victim object turn out to be at an unexpected spot. When this situation occurs, adversaries typically look for alternative system calls or adjust memory layout in an ad-hoc manner. In practice, there is no guarantee to find a memory layout through these two approaches. As a result, the side effect involved by system calls further reduces the number of system calls available and even jeopardizes the exploitability of that vulnerability.

3 OBJECT & SYSCALL IDENTIFICATION

To tackle the aforementioned challenges ❶ and ❷, an instinct reaction is to extensively enumerate system calls through a large corpus of test cases or fuzz testing and, at the same time, observe the change of objects on slab as well as keep track of function pointer dereference. In this way, one could easily correlate the system calls to the objects of his interest as well as corresponding function pointer dereference. For the following two reasons, this approach is however not suitable for our problem, even though a similar idea has already been applied to explore memory layout manipulation in the userland exploitation [15].

First, the code space of a Linux kernel is typically significantly larger than that of a userland application. It is impossible for an adversary to use regular test cases to identify all the sites where the objects of his interest are (de)allocated and corresponding function pointers are dereferenced. Second, Linux kernel contains hundreds of system calls, each of which takes as input various arguments. This makes it very inefficient to perform fuzz testing against various system calls. As we will show in Section 6, even by using a state-of-the-art kernel fuzzing tool, it is still difficult to track down the objects of interest in an effective and efficient fashion.

In this section, we propose a new technical approach to tackle the challenges ❶ and ❷ mentioned in the section above. At the high level, our approach first identifies object types of our interest by using static analysis. Using the type information and a set of heuristics, our approach further pinpoints the objects of our interest as well as the sites of our interest (i.e., the sites where the objects are (de)allocated or corresponding pointers can be potentially dereferenced). Along with this procedure, we record the name of the object type, object size and the cache that holds the object. In addition, we store the offset of function/object pointers in each object. With these designs, we can obtain the basic information for kernel objects. To be able to track down the system calls that could (de)allocate and dereference the objects of our interest, our approach performs kernel fuzzing under the guidance of a kernel call graph. More specifically, we first perform reachability analysis over the call graph and preserve only those system calls that could potentially reach to the sites of our interest. With this, we can reduce the number of system calls needed to explore and thus improve the effectiveness and efficiency of system call identification. For each site of our interest, we then perform fuzz testing using the results of reachability analysis, exploring the actual path towards that site. In this way, our approach can identify the arguments and context needed for the system calls to (de)allocate and dereference the object of our interest. In the following, we present the detail of this approach.

3.1 Identifying Objects & Sites of Interest

In a Linux kernel, there are thousands of objects. However, they are not equally critical for kernel exploitation. In the following, we first introduce how we identify the kernel objects critical for exploitation. Then, we describe how we track down the sites at which kernel (de)allocates these critical data objects. Finally, we discuss how we identify the dummy sites where kernel could potentially dereference a function pointer through an object of our interest (or more precisely speaking a victim object).

3.1.1 Critical kernel objects. Given a kernel vulnerability, there are two kinds of objects critical for the success of kernel exploitation. One is the victim object typically used for exploiting OOB and double free vulnerabilities. The other is the spray object generally used for facilitating the exploitation of UAF or double free vulnerabilities.

Victim object. A victim object typically contains a function or an object pointer. As is specified in Section 2.2.2, by overlaying this object on a vulnerable object freed twice or placing this object adjacent to an object overflowed, an adversary might be able to leverage the capability of the vulnerability to overwrite the pointer residing in the object and thus obtain the control over kernel execution. In this work, we, therefore, pinpoint victim objects by examining the object types. To be specific, for each structure and union type in kernel code, we examine whether it contains a function or an object pointer. For those with a function pointer enclosed, we deem them as victim objects. For those with an object pointer included, we track down the type of the object that the pointer refers to by following the dereference chain defined in each data object (e.g., `objA->objB->` ↪ `objC->...`). We deem an object as a victim object if, through its enclosed object pointers, we could perform multi-step dereference and eventually identify a function pointer dereference (e.g.,`objA->` ↪ `objB->objC->fptr`). It should be noted that, when identifying a victim object by using multi-step dereference, we exclude struct fields indicating a linked list (e.g., `struct list_head`) because such dereference chain forms a circle which could cause our approach to enter an unterminated procedure.

Spray object. The usages of a spray object include ❶ taking over the slot of a freed object – still referenced by a dangling pointer – as well as ❷ overwriting the content of that freed object. As a result, a spray object does not have to contain a function or object pointer but provides an adversary with the ability to copy arbitrary data from userland to kernel slab. In this work, we follow this characteristic and track down spray objects in kernel source code as follows. First, we retrieve the argument `dst` of the kernel function `copy_from_user(void *dst, void *src, unsigned long length)`. Then, we examine if the pointer `dst` is the return value of a slab allocation function such as `kmalloc()` and `kmem_cache_alloc()`.

3.1.2 Allocation and deallocation sites. In kernel exploitation, we usually allocate and free critical objects on the slab. In order to identify the system calls that can truly allocate or free a critical object, we first pinpoint the sites of (de)allocation in the kernel code source.

Allocation sites. For the functions taking the responsibility of allocation (e.g., `kmalloc()`), their return value is typically of type "void*". By looking at the site of a function call, it is therefore difficult to know whether that allocation site ties to an object of our interest (e.g., a victim or spray object). In order to solve this problem, we examine the def-use chain of the return value for each allocation function and deem the site of an allocation call as a site of allocation if and only if that return value is cast to the type identified.

Deallocation sites. For the kernel functions associated with deallocation (e.g., `kfree()`), they typically take as input the pointer of an object. Similar to the way to pinpoint allocation sites, we can, therefore, track down the deallocation sites tied to victim object by looking at the type of the object pointer passed to the deallocation function.

3.1.3 Dummy dereference sites. For many kernel exploitation methods, an adversary needs to dereference a function pointer through a victim object. This typically means two different situations. One is to dereference a function pointer residing in a victim object, and the other is to dereference a function pointer through a multi-step dereference chain (e.g., the example shown in Figure 1a). Technically, it is challenging to pinpoint these dereference sites in the kernel source code through a static interprocedural data flow analysis. On the one hand, this is because a static interprocedural data flow analysis needs to be performed on a control flow graph (CFG) and it is difficult to build an accurate kernel CFG. On the other hand, this is because Linux kernel has a soft interrupt mechanism such as Read-Copy Update (RCU), which deallocates an object and thus dereferences a function pointer in an asynchronous fashion.

To address the problem above, we first define and identify dummy dereference sites in Linux kernel source code. Then, we use a fuzz testing and a dynamic data flow analysis to track down the true dereference sites along with the corresponding system calls. In this paper, we describe our dynamic analysis and fuzz testing in Section 3.2.2. In the following, we describe how we define and identify dummy dereference sites. Regarding the function dereference through an RCU mechanism, we deem the statements pertaining to the RCU free (e.g., `kfree_call_rcu()` and `call_rcu_sched()`) as the dummy dereference sites. This is because when these kernel functions are invoked to deallocate a corresponding victim object, in an asynchronous manner, the Linux kernel will invoke the function referred by the function pointer residing in that victim object. With respect to non-RCU-related dereference, we treat the dereference of the enclosed pointer as our dummy dereference site. For example, given a victim object `A` containing a pointer `ptr`, we deem the dereference of the pointer `ptr` as our dummy dereference site.

3.2 Finding System Calls

The aforementioned static analysis approach gives us the ability to identify at which sites a particular type of objects could be (de)allocated (or at which sites pointers in those objects could be dereferenced). However, it does not provide us with the information of what system calls and their arguments we should use to interact with these objects and pointers. To address this problem, one instinctive reaction is to directly perform kernel fuzzing, through which we could explore the system calls pertaining to the sites of our interest. However, such an approach is not sufficiently helpful for exploit development. As we will describe in Section 4, in order to perform a successful kernel exploitation, an adversary usually needs to free an object intentionally allocated for obtaining a desired memory layout. In addition, he needs to dereference a function pointer in an intentionally-allocated victim object. By simply using kernel fuzzing, an adversary could identify the system calls and their arguments that could deallocate the type of objects we are interested in (or dereference the pointers of our interest). However, this approach neither guarantees we truly deallocate the objects we intentionally allocate nor ensure we actually dereference the function pointer in the intentionally-allocated victim object. For example, using the aforementioned static analysis technique, we identify one type of object that could be potentially helpful for exploitation. By using a sequence of system calls, we can allocate an object A in that type to a target slot. In the memory, there have already been multiple objects that share the same data structure as the object A. When using fuzz testing to explore the deallocation sites of our interest and expect the deallocation of that object A, we might encounter a situation where the fuzz testing hits the site of the deallocation (identified through static analysis) but actually free a different object sharing the same type as the object A. To tackle this issue, we develop an alternative approach that employs an under-context fuzzing approach along with dynamic data flow analysis to identify the system calls and corresponding arguments. In the following, we provide the detail of this approach.

3.2.1 Panic Anchors Setup. To be able to determine whether a system call triggers a site of our interest through fuzz testing, we instrument kernel source code and insert panic anchors (i.e., customized kernel panic functions) right behind each site of our interest. With these anchors, when kernel execution reaches the site of our interest, we can terminate kernel execution and trace back to the system calls and their arguments tied to corresponding sites. However, such a trivial approach inevitably introduces false positives. Linux kernel is a highly complex system. At the runtime, in addition to system calls under a user's control, many other kernel routines could also trigger execution to our anchor sites, e.g., exception signals from processes, interrupt signals from peripheral devices, and other kernel threads or userland processes. Without an ability to distinguish these irrelevant kernel routines, we inevitable introduce false positives (i.e., identifying the system calls not truly pertaining to object (de)allocation or pointer dereference).

In order to address this issue, we therefore augment our panic anchors with the ability to eliminate false positives. To be specific, inside each panic anchor, we first examine the kernel variable `system_state`, representing the state of the kernel execution. The panic anchor performs termination only if the value of this variable is equal to `SYSTEM_RUNNING`. This is because this value indicates the kernel is currently processing a system call requested from the userland but not responding to other kernel routines. In addition to the examination of kernel execution state, we check the field `comm` in the data structure `task_struct`. The value of this field specifies the userland program that triggers the current system call. By checking this value, we can easily determine whether the site of the interest is reached through the system call invoked by our fuzzing program and thus prevent the situation where the site of our interest is hit through other userland programs. In addition to the examination above, we store the addresses of the objects allocated. As we will specify in Section 3.2.2, along with under-context fuzzing, these addresses could help us pinpoint the system calls that truly free or dereference the objects we intentionally allocate.

3.2.2 Syscall Identification. As is mentioned above and evaluated in Section 6, when identifying system calls for object (de)allocation and pointer dereference, it is very ineffective and inefficient to exhaustively test various system call combinations. In this work, we address this problem by using the method below. First, we build a kernel call graph and perform reachability analysis from our target (de)allocation and dereference sites. Based on the result of reachability analysis, we then preserve only the system calls, which could potentially reach the sites of our interests and do not require the administrative privilege (i.e., `CAP_SYS_ADMIN`). In this work, we perform fuzzing against kernel by using a system call sequence produced from these identified system calls based on dependency information for object (de)allocation and pointer dereference. In the following, we provide the detail of our system call identification method.

Syscalls for allocation. To identify the system calls tied to critical object allocation, we perform fuzz testing against a Linux kernel. When a test case triggers an anchor site tied to object allocation, we profile each of the system calls by recording the kernel objects that every individual system call (de)allocates on the slab. In addition, we log the system calls in the test case and record their arguments accordingly. In this work, we construct a database to store these pieces of information. They are used for facilitating the manipulation of slab layout and under-context fuzzing. In the Appendix, we provide more details about the information stored in our database.

Syscalls for deallocation. When exploring the system calls tied to object deallocation, we first allocate a target kernel object by using the system calls identified through our fuzz testing. Under this context, we then log the address of that target object and perform fuzz testing. When a deallocation site is triggered by a test case generated by kernel fuzzer, we check whether the address of that deallocation object matches the address logged previously, and preserve the corresponding system calls only if a match is identified. With this design, we can ensure the system calls identified can truly deallocate the object that we intentionally allocate previously. Similarly, in addition to storing system calls and their arguments, we profile the system calls in the test case and store corresponding information in our database.

Syscalls for dereference. To identify the system calls that could dereference a function pointer through a target victim object, we first allocate that target victim object through the system calls identified. Under that context, we then perform kernel fuzzing and explore the system calls that can reach to the corresponding dummy dereference site. When a test case generated by kernel fuzzer triggers the dummy dereference site, we continue kernel execution and record each of the statements executed (i.e., execution trace) until kernel execution exits the current function. Using the execution trace, we build a use-define chain on which we identify all the function pointer dereferences, trace back through the chain and examine whether the dereference comes through the victim object that we allocate. Again, after confirming the dereference, we archive system calls and their arguments in our database.

4 LAYOUT MANIPULATION

The aforementioned database provides an attacker with the facilitation of identifying the system calls tied to the kernel object (de)allocation as well as the way to dereference corresponding function pointers. However, the kernel exploitation still lacks the ability to systematically adjust system calls so that one could obtain the desired memory layout (i.e., the challenge ❸ mentioned in Section 2.3). To address this issue, we first seek through the aforementioned database, matching a vulnerability capability with corresponding objects. For each vulnerability and object pair, we then manipulate slab by using a layout adjustment approach and thus obtain the desired layout for kernel exploitation. In the following, we describe the technical details.

4.1 Matching Vulnerability Capability

Recall that Section 2.2.2 describes four approaches commonly adopted in Linux kernel exploitation. When using these approaches, an attacker has to not only (de)allocate kernel objects but, more importantly, ensure the vulnerability gives him the ability to overwrite critical data. In order to pinpoint the kernel objects from the database that matches the capability of a vulnerability, we first model the capability of a vulnerability as well as the property of a data object. Guided by our modeling, we then design a technical approach to pairing objects with corresponding vulnerabilities.

4.1.1 Modeling Vulnerability and Object. Given a vulnerability, we describe the capability of that vulnerability by using the array $A_r[m]$. This array contains m elements, each of which specifies a memory region under an attacker's control. As is depicted in Figure 2a, each element contains a pair (l, h). It indicates the start and end offsets corresponding to the base address of a victim or spray object. In between the offsets, an attacker could overwrite data freely. As we can see from the figure, the element in the array has no overlaps (i.e., $\forall i, j,\ 1 \le i, j \le m, i \ne j,\ A_r[i].h < A_r[j].l$ or $A_r[j].h < A_r[i].l$).

In addition to the capability of a vulnerability, we describe the property of a kernel object by using the array $A_p[n]$. This array consists of n elements and each of them specifies the memory region where critical data (i.e., pointers and metadata) are actually located. As is illustrated in Figure 2b, each element in $A_p[n]$ represents an offset corresponding to the base address of a victim object. Through the offset along with the word size[2] w, we can easily pinpoint the memory area ($A_p[i] \sim A_p[i] + w$) at which the i^{th} pointer is stored.

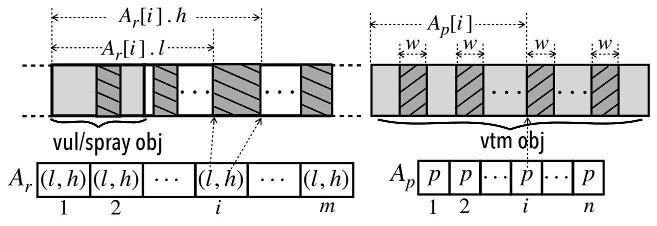
[2]The word sizes are 8 and 4 bytes for a 64-bit and 32-bit machine, respectively.

4.1.2 Pairing Vulnerability with Object. For various kernel vulnerabilities, we design different criteria and automated approaches to pair kernel objects with the capability of a kernel vulnerability. In the following, we describe the criteria. Due to the space limit, we leave the automated approach in Appendix.

OOB write. Given a vulnerability with an OOB write capability, there are two approaches to performing slab manipulation. One is to directly overwrite the metadata of a freed object and then allocate a victim object to a memory region under an attacker's control. The other is to place a victim object adjacent to the vulnerable object with the expectation of overwriting critical data (data or function pointers) residing in the victim object. For the first approach, the selection of victim objects is straightforward. The victim object could be any object that shares the same cache with the vulnerable object and encloses a pointer. For the second approach, the selection of victim objects is slightly complicated. To select the victim objects suitable for exploitation, we examine whether a memory region in the array $A_p[n]$ is included within one of the memory regions indicated by the array $A_r[m]$. We select data objects and the corresponding system calls from the database for memory layout manipulation based on the following two criteria. First, the object of our selection must be capable of being allocated at the cache same as the vulnerable object. Second, we must be able to identify an overlap successfully (see the example in Figure 3a).

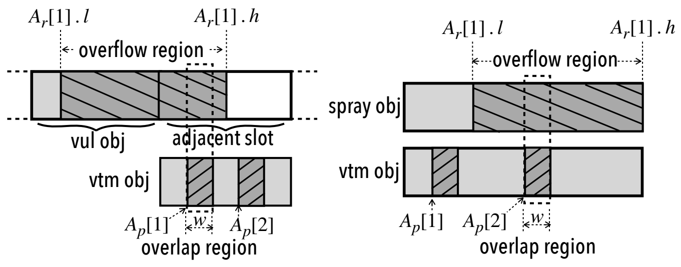
Use After Free. Given a UAF vulnerability, there are also two typical approaches to performing slab manipulation. One approach is to leverage the power of that vulnerability to modify metadata and then allocate an arbitrary victim object to the target address specified by that tampered metadata. The other approach is to overlay a spray object on top of a vulnerable object, overwrite that pointer and dereference that pointer for exploitation. For the first approach, the selection criterion of data objects is to simply have the selected victim data object share the same cache as the vulnerable object. For the second approach, the selection of data objects could be viewed as the identification of the spray objects. The criteria of spray object selection are to ❶ guarantee the spray object could be allocated in the same cache as the vulnerable object and ❷ ensure at least one field in the array $A_p[n]$ overlaps one of the memory regions indicated by the array $A_r[m]$ (see the example in Figure 3b).

Double Free. For vulnerabilities in double free, similar to the aforementioned two vulnerability types, there are also two approaches to manipulate slab. One is to overlay a victim and then a spray object on top of the vulnerable object with the expectation of overwriting a pointer residing in that victim object. The other is to overlay a spray object on the vulnerable object with the expectation of overwriting the metadata of that vulnerable object. With respect to the first approach, the selection of data objects is to find a victim object and then pair it with a spray object. Similar to the UAF, this must follow two criteria. First, victim, spray and vulnerable objects all have to share the same cache. Second, after we align all three objects, the spray object must cover at least one memory region indicated by an element of the array $A_p[n]$. Regarding the second approach, the selection task can be viewed as the identification of spray object, which has to follow the criteria – ❶ both vulnerable and spray objects share the same cache and ❷ the metadata field of the vulnerable object must overlap the spray object.



(a) Vulnerability capability (b) Kernel object property

Figure 2: Modeling vulnerability & object.



(a) Pairing OOB write (b) Pairing UAF/double free

Figure 3: Pairing vulnerability with objects.

4.2 Adjusting Slab Layout

Using the aforementioned approach and other information in our database, we can easily pinpoint the objects needed for slab layout manipulation and quickly figure out what system calls we should use to interact with the objects identified. However, as is illustrated in Section 2.3, when leveraging a system call to tamper with slab layout, it may involve side effects - allocate or deallocate many other data objects. This significantly influences an attacker's capability in obtaining the desired slab layout. As a result, we further develop a technical approach to systematically adjust the slab layout and eliminate the side effects introduced by those system calls.

Adjusting unoccupied slots. As is mentioned above, no matter which exploitation approaches one would adopt against a kernel vulnerability, he always has to allocate a target object to a corresponding freed slot (e.g., overlaying a spray object on top of a vulnerable object on a free list). In practice, it is very uncommon that an identified system call can perfectly place the target object to the target slot. In order to address this issue, we, therefore, adjust the free list chain by following the procedure below. First, we number all the unoccupied slots on the free list chain consecutively. Then, we identify the index of the target slot i (e.g., the slot where an accidentally freed object is located). By invoking the corresponding system calls to allocate a target object (e.g., a spray object), we record the index of the slot j where that target object is actually allocated. We compare the two indexes. If the index i is less than the index j ($i < j$), then we allocate $(j - i)$ objects to take over more unoccupied slots right before invoking the corresponding system calls to allocate the target object. Otherwise, we first allocate $(i - j)$ objects right before triggering the vulnerability. Second, we trigger the vulnerability and then free the objects we allocated. Finally, we invoke system calls to allocate the target object. With such a design, we can adjust the free list chain to a particular state, under which the invocation of that identified system call could perfectly position the target object to the target slot.

In this work, we accomplish the aforementioned free list chain adjustment by using system calls and corresponding objects archived in our database. More specifically, when allocating a certain number of objects, we first search our database and track down those objects that can be placed in the manipulated slab. Among the objects identified, we then choose only the objects that match the following criteria – ❶ the system calls tied to the objects do not introduce side effects, and ❷ the database archives the system calls to deallocate the object without side effects.

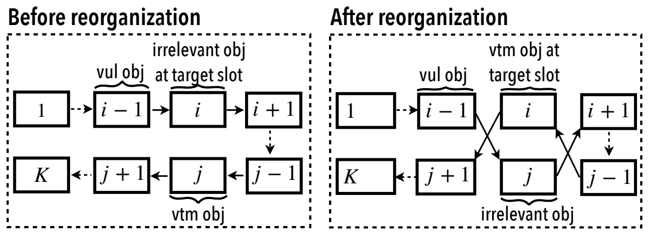


Figure 4: Reorganizing occupied slots.

It should be noted that we perform defragmentation [31, 48] right before the runtime environment preparation for a kernel vulnerability (e.g., establishing a network connection, opening files and mapping anonymous pages, etc). In addition, after defragmentation, we trigger the corresponding vulnerability and launch our slab manipulation immediately. With the first setup, we can force SLAB/SLUB allocator to create a new slab, reducing instability of our slab manipulation. With the second setup, we can minimize the influence of other kernel threads upon the slab layout.

Reorganizing occupied slots. In most cases, by following the procedure above, an adversary could place his target objects to the target slots and thus obtain the slab layout of his desire for further exploitation. However, for some vulnerabilities with an OOB write capability, knowing the way to take over the unoccupied slots is oftentimes not sufficient.

After defragmentation, the free list chain is sequentially organized or, in other words, the ordering of unoccupied slots perfectly matches their physical positions. As is depicted in Figure 4, after calling a system call to allocate a vulnerable object, the vulnerable object takes the $(i-1)^{th}$ slot on the free list chain. By analyzing the capability of the vulnerability, assume we discover the vulnerability gives the attacker the ability to perform an out-of-bounds write, which overwrites the data in its adjacent slot (i.e., the i^{th} slot). Then, following the exploitation approach mentioned in Section 2.2.2, we should allocate a victim object to the i^{th} slot. However, as is shown in Figure 4, when allocating the vulnerable object through a corresponding system call, we inevitably allocate an irrelevant kernel object, which takes the slot of our target. Assuming the irrelevant kernel object does not expose any critical data under the capability of that vulnerability, it then becomes challenging for us to exploit that vulnerability.

In order to tackle the challenge above, one instinctive reaction is to free that irrelevant object from the target slot by using a system call. However, such a method typically does not work simply because the free of the irrelevant objects may always incur the deallocation of the vulnerable objects. As a result, we develop the following method to reorganize free list chain and thus adjust occupied slots.

Before performing the reorganization, we first profile an undesired layout as follows. First, we manually extend a PoC program with the ability to call corresponding system calls to allocate a victim object right after it invokes system calls to allocate the vulnerable object. Then, we instrument this modified PoC program with `ftrace` [36] so that we can keep track of the data objects that each system call (de)allocates. By running this instrumented PoC program, we record the total number of objects that the PoC allocates on the slab, and store the index for the victim object as well as that for its desired slot. As is depicted in Figure 4, we assume a PoC program allocates K kernel objects on the slab, the victim object is located at the j^{th} slot, and the desired slot for this victim object is located at i^{th} place.

With the profiling information mentioned above, we re-order the free list chain prior to the allocation of the vulnerable object by following the procedure below. Using the system calls and data objects archived in the database, we first perform defragmentation and then allocate objects to take over K unoccupied slots on the free list chain. Second, we deallocate these objects in the reverse order except for swapping the order for the i^{th} and j^{th} objects. As is shown in Figure 4, following these allocation and deallocation operations, we reshape the free list chain. When following the same procedure above to allocate the vulnerable object and then the victim object, we can expect all the objects are successfully allocated at the originally slots except for the victim and that irrelevant objects swapped.

5 IMPLEMENTATION

We implemented the aforementioned technique and named it after `SLAKE`. As is mentioned above, our proposed technique involves both static and dynamic analysis. Therefore, `SLAKE` contains two major components taking responsibility for static and dynamic analysis, respectively. In the following, we describe critical implementation details. Due to the space limit, the readers could refer to the Appendix for more details about the database constructed by using `SLAKE`. In addition, the Appendix illustrates how `SLAKE` utilizes the database to assemble an exploitation template.

Static analysis. To perform static analysis, we compiled entire kernel code by using gllvm [18]. This gives us the ability to obtain the LLVM IR for entire Linux kernel. In this work, we developed two LLVM passes and thus performed the aforementioned static analysis against the Linux kernel IR. Our first LLVM pass is built for identifying the objects and the sites of the interest. Its implementation tracks down victim objects by using the type information preserved in LLVM IR, and gathers spray objects by searching `CallInst` to kernel I/O functions such as `copy_from_user()`. Following the design discussed in Section 3.1, for each identified object, this LLVM pass also pinpoints its (de)allocation sites as well as the name of `cache` holding that object. If an object contains a pointer, this pass further identifies its dereference chain.

Our second LLVM pass is the extension of an existing tool `KINT` [43], which takes the responsibility of kernel call graph construction. At the high level, `KINT` constructs a call graph by using static field-sensitive inter-procedural taint analysis. Through this static analysis, it could estimate the destinations for indirect calls and thus build a kernel call graph. In our implementation, we utilized `KINT` as our building block and customized it from two perspectives. First, we trim off nodes and corresponding edges in the call graph that pertain to functions in `.init.text` section. This is because these functions are no longer invoked after kernel booting and cannot be used to facilitate kernel exploitation. Second, we eliminate the edges that bridge two independent kernel modules because independent kernel modules do not have relationships between each other and the bridging edges in a call graph are false positives. In our implementation, we employ the `KConfig` file to identify such edges. To be specific, we implemented a python script that deems two modules dependent between each other only if it identified that pair of modules in the dependency attributes `depends` $\hookrightarrow$ `on` and `select`. In total, the two LLVM passes contain about 2,000 lines of C++ code, capable of running on LLVM 6.0 [34].

Dynamic analysis. As is mentioned in Section 3.2, considering the need to tie deallocation and dereference system calls to the object we intentionally allocate on the slab, we cannot directly rely upon fuzz testing. Therefore, we do not use the plugin like `KCOV` [49] to facilitate the identification of system calls because it provides no information regarding kernel objects. Instead, we wrote a GCC plugin responsible for implanting panic anchors into the Linux kernel. In addition, it instruments the Linux kernel, making it obtain the ability to store the address of each allocated object. As is described in Section 3.2, with all of these, we can utilize fuzz testing to explore the paths to these sites and thus facilitate the identification of system calls pertaining to object (de)allocation as well as function pointer dereference. In this work, we extended a Linux kernel fuzzing tool – Syzkaller [13] with the integration of Moonshine[29] – to perform fuzz testing against the instrumented Linux kernel. To be specific, in addition to the fuzzing templates provided by this tool, we introduced more fuzzing templates to Syzkaller. In total, the implementation of our Syzkaller extension along with the GCC plugin contains about 700 lines of C code.

As is described in Section 3.1, the dereference anchor sites do not always represent the sites of function pointer dereference. In order to truly pinpoint a function pointer dereference and ensure the dereference of that pointer is through a victim object, we also developed a GDB python script. Technically speaking, the GDB python script runs a Linux kernel on QEMU and sets breakpoints on the dereference anchors. Each time the script hits a breakpoint, it performs single-step execution and records each of the statements until the end of the current function. Since the statements recorded indicate the execution trace from the anchor site to the end of the current function, the python script further constructs a use-def chain, against which it performs backward data flow analysis, checks each function pointer dereference on the chain, and thus determine whether that function dereference is truly through the corresponding victim object. As part of `SLAKE`, last but not least, we implemented a ftrace util that instruments the programs generated by syzkaller. With the facilitation of this util, SLAKE logs slab activities tied to each system call and thus profiles the side effects incurred by system calls. In our implementation, the GDB script and ftrace util contain 200 lines of python code and approximately 400 lines of C code, respectively. We plan to publicly release the entire code space of SLAKE at the time of the acceptance of this work.

6 EXPERIMENT

In this section, we first introduce the setup of our experiment. Then, we evaluate how well SLAKE could identify the system calls pertaining to object (de)allocation and function pointer dereference. Finally, we showcase the effectiveness of SLAKE in exploit development.

6.1 Experiment Setup

To evaluate the effectiveness of SLAKE, we ran our experiment on Linux kernel v4.15, the latest long-term version at the time of the experiment. Against this version of Linux, we performed static analysis to build up an object database for core kernel (code covered in `defnoconfig`) as well as 32 commonly-adopted kernel modules pertaining to the file system, network, time management, inter-process communication, device drivers and security mechanisms. When compiling the Linux kernel image for dynamically exploring sites of interest in these modules, we additionally enabled other modules marked in `defconfig` plus KCOV so that kernel fuzzing can be booted normally. For each site of our interest, we set our kernel fuzzing for two hours. Throughout the entire experiment, we used 3 VM instances on a machine with the following configuration – Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHZ CPU and 64GB memory.

To obtain test cases for our evaluation, we reviewed previous research works [20, 45, 46, 48] pertaining to Linux kernel exploitation. Since the real-world vulnerabilities used in these works indicate a corpus of representative test cases, we took all of their slab-related vulnerabilities[3] to form a dataset for our evaluation. In addition, we exhaustively searched the CVE list [8] and complemented our dataset by selecting those slab-related vulnerabilities that satisfy the following criteria. ❶ There must be publicly available PoC programs demonstrating their capability in corrupting memory on the slab because SLAKE takes as input a PoC program. ❷ There must be an effective, lightweight approach to migrating these vulnerabilities into v4.15 kernel (i.e., `defconfig` plus vulnerable modules). ❸ The trigger of these vulnerabilities does not rely upon special hardware devices because this allows us to avoid the intensive labor and high cost for gathering various special hardware devices.

In total, we gathered 27 kernel vulnerabilities enclosed in our dataset. We argue this dataset is representative not only because they cover all the slab-related test cases used in the similar research but also they include other real-world vulnerabilities. To the best of our knowledge, this is the largest corpus of test cases used for evaluating research pertaining to Linux kernel exploitation. We release our code and exploits at [1] to foster future work.

[3]By slab-related vulnerabilities, we mean the vulnerabilities that corrupt slab/slub memory regions. Our SLAKE is developed for this kind of kernel vulnerabilities and therefore our test case corpus naturally excludes those non-slab-related vulnerabilities like stack or integer overflow etc.

6.2 Evaluation of Syscall Identification

Comparison between different call graphs. As is described in the section above, we build a kernel call graph by extending KINT [43], and then utilize static and dynamic analysis to identify system calls useful for exploitation. Technically speaking, in addition to our KINT-based approach, another approach to building a kernel call graph is to leverage the prototype matching used in [50]. In Table 1, we show comparison results.

First, from the column indicated by "# of v/s", we can observe that, in comparison with prototype-matching approach, our KINT-based approach could reduce the total number of kernel objects that can be potentially used for exploitation (128 vs 108). By manually examining those 20 kernel objects eliminated, we discover they are not the false positives but the kernel objects that are truly unreachable from any of the system calls. This indicates KINT-based call graph is more suitable for identifying kernel objects potentially useful for exploitation.

Second, from the column of "avg. syscall #", we can observe that our KINT-based call graph provides us with an extra benefit. That is to reduce the average number of candidate system calls reachable to the sites of our interest (257 vs 68). With this benefit, we do not need to test all 257 system calls[4] against each of the individual kernel modules when performing fuzz testing. As is specified in the column of "avg. time", this significantly reduces the time spent on dynamically finding target system calls for object (de)allocation and function pointer dereference (34 min vs 2 min). With this, we can pinpoint system calls tied to object (de)allocation and function pointer dereference in a more efficient fashion. It should be noted that, when using the prototype-matching call graph to guide fuzzing, we observe that SLAKE identifies less number of system calls. This does not imply that prototype-matching call graph cannot lead to the success of system call identification but simply means that SLAKE cannot track down corresponding system calls in less than 2 hours. Similarly, it should also be noted that SLAKE cannot find the paths to all the sites of our interest not because those sites cannot be reachable through system calls but because SLAKE needs much longer time to dynamically pinpoint those sites.

Comparison across kernel modules. From Table 1, we can also observe that, for some kernel modules (e.g., `BLOCK` and `ISO9660_FS`), SLAKE fails to identify any kernel objects useful for exploitation. This does not mean the failure of SLAKE. Rather, it is because these kernel modules are relatively small in code space (5,463 LOC on average), containing less structural variables among which none of them could potentially serve as victim or spray objects. In addition, we discover that SLAKE tracks down less number of candidate spray objects than that of candidate victim objects (104 vs 4). Among all 32 kernel modules plus the core kernel, there are only 3 modules truly contributing spray objects for kernel exploitation. To understand the shortage of spray objects, we manually examine the kernel code and explore the reason behind this observation. We discover this is because Linux kernel developers typically store data from userland on the stack and barely migrate them to the slab. While the shortage of spray objects might influence the way to perform heap spray and

[4]Linux kernel has 314 system calls. The 257 system calls are those tied to the kernel image we built.

Modules	Prototype-matching call graph				KINT-based call graph			
	Static Analysis		Dynamic Analysis		Static Analysis		Dynamic Analysis	
	# of v/s	avg. syscall #	# of alloc/free/deref	avg. time (min)	# of v/s	avg. syscall #	# of alloc/free/deref	avg. time (min)
Essential Part	39/0	257	15/3/5	46	23/0	40	16/5/6	3
AIO	3/0	257	1/0/0	23	3/0	2	2/1/2	2
ASSOCIATIVE_ARRAY	1/0	257	1/1/1	5	1/0	1	1/1/1	2
BLOCK	0/0	-	-	-	0/0	-	-	-
CGROUPS	1/0	257	1/1/1	11	1/0	1	1/1/1	2
EPOLL	3/0	257	0/0/0	-	3/0	3	1/0/0	1
EXT4_FS	8/0	257	1/0/0	5	8/0	140	3/0/0	3
FILE_LOCKING	1/0	257	0/0/0	-	1/0	4	1/0/0	2
FS_POSIX_ACL	1/0	257	1/1/1	18	1/0	19	1/1/1	2
FSNOTIFY	1/0	257	1/0/0	73	1/0	1	1/1/1	1
INET	13/1	257	0/0/0	-	13/1	3	8/2/3	5
IP_MROUTE	1/0	257	1/0/0	34	1/0	1	1/0/0	1
IPV6	6/0	257	0/0/0	-	6/0	6	1/0/0	2
ISO9660_FS	0/0	-	-	-	0/0	-	-	-
FAT_FS	0/0	-	-	-	0/0	-	-	-
JBD2	2/0	257	1/0/0	23	2/0	72	2/0/0	3
KEYS	5/2	257	3/0/1	33	5/2	1	7/0/3	4
NET	13/0	257	4/1/0	28	12/0	119	6/1/1	4
NETLABEL	1/0	257	1/0/0	41	1/0	43	1/0/0	2
PID_NS	1/0	257	1/0/0	14	1/0	1	1/1/1	1
POSIX_TIMERS	1/0	257	1/0/0	64	1/0	1	1/0/1	1
PROC_FS	3/0	257	1/0/0	16	2/0	58	3/0/0	3
SECCOMP	1/0	257	1/0/0	39	1/0	1	1/0/0	1
SECURITY_SELINUX	2/0	257	2/0/0	48	2/0	17	2/1/2	2
SND_HRTIMER	1/0	257	1/0/0	56	1/0	72	1/0/0	2
SND_SEQUENCER	3/0	257	1/0/0	23	3/0	93	3/1/1	1
SND_TIMER	2/0	257	2/0/0	50	2/0	71	2/0/0	1
SYSVIPC	2/1	257	0/0/0	-	2/1	28	3/0/0	4
TIMERFD	1/0	257	1/1/1	6	1/0	95	1/1/1	2
TTY	4/0	257	3/1/1	26	3/0	73	3/3/3	1
USB_MON	3/0	257	1/0/0	31	2/0	72	1/0/0	3
UTS_NS	1/0	257	0/0/0	-	1/0	1	1/0/0	2
Total	124/4	257	46/9/11	34	104/4	68	75/20/29	2

Table 1: The performance of SLAKE under the guidance of prototype-matching call graph and KINT-based call graph. # of v/s denotes the number of victim and spray objects identified by using static analysis; avg. syscall # represents the average number of syscalls, through reachability analysis, which can potentially reach to a site of our interest; # of alloc/free/deref indicates the number of syscalls through which one can truly allocate an object, free an object or dereference a function pointer; avg. time stands for the average amount of time that our dynamic analysis spends on finding each syscall.

Public	**victim:** file, subprocess_info, ccid, seq_file, tty_struct, ip_mc_socklist, key, sock
	spray: load_msg, SyS_add_key()
Additional	**victim:** seq_operations, perf_event_context, linux_binprm, vmap_area, kioctx_table, kioctx, assoc_array_edit, cgroup_namespace, ext4_allocation_context, ip_options_rcu, ip_mc_list, ip_sf_socklist, request_key_auth, pid_namespace, k_itimer, avc_node, sk_security_struct, snd_seq_timer, timerfd_ctx, tty_ldisc, tty_file_private
	spray: ip_options_get_from_user(), keyctl_update_key()

Table 2: The types of kernel objects identified by SLAKE. Note taht the "victim" indicates those types of objects that SLAKE not only successfully pinpoints syscall(s) to allocate but also identifies syscall(s) to dereference the pointer enclosed. The "public" indicates that these objects are also used in public exploits; the "additional" indicates the objects that have never been used in public exploits.

thus the exploitation of UAF and double free vulnerabilities, as we will show below, this does not restrict us from diversifying working exploits for real-world UAF and Double Free vulnerabilities.

Comparison between SLAKE and manual efforts. To examine how effective SLAKE is in terms of identifying kernel objects for exploitation, ideally, we would like to manually examine each system call and the objects they can (de)allocate and dereference. However, this is not possible even for a single system call. In the Linux kernel, in addition to kernel threads, software and hardware interrupts could occur at any time. They all can change the execution state of kernel and thus influence the kernel object that a system call could operate. To truly identify the set of objects pertaining to a system call manually, one must consider software/hardware interrupts and kernel threads at millions of sites over super complicated contexts. To the best of our knowledge, there has not yet been previous research or engineering efforts demonstrating the success of such manual analysis. Therefore, instead of doing manual analysis to figure out a ground-truth object set and then compare it with the results that SLAKE identifies, we utilize a different method below.

For the 27 test cases used in our experiment, we extensively searched their exploits publicly available. Then, we identified all the objects that have been used in these exploits and appeared in our examined kernel modules (32 kernel modules + 1 core kernel). Since these objects are summarized by security researchers, we treat them as the objects identified by manual efforts. In this work, we compared these objects with the ones that SLAKE identifies. In Table 2, we show the comparison results. As we can observe, SLAKE successfully identifies not only all the 10 objects from the public

CVE-ID	Type	Exploitation Methods			
		I	II	III	IV
N/A[47]	OOB	5 ($1^{\star}$)	-	-	5 (0)
2010-2959	OOB	13 ($1^{\star}$)	-	-	13 (0)
2018-6555	UAF	-	1($1^{\star}$)	-	-
2017-1000112	OOB	0 (1)	-	-	-
2017-2636	double free	-	0 (1)	-	-
2014-2851	UAF	-	0 (1)	-	-
2015-3636	UAF	-	3 (1)	-	2 (0)
2016-0728	UAF	-	3 (1)	-	4 (0)
2016-10150	UAF	-	3 (1)	-	-
2016-4557	UAF	-	2 (0)	-	-
2016-6187	OOB	-	-	-	6 (1)
2016-8655	UAF	-	3 (1)	-	-
2017-10661	UAF	-	3 (1)	-	-
2017-15649	UAF	-	3 (1)	-	-
2017-17052	UAF	-	0 (0)	-	-
2017-17053	double free	-	-	-	2 (1)
2017-6074	double free	-	3 (1)	12 (0)	-
2017-7184	OOB	10 (0)	-	-	10 (0)
2017-7308	OOB	14 (1)	-	-	14 (0)
2017-8824	UAF	-	3 (1)	-	-
2017-8890	double free	-	4 (1)	4 (0)	-
2018-10840	OOB	0 (0)	-	-	-
2018-12714	OOB	0 (0)	-	-	-
2018-16880	OOB	0 (0)	-	-	-
2018-17182	UAF	-	0 (0)	-	-
2018-18559	UAF	-	3(0)	-	-
2018-5703	OOB	0 (0)	-	-	-

Table 3: Exploitability demonstration and comparison. The numbers within the parentheses indicate the total amount of exploits publicly available and those out of the parentheses represent the amount of exploits generated through SLAKE. The star $\star$ denotes the objects used in the public exploit are not enclosed in the exploits developed through SLAKE. I ~ IV indicate the aforementioned exploitation methods pertaining to OOB, UAF, Double Free and metadata manipulation, respectively.

exploits, but also pinpoints 22 additional objects that have not yet been seen to be used for exploitation. To some extent, this implies the effectiveness of the kernel fuzzing as well as the effectiveness of SLAKE in terms of helping security researchers track down useful kernel objects.

6.3 Evaluation of Exploit Development

Comparison between public and SLAKE-generated exploits. Table 3 shows the number of unique working exploits[5] for each of the kernel vulnerabilities in our dataset.

First, we can observe that, for all 18 vulnerabilities already having public exploits, SLAKE could always find the data objects and the corresponding system calls used in that exploit except for the cases in the first 6 rows. Among all these 18 cases, there are 14 kernel vulnerabilities, against which SLAKE could also generate alternative working exploits using other distinct data objects. As we can further observe, on average, SLAKE could help security researchers develop 8 additional unique exploits. This implies SLAKE could provide a security researcher with the ability to diversify the way to perform kernel exploitation.

[5]As we mentioned in Section 2.1, we demonstrate exploitability through the control over the program counter. By working exploits, we, therefore, mean the exploits through which one could obtain control over the program counter. By distinct exploits, we mean those developed by using different victim or spray or both objects.

In addition, we can observe that, for those vulnerabilities that have not yet demonstrated exploitability through a public exploit, SLAKE could still assist security researchers in finding suitable objects to perform exploitation (e.g., CVE-2016-4557 and CVE-2017-7184 and CVE-2018-18559). While we cannot conclude un-exploitability through the lack of a public exploit demonstrating control flow hijacking, this observation – to some extent – implies that SLAKE could potentially escalate the exploitability for a target kernel vulnerability.

Some discussion of failure cases. In Table 2, we can also observe, out of 27 test cases in our dataset, there are 9 vulnerabilities that SLAKE cannot facilitate the development of their exploits.

As of the vulnerabilities pertaining to CVE-2017-1000112 and CVE-2018-5703, their PoC programs demonstrate only the capability in overwriting data inside the slab/slot pertaining to vulnerable objects. Under this capability, the exploitation of these vulnerabilities cannot use slab layout manipulation but the overwrite of critical data within the vulnerable objects. Therefore, the exploitation facilitation approaches tied to SLAKE cannot be applied to such cases. This does not dilute the value of SLAKE. This is in part because the exploitability of such vulnerabilities can be easily determined – requiring much expertise and manual efforts – but largely because among all the vulnerabilities identified over the past years there are fewer vulnerabilities that provide attackers with such a capability.

Regarding the vulnerabilities pertaining to CVE-2017-2636, CVE-2014-2851, and CVE-2018-17182, we found that, in order to perform an exploitation, we must allocate target objects to a special cache. From our database, we do not find objects suitable for such caches. Taking CVE-2014-2851 for example, we discover the vulnerable object (`group_info`) of this vulnerability is in a relatively small size (8 bytes). In our database, we fail to find a spray object that can be allocated on the cache same as that of the vulnerable object. In this work, we do not blame our proposed techniques for these cases because our experiment setup includes only those common kernel modules. In the real-world kernel images, many other kernel modules are included. They might carry the objects useful for exploitation.

With respect to vulnerabilities corresponding to CVE-2018-12714, CVE-2018-16880, CVE-2017-17052, and CVE-2018-10840, their PoC programs do not demonstrate a sufficient capability to write controllable data to a memory region. Taking CVE-2018-12714 and CVE-2018-16880 for example, we discover that, by following the control demonstrated through their PoC programs, an attacker does not have the freedom to control the value he writes to the target memory regions. This indicates, the exploitability of these vulnerabilities is heavily dependent upon the capability of these vulnerabilities. As we will discuss in Section 7, we will explore technical approaches to tackle this issue in the future.

7 DISCUSSION & FUTURE WORK

In this section, we discuss some related issues and future work.

Other OSes. To facilitate kernel exploitation, SLAKE utilizes an automated method to build a database which contains kernel objects useful for exploitation. In this work, we demonstrate this approach on Linux kernel. However, this automated approach can also be applied to facilitate exploitation for other open-source OSes (e.g., FreeBSD[11] and Android [33]). For those closed-source OSes like Windows, when utilizing our proposed approach for object identification and database construction, one has to devote energies to binary code reverse engineering.

Other allocators. In addition to a database carrying objects for exploitation, SLAKE introduces a systematic approach for slab layout manipulation. To extend this approach to other kernel allocators (e.g., SLOB allocator [25], buddy system [5]) or those used in userland (e.g., ptmalloc[12]), a modification is required. Take ptmalloc for example. Different from kernel memory management, ptmalloc might coalesce two freed chunks (like the slot in SLAB/SLUB allocator) into a single one before adding them to bins (like the cache in SLAB/SLUB allocator) for recycle. In order to customize our manipulation approach for this allocator, memory manipulation has to consider the binning and coalescing mechanism of ptmalloc. As part of our future work, we will therefore explore how to augment our memory manipulation approach for these different allocators.

Object identification. As is specified in Section 3.1, we determine a spray object by analyzing the argument of kernel functions (e.g.,`dst` of the function `copy_from_user()`. While this design could ensure SLAKE identifies most objects useful for heap spray, it could possibly ignore some special situations. For example, userland data can be copied first to the kernel stack through a system call and then migrated to the slab through a kernel function (e.g., `memcpy()`). To identify spray objects in this special situation, an accurate interprocedural data flow analysis is inevitable. Therefore, as another part of our future work, we also intend to augment SLAKE with the ability to perform inter-procedural data flow analysis and thus handle such special situations.

Other exploitation methods. In addition to the four exploitation methods discussed in Section 2, security researchers have developed other approaches for exploiting some special cases (e.g., [16, 21]) as well as heap-based use-before-initialization vulnerability. While those approaches are not as common as what we discussed in this paper, by integrating them into SLAKE, we could enrich the functionality of our technique. As a result, our future effort will also include extending SLAKE for more exploitation methods.

Vulnerability capability. As is described in Section 2, we manually extract vulnerability capabilities from a PoC program under the guidance of debugging tools. Technically speaking, this process could be potentially automated by using dynamic analysis methods such as symbolic tracing. Therefore, another line of our future work will be to explore the automation of this mechanism and enrich this functionality for SLAKE.

Recall that this work does not explore vulnerability capabilities not manifested through a PoC program. As is shown in Section 6, an inadequate capability could limit the exploitability of a vulnerability. We argue vulnerability capability exploration is a non-trivial problem, which might need the integration of various advanced techniques in program analysis. Last but not least, our future effort will therefore include exploring technical approaches to enriching capabilities for a vulnerability.

8 RELATED WORK

The works most relevant to ours include those pertaining to kernel exploitation as well as those automating the generation of exploits. In the following, we describe the existing works in these two kinds and discuss their difference from ours.

Kernel exploitation techniques. To facilitate heap spray in the process of kernel Use-After-Free exploitation, Xu et al. proposed two memory collision attack mechanisms [48]. In one of their attack mechanisms, they employed the memory recycling mechanism residing in kernel allocator. In another mechanism, they took advantage of the overlap between the physmap and the SLAB caches. To facilitate the exploitation of Use-Before-Initialization, Lu et al. proposed a deterministic stack spraying approach as well as an exhaustive memory spraying technique [24]. Both provide an attacker with the ability to manipulate data in uninitialized memory regions. To assist with the process of finding a useful exploitation primitive (e.g., control-flow hijack or arbitrary write/read), FUZE [46] proposed an automated technique that utilizes under-context fuzzing along with symbolic execution to explore exploitable machine states and thus expedite the development of working exploits for Use-After-Free vulnerabilities. To escalate the ability for vulnerabilities to bypass kernel mitigation, a recent work [45] introduces a general exploitation method which first converts a control-flow hijacking primitive into a classic stack overflow and then leverages the traditional code-reuse attack to circumvent SMEP/SMAP. In this work, we do not focus on facilitating kernel exploitation for a specific type of vulnerabilities nor developing general exploitation methods to bypass kernel mitigation. Rather, our research endeavor centers around building a technical approach to not only facilitate exploitation for various types of kernel vulnerabilities but also diversify the ways to perform memory layout manipulation and thus exploitation.

Exploit Automation techniques. There is a rich collection of research works on exploit generation automation. For example, using preconditioned symbolic execution and concolic execution techniques, Brumley et al. developed an automated approach to generate working exploits for stack overflow and format string vulnerabilities [3, 6]. Making use of symbolic tracing along with a combination of shellcode layout remediation and path kneading, Bao et al. developed ShellSwap that could automatically transplant existing shellcode and thus synthesize new shellcode for a target vulnerability [4]. With the facilitation of forward and backward taint analysis, Mothe et al. devised a technical approach to craft working exploits for simple vulnerabilities in user-mode applications [26]. Utilizing various dynamic analysis methods, a team from the UK and a team from China crafted working exploits for those heap overflow vulnerabilities residing in the userland applications [35, 44]. Using various program analysis, the Shellphish team at UCSB developed two systems (PovFuzzer and Rex) which give a security analyst the ability to turn a crash to a working exploit [38, 39, 41]. Regarding PovFuzzer, it repeatedly subtly mutates input to a vulnerable binary and observes the relationship between a crash and the input. With respect to Rex, it symbolically executes the input with the goal of jumping to shellcode or performing an ROP attack. To expedite the exploitation of vulnerabilities with the capability of out-of-bounds access, Heelan et al. utilized regression tests to obtain the knowledge of how to perform heap layout manipulation [15]. Recently, Ispoglou et al. proposed to automate data-only attack to bypass control-flow integrity in userland [19].

Given arbitrary memory write primitives, it chains basic blocks based on their semantics and thus achieves the exploitation goal.

In this work, we also develop a tool for facilitating exploit generation. However, our tool is fundamentally different from the aforementioned tools and techniques. First, rather than dealing with applications in the user space, our tool targets the Linux kernel where exploitation typically involves more complicated operations and more sophisticated memory layout. Second, rather than generating one single exploit for a target vulnerability, our tool explores all possible kernel objects and exploitation methods to output various working exploits. To some extent, this gives a security researcher the ability to diversify the ways of launching kernel exploitation and even escalate vulnerability exploitability.

9 CONCLUSION

In this paper, we built SLAKE to facilitate exploit development for kernel vulnerabilities. Technically speaking, SLAKE uses static and dynamic analysis techniques to track down data objects useful for kernel exploitation. Using SLAKE against 27 real-world kernel vulnerabilities, we show a security researcher can effectively identify kernel objects and the corresponding system calls to perform kernel exploitation. In addition, with the facilitation of SLAKE, we demonstrate a security researcher can also diversify his ways to perform kernel exploitation. For some kernel vulnerabilities, we also find that SLAKE can even escalate the exploitability for target vulnerabilities. With all of these demonstrations and findings, we conclude that static and dynamic analysis techniques could significantly empower the capability of security researchers in developing working exploits, and thus potentially benefit exploitability assessment for Linux kernel bugs.

REFERENCES

[1] 2019. Code and Exploits for SLAKE. (2019). https://github.com/chenyueqi/SLAKE.git.

[2] 0x3f97. 2018. cve-2017-8890 root case analysis. (2018). https://0x3f97.github.io/exploit/2018/08/13/cve-2017-8890-root-case-analysis/.

[3] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. 2014. Automatic Exploit Generation. *Commun. ACM* 57 (2014).

[4] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. 2017. Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits. In Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P).

[5] Daniel P. Bovet and Marco Cesati. 2010. Understanding the Linux Kernel. Elsevier.

[6] David Brumley, Pongsin Poosankam, Dawn Xiaodong Song, and Jiang Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P).

[7] Kees Cook. 2010. CVE-2010-2963 v4l compat exploit. (2010). https://outflux.net/blog/archives/2010/10/19/cve-2010-2963-v4l-compat-exploit/.

[8] The MITRE Corporation. 2019. common Vulnerability and Exposures. (2019). https://cve.mitre.org/cve/.

[9] SSD Secure Disclosure. 2017. SSD Advisory – Linux Kernel AF_PACKET Use-After-Free. (2017). https://ssd-disclosure.com/archives/3484.

[10] Jake Edge. 2014. The kernel address sanitizer. (2014). https://lwn.net/Articles/612153/.

[11] The FreeBSD Foundation. 2019. The FreeBSD Project. (2019). https://www.freebsd.org/.

[12] Wolfram Gloger. 2006. Wolfram Gloger's malloc homepage. (2006). http://www.malloc.de/en/.

[13] google. 2019. syzkaller - kernel fuzzer. (2019). https://github.com/google/syzkaller.

[14] Samuel Grob. 2014. Linux local root exploit for CVE-2014-0038. (2014). https://github.com/saelo/cve-2014-0038.

[15] S Heelan, T Melham, and D Kroening. 2018. Automatic Heap Layout Manipulation for Exploitation. In Proceedings of the 27th USENIX Security Symposium (USENIX Security).

[16] Jann Horn. 2018. A cache invalidation bug in Linux memory management. (2018). https://googleprojectzero.blogspot.com/2018/09/a-cache-invalidation-bug-in-linux.html.

[17] Ralf Hund, Thorsten Holz, and Felix C. Freiling. 2009. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In Proceedings of the 18th USENIX Security Symposium (USENIX Security).

[18] ianamason. 2019. Whole Program LLVM: wllvm ported to go. (2019). https://github.com/SRI-CSL/gllvm.

[19] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18).

[20] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. 2014. ret2dir: Rethinking Kernel Isolation. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security).

[21] Andrey Konovalov. 2017. Exploiting the Linux kernel via packet sockets. (2017). https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[22] Andrey Konovalov. 2017. A proof-of-concept local root exploit for CVE-2017-6074. (2017). https://github.com/xairy/kernel-exploits/blob/master/CVE-2017-6074/poc.c.

[23] Lexfo. 2018. CVE-2017-11176: A step-by-step Linux Kernel exploitation. (2018). https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part1.html.

[24] Kangjie Lu, M Walter, David Pfaff, and Stefan Nürnberger and Wenke Lee and Michael Backes. 2017. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS).

[25] Matt Mackall. 2005. slob: introduce the SLOB allocator. (2005). https://lwn.net/Articles/157944/.

[26] Rohit Mothe and Rodrigo Rubira Branco. 2016. DPTrace: Dual Purpose Trace for Exploitability Analysis of Program Crashes. In Black Hat USA Briefings.

[27] Vitaly Nikolenko. 2016. CVE-2014-2851 group_info UAF Exploitation. (2016). https://cyseclabs.com/page?n=02012016.

[28] Jon Oberheide. 2010. Linux Kernel CAN SLUB Overflow. (2010). https://jon.oberheide.org/blog/2010/09/10/linux-kernel-can-slub-overflow/.

[29] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In Proceedings of the 27th USENIX Security Symposium (USENIX Security).

[30] Christopher M. Penalver. 2016. How to triage bugs. (2016). https://wiki.ubuntu.com/Bugs/Importance.

[31] Enrico Perla and Massimiliano Oldani. 2010. A Guide to Kernel Exploitation. Elsevier.

[32] Alexander Popov. 2017. CVE-2017-2636: exploit the race condition in the n_hdlc Linux kernel driver bypassing SMEP. (2017). https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html.

[33] Android Open Source Project. 2019. Common Android Kernel Tree. (2019). https://android.googlesource.com/kernel/common/.

[34] LLVM Project. 2019. LLVM 6.0.0 Release Notes. (2019). http://releases.llvm.org/6.0.0/docs/ReleaseNotes.html.

[35] Dusan Repel, Johannes Kinder, and Lorenzo Cavallaro. 2017. Modular Synthesis of Heap Exploits. In ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS).

[36] Steven Rostedt. 2009. Debugging the kernel using Ftrace. (2009). https://lwn.net/Articles/365835/.

[37] Chris Salls. 2017. Exploiting CVE-2017-5123 with full protections. SMEP, SMAP, and the Chrome Sandbox! (2017). https://salls.github.io/Linux-Kernel-CVE-2017-5123/.

[38] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS).

[39] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK:(State of) The Art of War: Offensive Techniques in Binary Analysis. In Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P).

[40] Richard M. Stallman. 2019. GNU Debugger. (2019). https://www.gnu.org/software/gdb/.

[41] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS).

[42] Dmitry Vyukov. 2018. syzbot and the tale of thousand kernel bugs. (2018). https://events.linuxfoundation.org/wp-content/uploads/2017/11/

Syzbot-and-the-Tale-of-Thousand-Kernel-Bugs-Dmitry-Vyukov-Google.pdf.

[43] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Improving Integer Security for Systems with KINT. In Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI).

[44] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, BingChang Liu, Kaixiang Chen, and Wei Zou. 2018. Revery: From Proof-of-Concept to Exploitable. In Proceedings of the 25nd ACM SIGSAC Conference on Computer and Communications Security (CCS).

[45] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In Proceedings of the 28th USENIX Security Symposium (USENIX Security).

[46] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Wei Zou, and Xiaorui Gong. 2018. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In Proceedings of the 27th USENIX Security Symposium (USENIX Security).

[47] ww9210. 2019. exploit code for a bpf heap overflow vulnerability. (2019). https://github.com/ww9210/kernel4.20_bpf_LPE.

[48] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS).

[49] Masahiro Yamada and Jani Nikula. 2019. kcov:code coverage for fuzzing. (2019). https://github.com/torvalds/linux/blob/master/Documentation/dev-tools/kcov.rst.

[50] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits. In Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS).

10 APPENDIX

A AUTOMATED APPROACH TO PAIRING VULNERABILITY WITH OBJECT

We design an automated approach to pairing a kernel vulnerability with objects. The algorithm 1 describes the detail of this automated approach. As we can see in Algorithm 1, the input to the algorithm includes the type of the vulnerability (T), the name of cache for the vulnerable object (C), a Flag (F) indicating – for a UAF vulnerability – whether it is feasible to modify the meta-header after the vulnerable object is freed, an array A_r indicating memory regions under an attacker's control, and an array A_p indicating where critical data are located. In addition, we can observe, the output for the algorithm is a set of 3-tuple which are a spray object, a victim object, and the offset between the vulnerable and the victim object. Here, the offset describes desired slab layout. Considering that attackers can move a victim object to any slots on the slab, we further define an operation $\oplus$ to adjust the base address of critical data in the victim object. For example, when a victim object is slid two slots rightward, we modify A_p as $A_p \oplus (2 \times SZ)$ which adds $2 \times SZ$ to all pointer offets in A_p.

Algorithm 1 Matching Vulnerability Capability

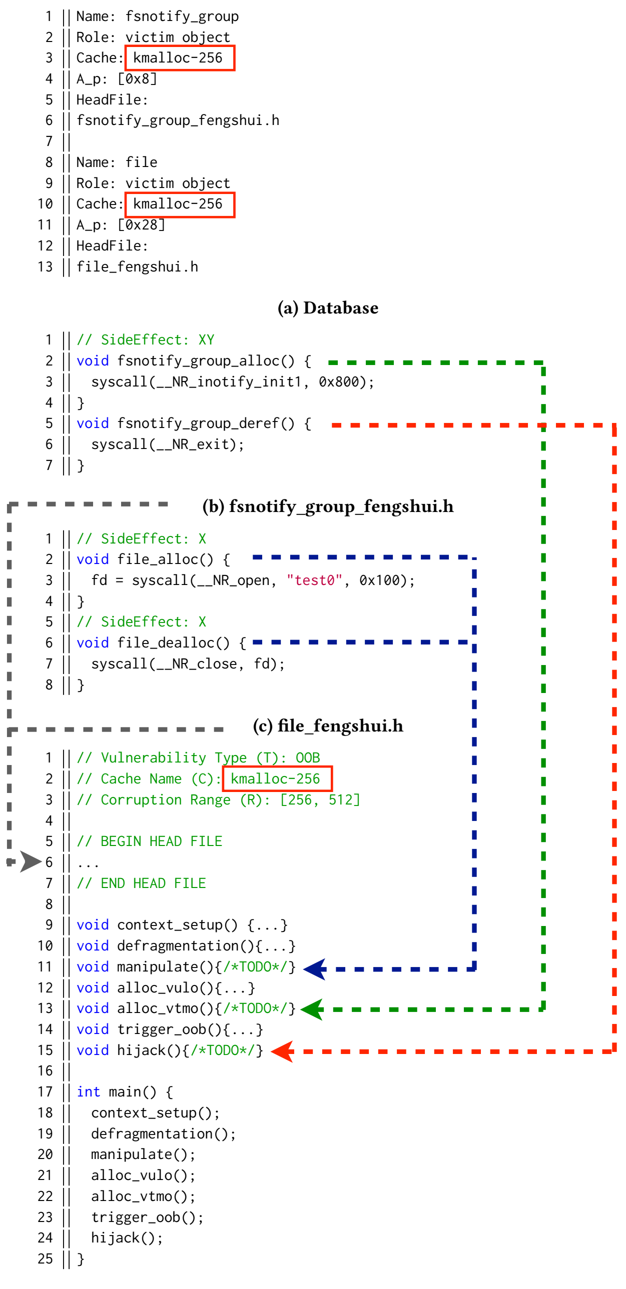
```
Input: $T$: Vulnerability Type; $C$: VulO Cache;
$F$: Meta Flag; $A_p$: Pointer Array; $A_r$: Range Array
Output: $S$: Set of 3-tuple <SprayObj, VtmObj, Offset>
procedure MatchVulCap($T$, $C$, $F$, $A_p$, $A_r$)
    if $T$ == UAF then
        $S$ = MatchUAF($C$, $A_p$)
    else if $T$ == double_free then
        $S$ = MatchDF($C$)
    else if $T$ == OOB then
        $S$ = MatchOOB($C$, $A_r$)
    return $S$

procedure MatchUAF($C$, $F$, $A_p$)
    $S = \emptyset$
    for all SprO $r1$, VtmO $r2$ using $C$ in database do
        $A_r$ = data range in $r1$
        if IsMatch($A_r$, $A_p$) then
            $S = S \cup <R_1, r1, \square, \square>$
        if $F$ then
            $S = S \cup <R_4, \square, r2, \square>$
    return $S$

procedure MatchDF($C$)
    for all SprO $r1$, VtmO $r2$ using $C$ in database do
        $A_r$ = data range in $r1$
        $A_p$ = pointer in $r2$
        if IsMatch($A_r$, $A_p$) then
            $S = S \cup <R_2, r1, r2, \square>$
        $A'_p = [0]$
        if IsMatch($A_r$, $A'_p$) then
            $S = S \cup <R_4, r1, r2, \square>$
    return $S$

procedure MatchOOB($C$, $A_r$)
    $SZ$ = size of slot in $C$
    for all $i, -10 \le i \le 10, i \ne 0$ do
        for all VtmO $r2$ using $C$ in database do
            $A_p$ = pointers in $r2$
            $A'_p = A_p \oplus (i \times SZ)$ // slide VtmO
            if IsMatch($A_r$, $A'_p$) then
                $S = S \cup <R_3, \square, r2, i>$
        $A_p = [i \times SZ]$
        if IsMatch($A_r$, $A_p$) then
            $S = S \cup <R_4, \square, r2, i>$
    return $S$

procedure IsMatch($A_r$, $A_p$)
    $m$ = length of $A_r$
    $n$ = length of $A_p$
    for all $i, j, 1 \le i \le m, 1 \le j \le n$ do
        if $A_r[i].l \le A_p[j] < A_p[j] + w \le A_r[i].h$ then
            return True
    return False
```

B DATABASE AND ITS USAGE

As is mentioned in this paper, using static and dynamic analysis, we construct a database in which we store information useful for kernel exploitation. In our implementation, we save these information in a text file. Here, we specify the information stored in the database and illustrate how to use the information to perform a slab layout manipulation by using an example shown in Figure 5.

As we can see in the figure, `fsnotify_group` is a victim object in cache `kmalloc-256`. Its critical data is in the offset `0x8`. Its allocation, free and dereference system calls are stored in the head file `fsnotify_group_fengshui.h`. In this head file, `fsnotify_group` can be allocated by system call `inotify_init1` and critical data can be dereferenced by system call `exit`. Since `inotify_init1` allocates an additional object to the cache `kmalloc-256` after `fsnotify_group`, we record the side effect tied to the allocation as `XY` where `X` denotes `fsnotify_group` and `Y` indicates an irrelevant object.

To manipulate slab layout by using the database above, we search the database under the guidance of the method introduced in Section 4.1. We can find that `fsnotify_group` is a perfect match because it can be allocated to the cache `kmalloc-256` and its critical data overlapping with the corruption region. Since we have an annotated PoC program in hand, demonstrating the capability of the vulnerability, we can include `fsnotify_group_fengshui.h` to the PoC and then insert a call to function `fsnotify_group_alloc()` in function `alloc_vtmo` ↪ `()` and a call to `fsnotify_group_deref()` in `hijack()` respectively. Using ftrace results, we however can find that the target slot is occupied by an irrelevant object. To address this issue, we therefore can reorganize occupied slots by following the method introduced in Section 4.2. For this case, we can complete reorganization by inserting (de)allocation of `file` in a specific order to the function `manipulate()` in the annotated PoC. This is because the object `file` shares the same cache with `fsnotify_group` and its (de)allocation involves no side effect.



```
Name: fsnotify_group
Role: victim object
Cache: kmalloc-256
A_p: [0x8]
HeadFile:
fsnotify_group_fengshui.h

Name: file
Role: victim object
Cache: kmalloc-256
A_p: [0x28]
HeadFile:
file_fengshui.h
```

(a) Database

```
// SideEffect: XY
void fsnotify_group_alloc() {
  syscall(__NR_inotify_init1, 0x800);
}
void fsnotify_group_deref() {
  syscall(__NR_exit);
}
```

(b) fsnotify_group_fengshui.h

```
// SideEffect: X
void file_alloc() {
  fd = syscall(__NR_open, "test0", 0x100);
}
// SideEffect: X
void file_dealloc() {
  syscall(__NR_close, fd);
}
```

(c) file_fengshui.h

```
// Vulnerability Type (T): OOB
// Cache Name (C): kmalloc-256
// Corruption Range (R): [256, 512]

// BEGIN HEAD FILE
...
// END HEAD FILE

void context_setup() {...}
void defragmentation(){...}
void manipulate(){/*TODO*/}
void alloc_vulo(){...}
void alloc_vtmo(){/*TODO*/}
void trigger_oob(){...}
void hijack(){/*TODO*/}

int main() {
  context_setup();
  defragmentation();
  manipulate();
  alloc_vulo();
  alloc_vtmo();
  trigger_oob();
  hijack();
}
```

(d) The PoC with Annotation

Figure 5: An example illustrating the database and its usage