The Pennsylvania State University
The Graduate School

INVESTIGATING EXPLOITABLE DESIGN PATTERNS FOR MORE ADVANCED PROTECTION DESIGN

A Dissertation in
Information Sciences and Technology
by
Yueqi Chen

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2022

The dissertation of Yueqi Chen was reviewed and approved by the following:

Xinyu Xing
Associate Professor of Computer Science, Northwestern University
Dissertation Advisor
Chair of Committee

Peng Liu
Professor of Information Sciences and Technology

Dinghao Wu
Professor of Information Sciences and Technology

Trent Jaeger
Professor of Computer Science and Engineering

Vasileios P. Kemerlis
Assistant Professor of Computer Science, Brown University
Special Member

Mary Beth Rosson
Professor of Information Sciences and Technology
Director of Graduate Program

Abstract

For a long time, due to the lack of systematic research on exploitable design patterns, software systems are not as secure as expected. With the discovery of massive vulnerabilities and the shortage of workforce, developers prioritize vulnerability remediation based on hype and media attention. They are unable to assess the severity of vulnerabilities because they are clueless about the implied exploitable design patterns. As a result, broadly adopted software inevitably contains exploitable but unpatched vulnerabilities. To prevent unpatched vulnerabilities from causing foreseen damage, the industry scrambles to design and deploy various defense schemes. However, as there is short of scientific methods for quantifying how well these defenses can mitigate exploitable design patterns, the adoption of defense mechanisms dramatically relies upon analysts' subjective judgment. A defense method with relatively high overhead might be ruthlessly abandoned, despite providing substantial protection for software systems.

To make meaningful progress, I propose to shift our research to study exploitable design patterns in a systematic approach. This approach provides security analysts with the ability to quantify the impact of an exploitable design pattern and thus facilitate the development and assessment of corresponding defense solutions. In this dissertation, I focus on investigating exploitable design patterns and designing advanced protection mechanisms in the operating system kernels. More specifically, I start from developing techniques to explore corruption capability of kernel vulnerabilities. Then, I leverage static analysis techniques and dynamic analysis techniques to identify design patterns that are commonly adopted in various OS kernels. Through the explored corruption capability, I evaluate the exploitability of the design pattern. Following this, I facilitate the evaluation by manipulating slab layout to obtain exploitable primitives. These primitives are further examined against existing defenses to demonstrate their limitations. This investigation procedure motivates the design of a new protection mechanism that can harden OS kernel on-demand and on-the-fly with negligible overhead.

In the future, I plan to further advance the exploitable design pattern investigation approach to make it a fundamental part of the entire software development lifecycle (SDLC). In pursuit of this goal, I will enrich techniques to deal with more attack forms under new contexts. Following this, I intend to optimize and re-construct existing defenses, extending them to mitigate exploitable design patterns in new contexts, under the guidance of quantitative evaluation.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First, I would like to express my sincere gratitude to Dr. Xinyu Xing for being so supportive, not only in research but also in personal life. Choosing Dr. Xing as my advisor is the rightest decision that I have made in the past five years. I learned from him critical thinking, pragmatism, and open-mindedness which are of great value to my future career development and exploration of remaining life.

Second, I want to thank Dr. Peng Liu for his valuable comments on the future direction part of my job talk which helps me better position my research works. I also want to say thanks to my other committee members, Dr. Dinghao Wu, Dr. Trent Jaeger, and Dr. Vasileios P. Kemerlis for their insightful and inspiring questions during my comprehensive examination and dissertation defense.

Third, I want to thank Wenbo Guo for his help as my closest cohort and guidance as a senior fellow. I am very grateful to my other peers and collaborators during my Ph.D. study for their support, enlightening and modeling. They are Jun Xu, Wei Wu, Dongliang Mu, Zhenpeng Lin, Yuhang Wu, Hua Wei, Dongkuang Xu, Dongsheng Luo, Gang Wang, Tiffany Bao, Yan Shoshitashivili, Adam Doupé, Kyle Zeng, Michael V Le, Dan Williams, Somesh Jha, Shengjian Guo, Kang Li, Yueh-Hsun Lin, Jimmy Su, Yuanyuan Shi, Peng Li, and many other nice people with whom I have connections.

I lived at State College for more than four years which I shall never forget. I spent the last half year of my PhD study in Evanston where I took dozens of interviews and went through my hardest time. Ade!

The research presented in this dissertation was supported in part by NSF grant CNS-1718459, ONR grant N00014-20-1-2008, and the IBM Ph.D. Fellowship Award. The findings and conclusions do not necessarily reflect the view of the funding agency.

Dedication

To my mother, Fengwei Fu, and my father, Wenwei Chen for their unconditional love and support. I can never make it without their encouragement.

Chapter 1 Introduction

In 2015, there was a use-after-free vulnerability (i.e., `CVE-2015-0057`) reported in Microsoft Windows system. This vulnerability was exploited in the wild by attackers using a data structure named `struct tagPROPList` [2]. The attackers leverage this structure to achieve local privilege escalation, becoming an administrator from a common user. This vulnerability was fixed after the disclosure while the design of `struct tagPROPList` was left in the Windows system. Two years later, in 2017, a heap buffer overflow vulnerability (i.e., `CVE-2017-7184`) was reported in the Linux kernel [3]. The attackers leverage another structure (e.g., `struct xfrm_replay_state_esn`) to exploit this heap buffer overflow and finally leaked sensitive information. Similarly, this vulnerability was also fixed and the `struct xfrm_replay_state_esn` was kept in the Linux kernel.

Although the two attacks targeted at different vulnerability types in two distinct systems, the two structures `struct tagPROPList` implemented by Microsoft team and `struct xfrm_replay_state_esn` developed by open source community, share something in common: both structures have an integer field and a buffer field. The integer field indicates the size of the buffer which is flexible and determined dynamically. By overwriting the length field, the attackers can mislead the kernel to miscalculate the size of the buffer and thus realize illegal read and write.

The two structures described above exemplify the concept of exploitable design patterns. That is, different systems could follow the same design pattern which can be misused to exploit various vulnerabilities that have been reported or will be reported in the future. For a long time, due to the lack of systematic research on exploitable design patterns, software systems are not as secure as expected. With the discovery of massive vulnerabilities and the shortage of workforce, developers prioritize vulnerability remediation based on hype and media attention. They are unable to assess the severity of vulnerabilities because they are clueless about the implied exploitable design patterns.

As a result, broadly adopted software inevitably contains exploitable but unpatched vulnerabilities. To prevent unpatched vulnerabilities from causing foreseen damage, the industry scrambles to design and deploy various defense schemes. However, as there is short of scientific methods for quantifying how well these defenses can mitigate exploitable design patterns, the adoption of defense mechanisms dramatically relies upon analysts' subjective judgment. A defense method with relatively high overhead might be ruthlessly abandoned, despite providing substantial protection for software systems. This results in the remaining existence of exploitable design patterns which will continue to be misused by attackers in the future.

To make meaningful progress, I propose to shift our research to study exploitable design patterns in a systematic approach. This approach provides security analysts with the ability to quantify the impact of an exploitable design pattern and thus facilitate the development and assessment of corresponding defense solutions. From a technical perspective, my proposed method contains two critical steps.

Investigating Exploitable Design Patterns. The first part of this dissertation demonstrates how to identify design patterns that are commonly adopted by different vendors to implement various systems. After this, we develop techniques to evaluate the exploitability of design patterns by exploring vulnerability capabilities, manipulating memory layout, and bypassing existing mitigations. Through investigation, we can learn how exploitable the design pattern is and what hinders existing protections from mitigating it.

Constructing Advanced Protection Design. In the second part of this dissertation, we concentrate on the design of a protection that is universal enough to work for all heap-based exploitation techniques. The key idea of this design directly stems from the investigation of exploitable design patterns, which justifies the feasibility of proposed methodology and sheds light on the future directions.

1.1 Organization

This dissertation makes several contributions, most with published results in academic conferences. The following briefly summarizes these works and acts as an outline for the remaining of this dissertation.

We propose to investigate exploitable design patterns so as to design more advanced software system protections. First, Chapters 2 to 5 focus on extracting design patterns

and developing techniques to evaluate their exploitability. In Chapter 6, we concentrate on the design of a protection mechanism that can be enforced on-demand against heap-based exploitation.

Chapter 2 presents `FUZE`, a new framework to explore the capability of kernel vulnerabilities, especially Use-After-Free errors. he design principle behind this technique is that we expect the ease of crafting an exploit could augment a security analyst with the ability to evaluate the exploitability of a kernel UAF vulnerability. Technically, `FUZE` utilizes kernel fuzzing along with symbolic execution to identify, analyze and evaluate the system calls valuable and useful for kernel UAF exploitation. In addition, it leverages dynamic tracing and an off-the-shelf constraint solver to guide the manipulation of vulnerable object. To demonstrate the utility of `FUZE`, we implement `FUZE` on a 64-bit Linux system by extending a binary analysis framework and a kernel fuzzer. Using 15 realworld kernel UAF vulnerabilities on Linux systems, we then demonstrate `FUZE` could not only escalate kernel UAF exploitability but also diversify working exploits. In addition, we show that `FUZE` could facilitate security mitigation bypassing, making exploitability evaluation less challenging and more efficient.
Chapter 2 appeared in the *Proceedings of the 27th USENIX Security Symposium (USENIX Security 2018)* [4]

Chapter 3 presents `ELOISE`. In chapter 2, we developed `FUZE` to explore the corruption capability of vulnerabilities which can be further used to perform exploitation and bypass protections. For example, security researchers have demonstrated an exploitation method that utilizes the characteristic of elastic kernel objects to bypass KASLR, disclose stack-/heap cookies, and even perform arbitrary read in the kernel. While this exploitation method is considered a commonly adopted approach to disclosing critical kernel information, there is no evidence indicating that elastic kernel object is an exploitable design patterns. It is because the effectiveness of this exploitation method is demonstrated only on anecdotal kernel vulnerabilities. It is unclear whether such a method is useful for a majority of kernel vulnerabilities.

To answer this question, we propose a systematic approach. It utilizes static/dynamic analysis methods to pinpoint elastic kernel objects and then employs constraint solving to pair them to corresponding kernel vulnerabilities. In this work, we implement our proposed method as a tool - `ELOISE`. Using this tool on three popular OSes (`Linux`, `FreeBSD`, and `XNU`), we discover that elastic object is a design pattern as it is pervasive in general caches. Evaluating the effectiveness of these elastic objects on 40 kernel vulnerabilities across

three OSes, we observe that they can enable most of the vulnerabilities to bypass KASLR and heap cookie protector. Besides, we also observe that these elastic objects can even escalate the exploitability of some vulnerabilities allowing them to perform arbitrary read in the kernel. As such, we conclude that elastic object is an exploitable design pattern. Motivated by this, we introduce a new defense mechanism to mitigate the threat of elastic kernel objects. We prototype our defense mechanism on Linux, showing this mechanism introduces negligible overhead. In Chapter 6, we will further developed this protection to make it more advanced and universal.
Chapter 3 appeared in the *Proceedings of 27th ACM SIGSAC Conference on Computer and Communications Security (CCS 2020) [5]*

Chapter 4 presents SLAKE. To evaluate the exploitability of investigated design patterns, a security analyst usually has to manipulate slab and thus demonstrate the capability of obtaining exploitable primitive (e.g., the control over a program counter or performing arbitrary read/write). 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 the evaluation. 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.
Chapter 4 appeared in the *Proceedings of 26th ACM SIGSAC Conference on Computer and Communications Security (CCS 2019) [6]*

Chapter 5 presents KEPLER. In chapter 4, we developed SLAKE which leverages the design pattern to obtain exploitable primitives such as control-flow hijacking primitives. However, obtaining primitives is insufficient to evaluate exploitability because of widely deployed exploit mitigations in an off-the-shelf modern Linux kernel. We therefore

propose KEPLER to facilitate exploit generation by automatically generating a "single-shot" exploitation chain. KEPLER accepts as input a control-flow hijacking primitive and bootstraps any kernel ROP payload by symbolically stitching an exploit chain taking advantage of prevalent kernel coding style and corresponding gadgets. Comparisons with previous automatic exploit generation techniques and previous kernel exploit techniques show KEPLER effectively facilitates evaluation of control-flow hijacking primitives in the Linux kernel.
Chapter 5 appeared in the *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019) [7]*

Chapter 6 presents HotBPF. In previous chapters, we developed tools to investigate exploitable design patterns and demonstrate the limitations of existing protection mechanisms. More specifically, we developed FUZE to explore the capability of vulnerabilities which facilitates exploitation, ELOISE to identify elastic structure as a design pattern and demonstrate that its exploitability given different vulnerability capabilities, SLAKE to facilitate exploitability evaluation by manipulating memory layout and obtaining exploitable primitives. Finally, we proposed KEPLER to evaluate existing mitigations using the obtained primitives. The results, unfortunately, show that existing protections fail to prevent exploitation in front of various potential exploitable design patterns. Although in Chapter 3, we designed an isolation-based mitigation to protect the integrity of elastic structures, it is not general enough to be effective for other exploitable design patterns. In this chapter, we introduce a new defense named after HotBPF to isolate corruption and thus protect the Linux kernel from attacks that start from memory corruption and take advantage of a variety of exploitable design patterns.

Oftentimes there can be a large window between a kernel vulnerability disclosure and its remediation, leaving the system open for exploitation. We present the design of a mechanism named HotBPF that can protect the Linux kernel from memory exploitation during this time window. This protection mechanism has the following advantages. First, it can automatically deploy protection immediately after the vulnerability disclosure. Second, this protection can be enabled on-the-fly which means there is no need to disrupt critical services to recompile and reboot the system. Third, compared with existing memory exploitation mitigations in the Linux kernel, this protection can offer stronger security protection by preventing cross-cache exploitation. Fourth, this protection is independent of hardware features and hypervisor virtualization. So it can be widely deployed in a variety of scenarios (e.g., embedded systems, desktops, and cloud servers). Last but not least, this protection is lightweight as it introduces an overall 2% - 3%

performance overhead.
Chapter 6 is under submission to the *Proceedings of the 32nd USENIX Security Symposium (USENIX Security 2023)*

Chapter 7 concludes this dissertation and discuss future research directions.

Chapter 2
FUZE: Start from Exploring Kernel Vulnerability Capability

As mentioned in Chapter 1, vulnerability exploitation starts from memory corruption. In practice, however, the corruption capability of reported vulnerability is seldom evaluated in a systematic approach, which hinders the accurate and comprehensive investigation of exploitable design patterns. In this paper, we therefore propose `FUZE`, a new framework to explore the capability of kernel vulnerabilities, especially Use-After-Free (UAF) vulnerabilities. The design principle behind this technique is that we expect the ease of crafting an exploit could augment a security analyst with the ability to evaluate the exploitability of a kernel UAF vulnerability. Technically, `FUZE` utilizes kernel fuzzing along with symbolic execution to identify, analyze and evaluate the system calls valuable and useful for kernel UAF exploitation. In addition, it leverages dynamic tracing and an off-the-shelf constraint solver to guide the manipulation of vulnerable object. To demonstrate the utility of `FUZE`, we implement `FUZE` on a 64-bit Linux system by extending a binary analysis framework and a kernel fuzzer. Using 15 realworld kernel UAF vulnerabilities on Linux systems, we then demonstrate `FUZE` could not only escalate kernel UAF exploitability but also diversify working exploits. In addition, we show that `FUZE` could facilitate security mitigation bypassing, making exploitability evaluation less challenging and more efficient.

2.1 Introduction

It is very rare for a software team to ever have sufficient resources to address every single software bug. As a result, software vendors such as Microsoft [8] and Ubuntu [9] design and develop various strategies for prioritizing their remediation work. Of all of those

strategies, remediation prioritization with exploitability is the most common one, which evaluates a software bug based on ease of its exploitation. In practice, determining the exploitability is however a difficult, complicated and lengthy process, particularly for those Use-After-Free (UAF) vulnerabilities residing in OS kernels.

Use-After-Free vulnerabilities [10] are a special kind of memory corruption flaw, which could corrupt valid data and thus potentially result in the execution of arbitrary code. When occurring in an OS kernel, they could also lead to privilege escalation [11] and critical data leakage [12]. To exploit such vulnerabilities, particularly in an OS kernel, an attacker needs to manually pinpoint the time frame that a freed object occurs (i.e., vulnerable object) so that he could spray data to its region and thus manipulate its content accordingly. To ensure that the consecutive execution of the OS kernel could be influenced by the data sprayed, he also needs to leverage his expertise to manually adjust system calls and corresponding arguments based on the size of a freed object as well as the type of heap allocators. We showcase this process through a concrete example in Section 2.2.

To facilitate exploitability evaluation, an instinctive reaction is to utilize the research works proposed for exploit generation, in which program analysis techniques are typically used to analyze program failures and produce exploits accordingly (e.g., [13–16]). However, the techniques proposed are insufficient for the problem above. On the one hand, this is due to the fact that the program analysis techniques used for exploit generation are suitable only for simple programs but not the OS kernel which has higher complexity and scalability. On the other hand, this is because their technical approaches mostly focus on stack or heap overflow vulnerabilities, the exploitation of which could be possibly facilitated by simply varying the context of a PoC program, whereas the exploitation of a UAF vulnerability requires the spatial and temporal control over a vulnerable object, with the constraints of which a trivial context variation typically does not benefit exploitability exploration.

In this work, we propose `FUZE`, an exploitation frame- work to evaluate the exploitability of kernel Use-After-Free vulnerabilities. In principle, this framework is similar to the technical approaches proposed previously, which achieves exploitability evaluation by automatically exploring the exploitability of a vulnerability. Technically speaking, our framework however follows a completely different design, which utilizes a fuzzing technique to diversify the contexts of a kernel panic and then leverages symbolic execution to explore exploitability under different contexts.

To be more specific, our system first takes as input a PoC program which does

not perform exploitation but causes a kernel panic. Then, it utilizes kernel fuzzing to explore various system calls and thus to mutate the contexts of the kernel panic. Under each context pertaining to a distinct kernel panic, FUZE further performs symbolic execution with the goal of tracking down the primitives potentially useful for exploitation. To pinpoint the primitives truly valuable for exploiting a UAF vulnerability and even bypassing security mitigation, FUZE summarizes a set of exploitation approaches commonly adopted, and then utilizes them to evaluate primitives accordingly. In Section 2.3, we will describe more details about this exploitation framework. Different from the existing techniques (e.g., [13–16]), the proposed exploitation framework is not for the purpose of fully automating exploit generation. Rather, it facilitates exploitability evaluation by easing the process of exploit crafting. More specifically, FUZE facilitates exploit crafting from the following aspects.

First, it augments a security analyst with the ability to automate the identification of system calls that he needs to take advantages for UAF vulnerability exploitation. Second, it allows a security analyst to automatically compute the data that he needs to spray to the region of the vulnerable object. Third, it facilitates the ability of a security analyst to pinpoint the time frame when he needs to perform heap spray and vulnerability exploitation. Last but not least, it provides security analysts with the ability to achieve security mitigation bypassing.

As we will show in Section 2.6, with the facilitation from all the aforementioned aspects, we could not only escalate kernel UAF exploitability but also diversify working exploits from various kernel panics. In addition, we demonstrate FUZE could even help security analysts to craft exploits with the ability to bypass broadly-deployed security mitigation such as SMEP and SMAP. To the best of our knowledge, FUZE is the first exploitation framework that can facilitate exploitability evaluation for kernel Use-After-Free vulnerabilities.

In summary, this work makes the following contributions.

- We designed FUZE, an exploitation framework that utilizes kernel fuzzing along with symbolic execution to facilitate kernel UAF exploitation.
- We implemented FUZE to facilitate the process of exploit generation by extending a binary analysis framework and a kernel fuzzer on a 64-bit Linux system.
- We demonstrated the utility of FUZE in crafting working exploits as well as facilitating security mitigation circumvention by using 15 real world UAF vulnerabilities in Linux kernels.

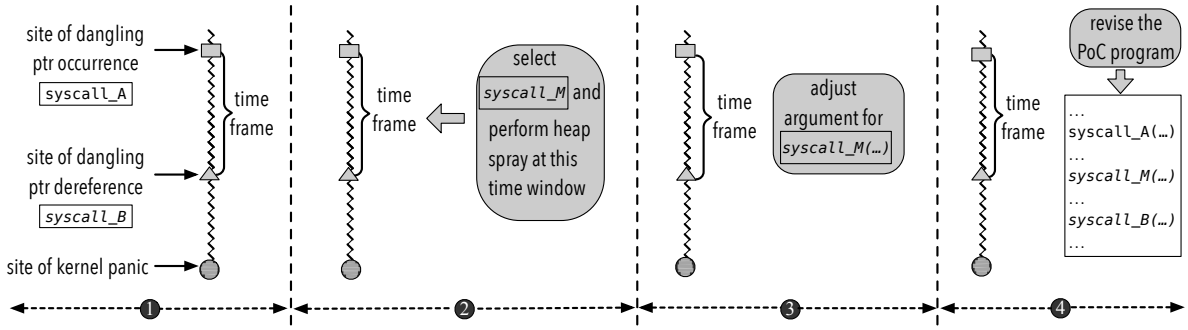


Figure 2.1. The typical workflow of crafting a working exploit. ❶ Identifying the time window between the occurrence of dangling pointer and its dereference; ❷ selecting the proper system call `syscall_M` to perform heap spray; ❷ adjusting the argument of the system call `syscall_M`; ❸ introducing the system call `syscall_M` and revising the original PoC program accordingly. Note that the zigzag line indicates the kernel execution, and `syscall_A` and `syscall_B` denote the system calls that attach to the occurrence of the dangling pointer and its dereference respectively.

2.2 Background and Challenge

To craft an exploit for a UAF vulnerability residing in an OS kernel, a security analyst needs to analyze a PoC program that demonstrates a UAF vulnerability with a kernel panic but not exploits the real target. From that program, he then typically needs to take the following steps in order to perform a successful exploitation.

First, the security analyst needs to pinpoint the system call(s) resulting in the occurrence of a dangling pointer as well as the dereference of that pointer (see ❶ in Figure 2.1). Second, he needs to analyze the freed object that the dangling pointer refers to based on the size of the object as well as the types of heap allocators. Thus, he can identify a system call to perform a heap spray within the time frame tied to the occurrence and dereference of that dangling pointer (see ❷ in Figure 2.1).

Generally speaking, the objective of the heap spray is to take over the freed object and thus leverage the data sprayed to redirect the control flow of the system to unauthorized operations, such as privilege escalation or critical data leakage. As a result, the security analyst also needs to carefully compute the content of the data sprayed based on the semantic of the PoC program, and thus adjust the arguments of the system call selected for performing heap spray, before he finally revises the PoC program for exploitation in a manual fashion. As is specified in ❸ and ❹, we depict the last step in Figure 2.1.

In the past, research (e.g., [17]) has focused on how to augment a security analyst with the ability to select a system call and perform an effective heap spray (i.e., facilitating

type=table

```
1  void *task1(void *unused) {
2      int err = setsockopt(fd, 0x107, 18, ..., ...);
3  }
4
5  void *task2(void *unused) {
6      int err = bind(fd, &addr, ...);
7  }
8
9  void loop_race() {
10     while(1) {
11         fd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
12         // create two racing threads
13         pthread_create (&thread1, NULL, task1, NULL);
14         pthread_create (&thread2, NULL, task2, NULL);
15
16         pthread_join(thread1, NULL);
17         pthread_join(thread2, NULL);
18
19         close(fd);
20     }
21 }
```

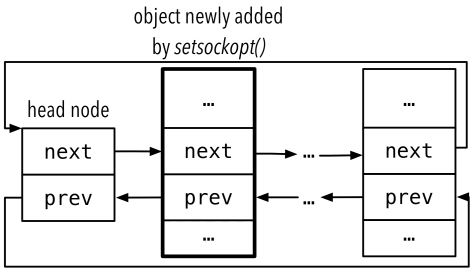
Table 2.1. A PoC code fragment pertaining to the kernel UAF vulnerability (`CVE-2017-15649`).

the step ❷ shown in Figure 2.1). To some extent, this does facilitate the process of crafting exploits. By simply following the typical workflow mentioned above along with the facilitation in the step ❷, however, it is still challenging and oftentimes infeasible for a security analyst to craft a working exploit for a real-world UAF vulnerability. As we will elaborate below through a real-world UAF vulnerability, this is due to the fact that a PoC program barely provides a useful running context, under which a security analyst can perform successful exploitation.

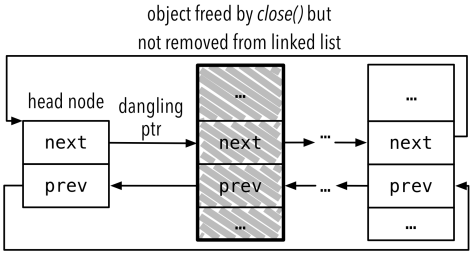
2.2.1 PoC Program for Kernel UAF Vulnerability

Table 2.1 shows a PoC program in C code, capable of triggering the kernel UAF vulnerability indicated by `CVE-2017-15649`. As is shown in line 3, `setsockopt()` is a system call in Linux. Upon its invocation over a certain type of socket (created in line 13), it creates a new object in the Linux kernel, and then prepends it at the beginning of a doubly linked list (see Figure 2.2).

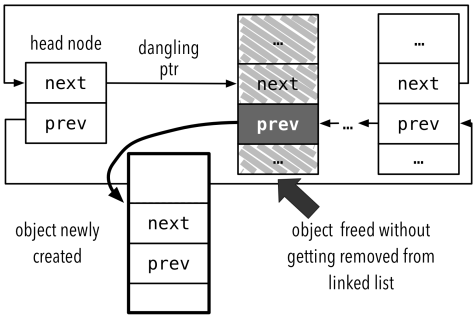
In line 16 and 17, the PoC program creates two threads, which invoke system calls `setsockopt()` and `bind()`, respectively. By repeatedly calling these two lines of code through an infinite loop, the PoC creates a race condition which results in an accidental



Inserting a new object to doubly linked list.



Triggering a free operation with a dangling pointer left behind.



Writing unmanageable data to a memory chunk freed previously.

Figure 2.2. Demonstrating a kernel panic triggered through a real-world kernel Use-After-Free vulnerability indicated by `CVE-2017-15649`.

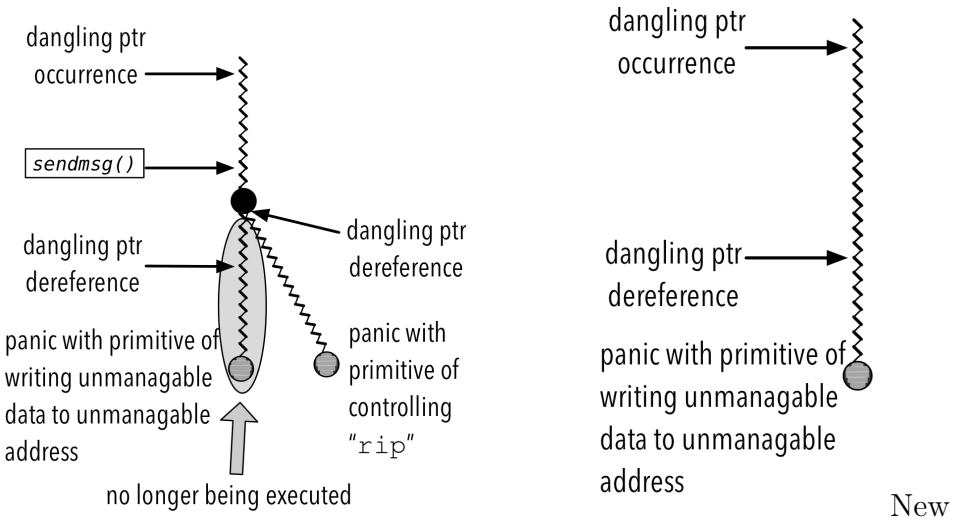
manipulation to the flag residing in the newly added object.

At the end of each iteration, the PoC invokes system call `close()` to free the object newly added. Because of the unexpected manipulation, the Linux kernel fails to overwrite the "next link" in the head node and thus leaves a dangling pointer pointing to a freed object (see Figure 2.2).

In the consecutive iteration of the occurrence of the dangling pointer, the PoC program invokes system calls and creates a new object once again. As is shown in Figure 2.2, at the time of prepending the object to the list, a system call dereferences the dangling pointer and thus modifies data in the "previous link" residing in the freed object, resulting in an unexpected write operation which further triggers a kernel panic in consecutive kernel execution.

2.2.2 Challenge of Crafting Working Exploits

Following the typical workflow specified in Figure 2.1 to craft an exploit for the vulnerability above, in the step ❷, a security analyst needs to identify a proper system call,



Original running context. New running context.

Figure 2.3. Context variation before and after. The original context is indicated by the PoC program in Table 2.1 and the new context is obtained through the insertation of the new system call `sendmsg()`.

use it to perform heap spray and thus turn the PoC into a working exploit. By taking a close look at the unexpected write primitive that the aforementioned PoC left behind, however, we can easily observe that this write operation provide an analyst only with an ability to write the address of a new object to the kernel heap region indicated by the dark-gray box in Figure 2.2.

Given that the allocation of heap objects is under the control of Linux kernel, and an analyst could only have limited influence upon the allocation, we can safely conclude that the unexpected write primitive only gives the analyst the privilege to write an *unmanageable* data (i.e., the address of the new object) to an *unmanageable* heap address in Linux kernel. In other words, this implies that the analyst cannot take advantage of the unexpected write operation to manipulate the instruction pointer `rip` and thus carry out a control flow hijacking, nor leverage it to manipulate critical data in the Linux kernel so that it could fulfill a privilege escalation.

2.3 Overview

While the running example above shows the difficulty of crafting a working exploit, it does not mean the aforementioned vulnerability is unexploitable. In fact, by inserting the system call `sendmsg()` with carefully crafted arguments into the aforementioned PoC program right behind line 22, we can introduce new operations in between the occurrence

of the dangling pointer and its dereference. Since the system call `sendmsg()` has the capability of dereferencing the data in the object newly prepended in the doubly linked list, when an accidental free operation occurs and a dangling pointer appears, it has the ability to dereference the dangling pointer prior to the system call defined in the original PoC and thus changes the way how kernel experiences panic.

As is illustrated in Figure 2.3, the new kernel panic (or in other words the new PoC program) represents a new running context, where the system call `sendmsg()` retrieves the data in the freed object, dereferences it as an invalid function pointer and thus drives the kernel to a new panic state. Different from the original running context indicated by the PoC program in Table 2.1, we can easily observe, this new context provides a security analyst with a new primitive, with which he can spray data carefully crafted, manipulate the instruction pointer `rip` and thus perform a control flow hijack. As we will demonstrate in Section 6, this context even provides a security analyst with the ability to bypass kernel security mitigation such as `SMEP` and `SMAP`.

Motivated by this observation, we propose a technical approach to facilitate the context variation of a PoC program. Along with other techniques that will be introduced in the following sections, we name them `FUZE`, an exploitation framework. The design philosophy behind the framework is that context variation could facilitate the identification of exploitation primitives, with which crafting working exploits can be potentially expedited and the exploitability of kernel UAF vulnerabilities can be significantly escalated. In the following, we discuss the considerations that go into the design of `FUZE` as well as the high level design of this exploitation framework.

2.3.1 Requirement for Design

As is mentioned earlier in Section 2.1, the ultimate goal of `FUZE` is not to yield a working exploit automatically but to facilitate the ability of a security analyst to craft a working exploitation. As a result, we decide to design `FUZE` to facilitate exploit crafting from the following four aspects. First, `FUZE` must provide a security analyst with the ability to track down the vulnerable object, the occurrence of a dangling pointer and its dereference. With this ability, an analyst could rapidly and easily select a proper system call as well as pinpoint the right time window to perform heap spray (i.e., facilitating the steps ❶ and ❷ in Figure 2.1). Second, `FUZE` must augment a security analyst with the ability to synthesize new PoC programs that would drive kernel to panic in different contexts. With this, an analyst could perform context variations in a highly efficient fashion with minimal manual efforts. Third, `FUZE` must be able to extend the ability of an analyst to

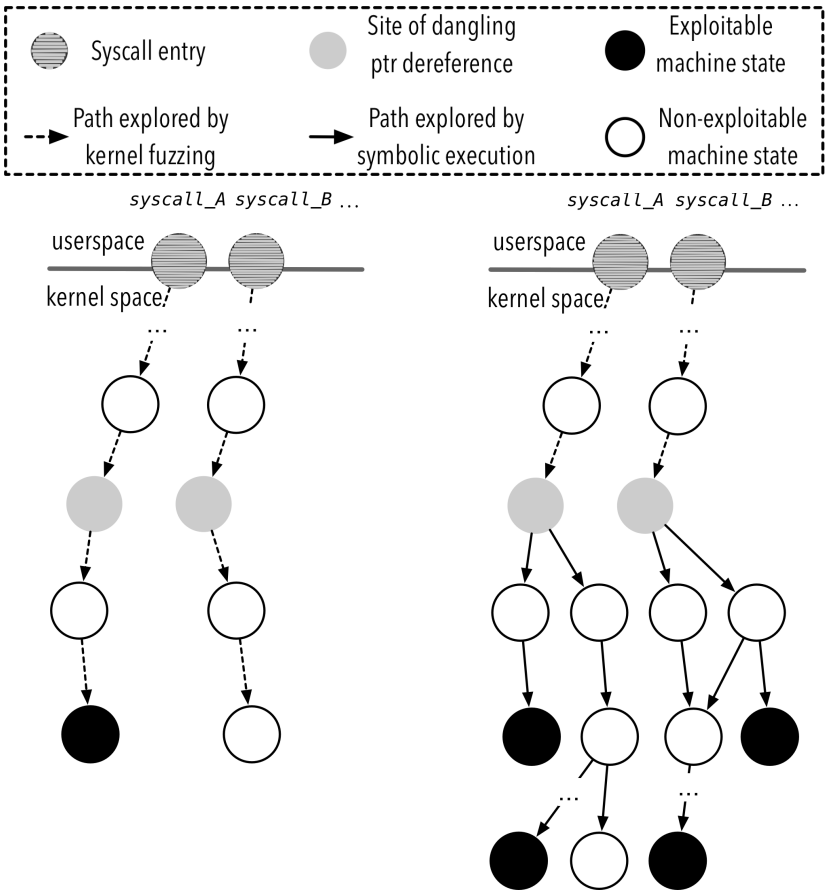


Figure 2.4. An illustration of evaluating contexts and identifying exploitable machine states using kernel fuzzing and symbolic execution. Note that "non-exploitable machine state" denotes the state from which we have not yet had sufficient knowledge to perform an exploitation.

automatically select the useful contexts. This is because newly-generated contexts do not unveil whether they could be used for exploitation, and security analysts typically have difficulty in determining which contexts are useful for successful exploitation. Given the fact that kernel security mitigation widely deployed can easily hinder an exploitation attempt, this determination usually becomes even more difficult and oftentimes involves intensive human efforts. Last but not least, `FUZE` must give a security analyst the capability to automatically derive the data that needs to be sprayed in between the occurrence of a dangling pointer and its dereference. This is because crafting data to take over the freed region and perform exploitation typically needs significant expertise as well as tremendous manpower.

2.3.2 High Level Design

To satisfy the requirements mentioned above, we design FUZE to first run a PoC program and perform analysis using off-the-shelf address sanitizer. Along with the facilitation of a dynamic tracing approach, FUZE could identify the critical information pertaining to the vulnerable objects as well as the time window needed for consecutive exploitation.

Using the information identified, we then design FUZE to automatically vary the contexts of that PoC for the purpose of easing the process of synthesizing new PoC programs. Recall that we alter the context of a PoC program by inserting a new system call that dereferences the vulnerable object in between the occurrence of the dangling pointer and its dereference (see Figure 2.2.2). Technically speaking, we therefore design and develop an under-context fuzzing approach, which automatically explores the kernel code space in the time window identified and thus pinpoints the system calls (and corresponding arguments) that can drive the kernel panic in a new context.

Similar to the context represented by that original PoC, a new context (i.e., new kernel panic) does not necessarily assist an analyst to craft a working exploit. Moreover, as is mentioned above, a security analyst generally has difficulty in determining, following which contexts he could craft a working exploit. Therefore, we further design FUZE to automatically evaluate each of the new contexts. Intuition suggests that we could summarize a set of exploitable machine states based on the exploitation approaches commonly adopted. For each context, we could then examine whether the corresponding terminated kernel state matches one of these exploitable machine states. As is illustrated in Figure 2.3.1, this would allow FUZE to filter out those contexts truly useful for exploitation.

However, this intuitive design is problematic. In addition to the system call selected, the terminated kernel state (i.e., the site where a kernel experiences panic) is dependent upon the remanent content in the freed object. Given that an attacker has the full control over the content in the freed object, using the aforementioned approach that takes only the consideration of system calls, we may inevitably disregard some contexts that allow a security analyst to perform a successful exploitation. Rather than following the intuitive approach above, our design therefore sets each byte of the freed object as a symbolic value and then perform symbolic execution under each context. As is shown in Figure 2.3.1, this allows FUZE to explore the exploitable machine states in a more complete fashion and thus thoroughly pinpoint the set of contexts useful for exploitation.

It should be noted that, as is depicted in Figure 2.3.1, symbolic execution under the context does not mean that symbolically executing kernel code at the site of kernel panic.

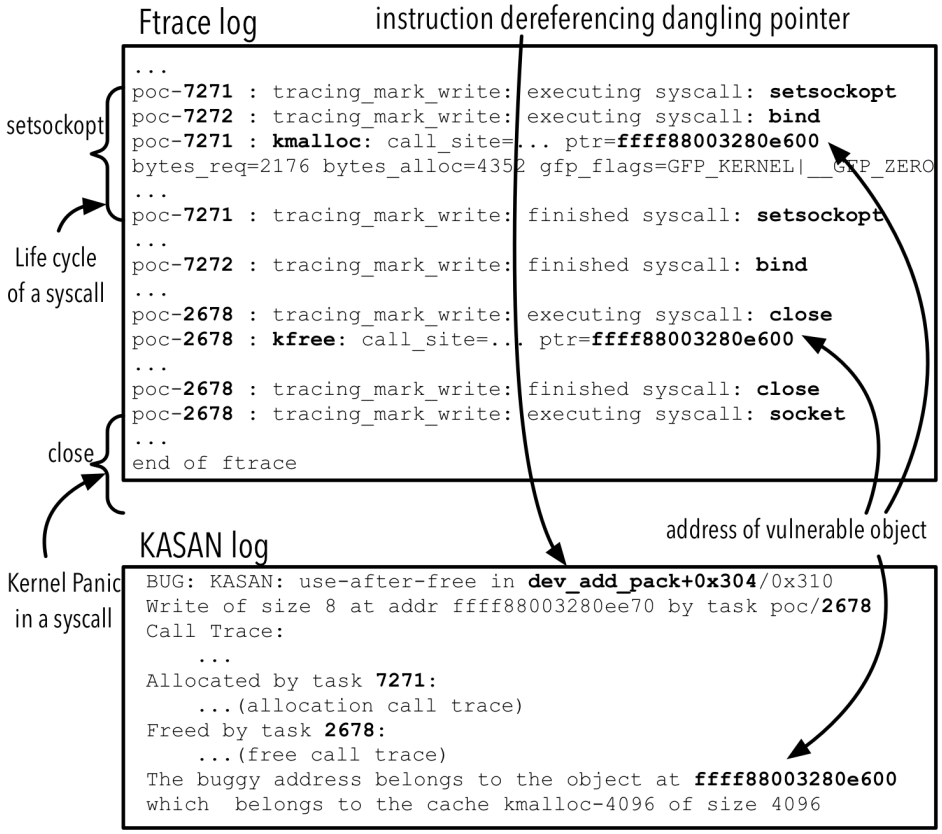


Figure 2.5. A KASAN log obtained from kernel address sanitizer as well as a kernel trace obtained through dynamic tracing.

Rather, it means that we perform symbolic execution right after the site of dangling pointer dereference. As we will demonstrate and discuss in the following section, such a design could prevent incurring path explosion without reaching to any sites useful for exploitation. In addition, it enables FUZE to use off-the-shelf constraint solvers to accurately compute the content that needs to spray in between the occurrence of a dangling pointer and its dereference.

2.4 Design

In this section, we discuss the technical details of FUZE. More specifically, we first describe how FUZE extracts information needed for exploitation facilitation. Second, we describe how FUZE utilizes this information to initialize running contexts, perform kernel fuzzing and thus achieve context variation. Third, we specify how FUZE performs symbolic execution, pinpoints exploitable machine states and thus accomplish context evaluation

as well as the computation for the data sprayed. Finally, we discuss some limitations and other technical details.

2.4.1 Critical Information Extraction

As is mentioned above, FUZE takes as input a PoC program. Then, it extracts information needed for consecutive exploitation by using an off-the-shelf kernel address sanitizer KASAN [18] along with a dynamic tracing mechanism. Here, we describe the information extracted through kernel address sanitizer as well as the design of the dynamic tracing mechanism, followed by how we leverage them both to identify other critical information for exploitation.

Information from Kernel Address Sanitizer. KASAN is a kernel address sanitizer, which provides us with the ability to obtain information pertaining to the vulnerability. To be specific, these include (1) the base address and size of a vulnerable object, (2) the program statement pertaining to the free site left behind a dangling pointer and (3) the program statement corresponding to the site of dangling pointer dereference.

Design of Dynamic Tracing. In addition to the information extracted through KASAN, consecutive exploitation needs information pertaining to the execution of system calls that trigger vulnerabilities. As a result, we design a dynamic tracing mechanism to facilitate the ability of extracting such information. To be specific, we first trace the addresses of the memory allocated and freed in Linux kernel as well as the process identifiers (PID) attached to these memory management operations. In this way, we could enable memory management tracing and associate memory management operations to our target PoC program. Second, we instrument the target PoC program with the Linux kernel internal tracer (`ftrace`). This could allow us to obtain the information pertaining to the system calls invoked by the PoC program.

Other Critical Information Extraction. With the facilitation of dynamic tracing along with KASAN log, we can extract other critical information needed for exploitation. To illustrate the new information obtained through this combination, we take for example the kernel trace and KASAN log shown in Figure 2.5. Using the information obtained through KASAN, we can easily identify the address of the vulnerable object (`0xffff88003280e600`) and tie it to the free operation indicated by `kfree()`. With PID associated with each memory management operation, we can then pinpoint the life cycle of system calls on the trace and thus identify `close()`, the system call tied to the free operation.

Since system call `socket()` manifests as an incomplete trace, we can easily pinpoint that it serves as the system call that dereferences the dangling pointer. From the KASAN

log, we can also identify `dev_add_pack+0x304/0x310`, the instruction that dereferences a dangling pointer. Associating this information with debugging information and source code, we can easily understand how the dangling pointer was dereferenced and further track down which variable this dangling pointer belongs to.

```
PoC_wrapper(){ // PoC wrapping function
    syscallA(...); // free site
    return; // instrumented statement
    syscallB(...); // dangling pointer dereference site
}
```

Wrapped PoC program that encloses free and dangling pointer dereference in two separated system calls without race condition involvement.

```
PoC_wrapper(){ // PoC wrapping function
    while(true){ // Race condition
        threadA(...); // dangling pointer dereference site
        threadB(...); // free site
        // instrumented statements
        if (!ioctl(...)) // interact with a kernel module
            return;
    }
}
```

Wrapped PoC program that encloses free and dangling pointer dereference in two separated system calls with race condition involvement.

Table 2.2. The wrapping functions preventing dangling pointer dereference.

type=table

```
pid = fork();
if (pid == 0)
    PoC_wrapper(); // PoC wrapper function running inside namespaces
else
    fuzz(); // kernel fuzzing
```

Table 2.3. The pseudo-code indicating the way of performing concurrent kernel fuzz testing.

2.4.2 Kernel Fuzzing

Recall that FUZE utilizes kernel fuzzing to explore other system calls and thus diversifies running contexts for exploitation facilitation. In the following, we describe the detail of our kernel fuzzing. To be specific, we first discuss how to initialize a context for fuzz testing. Then, we describe how to set up kernel fuzzing for system call exploration.

2.4.2.1 Fuzzing Context Initialization

As is mentioned in Section 2.3, we utilize kernel fuzzing to identify system calls that also dereference a dangling pointer. To do this, we must start kernel fuzzing after the occurrence of a dangling pointer and, at the same time, ensure the fuzz testing is not intervened by the pointer dereference specified in the original PoC. As a result, we need to first accurately pinpoint the site where a dangling pointer occurs as well as the site where the pointer is dereferenced by the system call defined in the PoC program. As is demonstrated above, this can be easily achieved by using the information extracted through `KASAN` and dynamic tracing.

With the two critical sites identified, our next step is to eliminate the intervention of the system call that is specified in the original PoC and also capable of dereferencing the dangling pointer. To do this, an intuitive approach is to monitor memory management operations and then intercept kernel execution so that it could redirect the execution to the kernel fuzzing right after the occurrence of a dangling pointer. Given the complexity of execution inside kernel, this intrusive approach however cannot guarantee the correctness of kernel execution and even makes the kernel experience an unexpected panic.

To address this technical problem, we design an alternative approach. To be specific, we wrap a PoC program as a standalone function, and then instrument the function so that it could be augmented with the ability to trigger a free operation but refrain reaching to the site of dangling pointer dereference. With this design, we could encapsulate initial context construction for kernel fuzzing without jeopardizing the integrity of kernel execution.

Based on the practices of free operation and dangling pointer dereference defined in a PoC program, we design different strategies to instrument a PoC program (i.e., the wrapping function). As is illustrated in Table 2.4.1, for a single thread PoC program with a free operation and consecutive dereference occurring in two separated system calls, we instrument the PoC program by inserting a return statement in between the system calls because this could prevent the PoC itself entering the dangling pointer dereference site defined in the PoC program. For a multiple-thread PoC program, like the one shown in Table 2.1, the dangling pointer could occur in the kernel at any iteration. Therefore, our instrumentation for such PoC programs inserts system call `ioctl` at the end of the iteration. Along with a customized kernel module, the system call examines the occurrence of the dangling pointer and performs PoC redirection accordingly (see Table 2.4.1).

`KASAN` checks the occurrence of a dangling pointer at the time of its dereference, and

we need to terminate the execution of a PoC before the dereference of a dangling pointer. As a result, we cannot simply use `KASAN` to facilitate the ability of the kernel module to identify dangling pointers.

To address this issue, we follow the procedure below. From the information obtained from `KASAN` log, we first retrieve the code statement pertaining to the dereference of the dangling pointer. Second, we perform a data flow analysis on the kernel source code to track down the variable corresponding to the object freed but leaving behind a dangling pointer. Since such a variable typically presents as a global entity, we can easily obtain its memory address from the binary image of the kernel code. By providing the memory address to our kernel module, which monitors the allocation and free operations in kernel memory, we can augment the kernel module with the ability to pinpoint the occurrence of the target object as well as alert system call `ioctl` to redirect the execution of the wrapping function to the consecutive kernel fuzzing.

2.4.2.2 Under-Context Kernel Fuzzing

To perform kernel fuzzing under the context initialized above, we borrow a state-of-the-art kernel fuzzing frame- work, which performs kernel fuzzing by using sequences of system calls and mutating their arguments based on branch coverage feedbacks. Considering an initial context could represent different environment for triggering an UAF vulnerability, we set up this kernel fuzzing framework in two different approaches.

In our first approach, we start our kernel fuzzing right after the fuzzing context initialization. Since we wrap an instrumented PoC program as a standalone function, this can be easily achieved by simply invoking the wrapping function prior to the kernel fuzzing. In our second approach, we set up the fuzzing framework to perform concurrent fuzz testing. In Linux system, namespaces are a kernel feature that not only isolates system resources of a collection of processes but also restricts the system calls that processes can run. For some kernel UAF vulnerabilities, we observed that the free operation occurs only if we invoke a system call in the Linux namespaces. In practice, this naturally restricts the system call candidates that we can select for kernel fuzzing. To address this issue, we fork the PoC program prior to its execution and perform kernel fuzzing only in the child process. To illustrate this, we show a pseudo code sample in Figure 2.3. As we can observe, the program creates two processes. One is running inside namespaces responsible for triggering a free operation, while the other executes without the restriction of system resources attempting to dereference the data in the freed object.

In addition to setting up kernel fuzzing for different initial contexts, we design two

mechanisms to improve the efficiency of the kernel fuzzing framework. First, we escalate fuzzing efficiency by enabling parameter sharing between the initial context and the fuzzing framework. For kernel UAF vulnerabilities, their vulnerable objects are typically associated with a file descriptor, an abstract indicator used for accessing resources such as files, sockets and devices. To expedite kernel fuzzing for hitting these vulnerable objects, we set up the parameters of system calls by using the file descriptor specified in the initial fuzzing context.

Second, we expedite kernel fuzzing by reducing the amount of system calls that the fuzzing framework has to examine. In Linux system 4.10, for example, there are about 291 system calls. They correspond to different services provided by the kernel of the Linux system. To identify the ones that can dereference a dangling pointer, a straightforward approach is to perform fuzz testing against all the system calls. It is obvious that this would significantly downgrade the efficiency in finding the system calls that are truly useful for exploitation facilitation.

To address this problem, we track down a vulnerable object using the information obtained through the aforementioned vulnerability analysis. Then, we search this object in all the kernel modules. For the modules that contain the usage of the object, we retrieve the system calls involved in the modules by looking up the `SYSCALL_DEFINEx()` macros under the directory pertaining to the modules. In addition, we include the system calls that belong to the subclass same as the ones already retrieved but not present in the modules. It should be noticed that this approach might result in the missing of the system calls capable of dereferencing dangling pointers. As we will show in Section 2.6, this approach however does not jeopardize our capability in finding system calls useful for exploitation.

2.4.3 Symbolic Execution

As is mentioned in Section 2.3.2, we perform symbolic execution under the context with the goal of determining whether a context could direct kernel execution to an exploitable machine state. In the following, we first describe how to set up symbolic execution based on the context obtained through the aforementioned kernel fuzzing. Then, we discuss how to identify the machine states truly useful for exploitation by using symbolic execution.

2.4.3.1 Symbolic Execution Setup

The random input fed into kernel fuzzing could potentially crash kernel execution without providing useful primitives for exploitation (e.g., writing arbitrary data to an arbitrary address). As a result, we start our symbolic execution right before the site where kernel fuzzing dereferences a dangling pointer. To do this, we need to pinpoint the site of dangling pointer dereference, pause kernel execution and pass the running context to symbolic execution.

Different from kernel fuzzing, symbolic execution cannot leverage kernel instrumentation to facilitate this process. This is simply because we use symbolic execution for exploit generation and the exploit derived from instrumented kernel cannot be effective in a plain Linux system.

To address this issue, we utilize the information obtained through `KASAN` and dynamic tracing. As is mentioned in Section 2.4.1, the information obtained carries the code statement pertaining to the dereference of a dangling pointer. Since this information represents in the source code level, we can easily map it to the plain Linux system, and set a breakpoint at that site.

This approach could guarantee to catch the occurrence of a dangling pointer. However, the setup of the breakpoint could intervene kernel execution even at the time when the dangling pointer does not occur. This is because the statement could also involve in regular kernel execution. To reduce unnecessary intervention, we design `FUZE` to automatically retrieve the log obtained from the aforementioned dynamic tracing, and then examine if the pointer pertaining to the statement refers to an object that has already been freed at time the execution reaches to the breakpoint. We force the kernel to continue its execution if the freed object is not observed. Otherwise, we pause kernel execution and use it as the initial setting for consecutive symbolic execution.

2.4.3.2 Exploitable Machine State Identification

Starting from the initial setting, we create symbolic values for each byte of the freed object. Then, we symbolically resume kernel execution and explore machine states potentially useful for vulnerability exploration. To identify machine states exploitable, we define a set of primitives indicating the operations needed for exploitation. Then, we look up these primitives and take them as candidate exploitable states while performing symbolic execution.

Since primitives represent only the operations generally necessary for exploitation, but

not reflect their capability in facilitating exploitation, we further evaluate the primitives guided by exploitation approaches commonly adopted, and deem those passing the evaluation as our exploitable states. In the following, we specify the primitives that `FUZE` looks up and detail the way of performing primitive evaluation.

Primitives Specification. We define two types of primitives – *control flow hijacking* and *invalid write*. They are commonly necessary for performing exploitation under a certain assumption.

A control flow hijacking primitive describes a capability that allows one to gain a control over a target destination. To capture this primitive during symbolic execution, we examine all indirect branching instructions and determine whether a target address carries symbolic bytes (e.g., `call rax` where `rax` carries a symbolic value). This is because the symbolic value indicates the data we could control and its occurrence in an indirect target implies our control over the kernel execution.

An invalid write primitive represents an ability to manipulate a memory region. In practice, there are many exploitation practices dependent upon this ability. To identify this primitive during symbolic execution, we pay attention to all the write instructions and check whether the destination address or the source register or both carry symbolic bytes (e.g., `mov qword ptr [rdi], rsi` where both `rdi` and `rsi` contain symbolic values). The insight of this primitive is that the symbolic value indicates the data we could control and its occurrence in a source register or a destination address or simultaneously both implies a certain level of control over an memory area.

Primitive Evaluation. As is described above, it is still unclear whether one could utilize the aforementioned primitives to facilitate his exploitation. Given a control flow hijacking primitive, for example, it may be still challenging for one to exploit an UAF vulnerability because of the mitigation integrated in modern OSes (e.g., `SMEP` and `SMAP`). To select primitives truly valuable for exploitation (i.e., exploitable machine states), we evaluate primitives as follows.

As is specified in [19], with `SMEP` enabled, an attacker can use the following approach to bypass `SMEP` and thus perform control flow hijacking. First, he needs to redirect control flow to kernel gadget `xchg eax, esp; ret`. Then, he needs to pivot the stack to user space by setting the value of `eax` to an address in user space. Since the attacker has the full control to the pivot stack, he could prepare an ROP chain using the stack along with the instructions in Linux kernel. In this way, the attacker does not execute instructions residing in user space directly. Therefore, he could fulfill a successful control flow hijack attack without triggering `SMEP`.

In this work, we use this approach to guide the evaluation of primitives. At the site of the occurrence of a control flow hijacking primitive, we retrieve the target address pertaining to the primitive as well as the value in register `eax`. Since the target address carries a symbolic value, we check the constraint tied to the symbolic value and examine whether the target could point to the address of the aforementioned gadget. Then, we further examine if the value of `eax` is within range (`0x10000`, τ). Here, (`0x10000`, τ) denotes the valid memory region. $0x10000$ represent the end of an unmapped memory region, and τ indicates the upper bound of the memory region in user space.

Given `SMEP` enabled, another common approach [20] for bypassing `SMEP` and performing control flow hijacking is to leverage an invalid write to manipulate the metadata of the freed object. In this approach, one could leverage this invalid manipulation to mislead memory management to allocate a new object to the user space. Since one could have the full control to the user space, he could modify the data in the new object (e.g., a function pointer) and thus hijack the consecutive execution of Linux kernel.

To leverage this alternative approach to guide our evaluation, we retrieve the source and destination pertaining to each invalid write primitive. Then, we check the value held in the destination. If that points to the metadata of the freed object, we further inspect the constraint tied to the source. We deem a primitive matches this alternative exploitation approach only if the source indicates a valid user space address or provides one with the ability to change the metadata to an address in user space.

In addition to the approaches for bypassing `SMEP`, there is a common approach [21] to bypass `SMAP` and perform control flow hijacking. First, an attacker needs to set register `rdi` to a predefined number (e.g., `0x6f0` in our experiment). Then, he needs to redirect the control flow to function `native_write_cr4()`. Since the function is responsible for setting register `CR4` – the 21st bit of which controls the state of `SMAP` – and `rdi` is the argument of this function specifying the new value of `CR4`, he could disable `SMAP` and thus perform a control flow hijack attack.

To use this approach to guide our primitive evaluation, we examine each control flow hijacking primitive and at the same time check the value in register `rdi`. To be specific, we check the constraints tied to register `rdi` as well as the target of the indirect branching instruction. Then, we use a theorem solver to perform a computation which could determine whether the target could point to the address of `native_write_cr4()` and at the same time `rdi` could equal to the predefined number.

It should be noticed that this work does not involve leveraging information leak for bypassing `KASLR` and acquiring the base address of kernel code segment. This is

because there have been already a rich collection of works that could easily facilitate the acquirement of the base address of kernel code segment (e.g., [22, 23]) and the facilitation of information leak provided by FUZE is neither a necessary nor a sufficient condition for successful exploitation. In addition, it should be noted that the symbolic execution applied above naturally provides FUZE with the ability to compute the data that needs to be sprayed to the freed object. In this work, we therefore utilize off-the-shelf constraint solver (i.e., SMT) to compute values for all the symbolic variables while the symbolic exploration reaches to the machine states exploitable.

2.4.4 Technical Discussion

Here, we discuss some technical limitations and other design details related to kernel fuzzing and symbolic execution.

Symbolic address. When symbolically executing instructions in Linux kernel for exploitable state exploration, the symbolic execution might encounter an uncertainty where an instruction accesses an address indicated by a symbolic value. Without a concretization to the symbolic value, the symbolic address could block the execution without providing us with primitives useful for exploitation. To address this issue, our design concretizes the symbolic value with a valid user-space address carrying the content to which we have the complete control.

With this design, it is not difficult to note that, crafting an exploit with the symbolic address involved, one would have the difficulty in bypassing SMAP because an access to the user space is a clear violation to the protection of user-space read and write. However, as we will demonstrate in Section 2.6, in practice, this does not jeopardize the effectiveness of FUZE in bypassing security mitigation. This is because FUZE has the ability to identify useful primitives through different execution paths which do not involve symbolic addresses.

Entangled Free and dereference. Recall that FUZE performs under-context fuzzing and diversifies contexts based on the practice of how a PoC program performs object free and dereference (see the two different approaches in Table 2.3). In practice, a PoC might utilize a single system call to perform object free and its dereference. For cases following this practice, FUZE uses symbolic execution for exploitable state exploration but not performs kernel fuzzing. This is simply because we cannot eliminate the intervention of the consecutive dereference after a dangling pointer occurs, and the time window left for fuzzing is relatively short. While such a design limits the context that we can explore, it does not significantly influence the utility of FUZE. As we will show in Section 2.6,

FUZE still provides us with the facilitation for UAF exploitation even if there is only one context for exploration.

2.5 Implementation

We have implemented a prototype of FUZE which consists of three major components – ❶ dynamic tracing, ❷ kernel fuzzing and ❸ symbolic execution. To perform exploration for vulnerability exploitability, FUZE takes a 64-bit Linux system vulnerable to UAF exploitation and runs it on QEMU emulator with KVM enabled. In this section, we present some important implementation details.

Dynamic tracing. To track down system calls as well as memory management operations in Linux kernel, we used `ftrace` to record information related to the memory allocation and free such as `kmalloc()`, `kmem_cache_allocate()`, `kfree()` and `kmem_cache_free()` etc.

Since Linux kernel might utilize RCU, a synchronization mechanism, to free an object, which could potentially fail our dynamic tracing to pinpoint a dangling pointer at the right site, we also force our dynamic tracing component to invoke `sleep()`. To be specific, our implementation inserts function `sleep()` right after the system call responsible for free operations, particularly for the PoC programs where free and dereference operations are separated in two different system calls but not introduce a race condition. For the PoC programs which trigger dangling pointers through a race condition (e.g., the PoC program shown in Table 2.1), we insert function `sleep()` at the end of each iteration.

Kernel fuzzing. As is described in Section 2.4.2, we need to identify candidate system calls potentially useful for exploitation using kernel fuzzing. To do this, we can utilize `syzkaller` [24], an unsupervised coverage-guided kernel fuzzer. However, `syzkaller` defines and summarizes only a limited set of system calls specified in `sys/linux/*.txt`. Considering this set may not include the system calls which we have to perform fuzz testing against, our implementation complements declarative description for 16 system calls.

In addition, we augmented `syzkaller` with the ability to distinguish the kernel panics that are truly attributed to the system calls used by `syzkaller`. When per- forming kernel fuzzing, we expect the system calls used by `syzkaller` could dereference a dangling pointer and thus obtain a new running context for consecutive exploitation. However, it is possible that a dangling pointer is dereferenced by other processes and result in kernel panics. To address this, our implementation extends `syzkaller` to check the kernel panic based on the process ID as well as the process name.

Symbolic execution. We developed our symbolic execution component by using `angr` [25], a binary analysis framework. To enable it to symbolically execute Linux kernel, we first take a kernel snapshot right before dangling pointer dereference. Then, we use the `QEMU` console interface to retrieve current register values, kernel code section and the page where the vulnerable object resides. Considering the symbolic execution might request the access to a page not loaded as the input to `angr` in its consecutive execution, we also detect uninitialized memory access by hooking the operations of `angr` (e.g., `mem_read`, `mem_write`) and migrate target pages based on the demand of symbolic execution with a broker agent. Last but not least, we extended `angr` to deal with symbolic address issues by adding concretization strategy classes.

2.6 Case Study

In this section, we demonstrate the utility of `FUZE` using real-world kernel UAF vulnerabilities. More specifically, we present the effectiveness and efficiency of `FUZE` in exploitation facilitation. In addition, we discuss those kernel UAF vulnerabilities, the exploitation of which `FUZE` fails to provide with facilitation.

2.6.1 Setup

To demonstrate the utility of `FUZE`, we exhaustively searched Linux kernel UAF vulnerabilities archived across the past 5 years. We excluded the UAF vulnerabilities that tie to special hardware devices to experiment as well as those that we failed to discover PoC programs corresponding to the `CVEs`. In total, we obtained a dataset with 15 kernel UAF vulnerabilities residing in various versions of Linux kernels. We show these vulnerabilities in Table 2.4.

Recall that `FUZE` needs to perform fuzzing and symbolic execution in two different settings. For each Linux kernel corresponding to the `CVE` selected, we therefore enabled debug information and compiled it in two different manners – with and without `KASAN` and `KCOV` enabled. For some vulnerabilities, we also migrate UAF vulnerabilities from the target version of a Linux kernel to a newer version by reversing the corresponding patch in the newer version of the Linux kernel. This is because some obsolete Linux kernels are not compatible to `KASAN`. As is mentioned in Section 2.4.3, the address space layout randomization is out of the scope of this work. Last but not least, we therefore disabled `CONFIG_RANDOMIZE_BASE` option in all Linux kernels that we experiment.

CVE-ID	# of public exploits		# of generated exploits	
	SMEP	SMAP	SMEP	SMAP
2017-17053	0	0	1	0
2017-15649	0	0	3	2
2017-15265	0	0	0	0
2017-10661	0	0	2	0
2017-8890	1	0	1	0
2017-8824	0	0	2	2
2017-7374	0	0	0	0
2016-10150	0	0	1	0
2016-8655	1	1	1	1
2016-7117	0	0	0	0
2016-4557	1	1	4	0
2016-0728	1	0	3	0
2015-3636	0	0	0	0
2014-2851	1	0	1	0
2013-7446	0	0	0	0
Overall	5	2	19	5

Table 2.4. Exploitability comparison with and without FUZE.

CVE-ID	Fuzzing		Symbolic Execution		
	Time	# of syscalls	Min # of BBL	Max # of BBL	Ave # of BBL
2017-17053	NA	NA	6	18	13
2017-15649	26 m	433	4	39	21
2017-15265	NA	NA	4	5	5
2017-10661	2 m	26	7	14	11
2017-8890	139 m	448	13	86	48
2017-8824	99 m	63	2	33	23
2017-7374	NA	NA	NA	NA	NA
2016-10150	NA	NA	1	1	1
2016-8655	1m	448	4	27	14
2016-7117	NA	NA	1	1	1
2016-4557	1 m	133	3	48	29
2016-0728	1 m	7	21	31	26
2015-3636	NA	NA	NA	NA	NA
2014-2851	146 m	1203	1	5	3
2013-7446	209 m	448	1	2	1

Table 2.5. The Efficiency of fuzzing and symbolic execution.

Regarding the configuration of FUZE, we performed kernel fuzzing and symbolic execution using a machine with Intel(R) Xeon(R) CPU E5-2630 v3 2.40GHz CPU and 256GB of memory. We limited our kernel fuzzing to operate for 12 hours with 4 instances, and fine-tuned our symbolic execution as follows. First, we restricted the maximum number of basic blocks on a single path to be less than 200. Second, we performed symbolic execution only for 5 minutes. Last but not least, for loops, we set symbolic

execution to perform iterations for at most 10 times. With this setup, we could prevent the explosion of our symbolic execution.

To showcase FUZE can truly benefit the exploitation, we performed end-to-end exploitation using the exploitable machine states we identified. To be specific, we computed the data that needs to be sprayed based on the constraints tied to the exploitable states. Then, we performed the heap spray with three different system calls – `add_key()`, `msgsnd ↪ ()`, `sendmsg()` – by following the techniques introduced in [17]. To fulfill exploitation using the exploitable states identified, we eventually redirect the execution to an ROP chain [19] commonly used for exploitation. To illustrate the exploits generated through the facilitation of FUZE, we have released some example exploits along with the virtual machine at [26].

2.6.2 Effectiveness

Table 2.4 specifies the amount of distinct exploits publicly available for each kernel UAF vulnerability as well as their capability of bypassing mitigation mechanisms commonly adopted (i.e., SMEP and SMAP). We use this as our baseline to compare with exploits generated under the facilitation of FUZE. We show this comparison side-by-side in Table 2.4.

With regard to the ability to perform exploitation and bypass SMEP illustrated in Table 2.4, we first observe that there are only 5 publicly available exploits capable of bypassing SMEP whereas FUZE enables exploitation and SMEP-bypassing for 5 additional vulnerabilities. This indicates the facilitation of FUZE could not only significantly improve possibility of generating exploits but, more importantly, escalate the capability of a security analyst (or an attacker) in bypassing security mitigation.

For all the vulnerabilities that an attacker could exploit and bypass SMEP, we also observe a significant increase in the amount of unique exploits capable of bypassing SMEP. This indicates that our kernel fuzzing could diversify the running contexts and thus facilitate our symbolic execution to identify machine states useful for exploitation. It should be noticed that we count the amount of distinct exploits shown in Table 2.4 based on the number of contexts capable of facilitating exploitation but not the exploitable states we pinpointed. This means that, the exploits crafted for the same UAF vulnerability all utilizes different system calls to perform control flow hijacking and mitigation bypassing.

Regarding the capability of disabling SMAP shown in Table 2.4, we discovered only 2 exploits publicly available and capable of bypassing SMAP. They attach to 2 different vulnerabilities – CVE-2016-8655 and CVE-2016-4557. Using FUZE to facilitate exploit generation,

we observe that FUZE could enable and diversify exploitation as well as SMAP-bypassing for 2 additional vulnerabilities (see CVE-2017-8824 and CVE-2017-15649 in Table 2.4). In addition, we notice that FUZE fails to facilitate SMAP-bypassing for CVE-2016-4557 even though a public exploit has already demonstrated its ability to perform exploitation and bypass SMAP. This is for the following reason. As is described in Section 2.4.3, FUZE explores exploitability through control flow hijacking. For some exploitation such as privilege escalation, control flow hijacking is not a necessary condition. In this case, the exploit publicly available performs privilege escalation which bypasses SMAP without leveraging control flow hijacking.

In addition to the ability of bypassing mitigation and diversifying exploits, Table 2.4 reveals the capability of FUZE in facilitating exploitability. As we will discuss in the following session, there are 4 kernel UAF vulnerabilities for which FUZE cannot perform fuzzing because the PoC programs obtained all perform free and dereference operations in the same system call. However, we observe that FUZE can still facilitate exploit generation particularly for the vulnerabilities tied to CVE-2017-17053 and CVE-2016-10150. This is for the following reason. Kernel fuzzing is used for diversifying running contexts. Without its facilitation, FUZE only performs symbolic execution and explores machine states exploitable under the context tied to the PoC program. For the two vulnerabilities above, their running contexts attached to the PoC programs have already carried valuable primitives, which symbolic execution could track down and expose for exploit generation.

Last but not least, Table 2.4 also specifies some cases which FUZE fails to facilitate exploitation. However, this does not imply the ineffectiveness of FUZE. For the case tied to CVE-2015-3636, the vulnerability can be triggered only in the 32-bit Linux system, in which the Linux kernel has to access a fixed address defined by marco LIST_POISON prior to an invalid free. In a 64- bit Linux system on an x86 machine, this address is unmappable and thus this vulnerability cannot be triggered. For the case tied to CVE-2017-7374, the NVD website [27] categorizes it into a kernel UAF vulnerability. After carefully investigating the PoC program and analyzing the root cause of this vulnerability, we discovered that the root cause behind this vulnerability is actually a null pointer dereference. In other words, the vulnerability could make kernel panic only at the time when a system call dereferences a null pointer. Up until the submission of this work, for the cases tied to CVE-2013-7446, CVE-2017-15265 and CVE-2016-7117, both exhaustive search and FUZE have not yet discovered any exploits indicating their ability to perform exploitation. This is presumably because these vulnerabilities could result in only a Denial-of-Service to the target system or they could be exploitable only in support of other vulnerabilities.

2.6.3 Efficiency

Table 2.5 specifies the time spent on identifying the first context capable of facilitating exploitation or, in other words, the context from which the consecutive symbolic execution could successfully track down an exploitable machine state. We observe that `FUZE` could perform fuzz testing against 9 vulnerabilities. For all of them, `FUZE` could pinpoint a valuable context within about 200 minutes, which indicates a relatively high efficiency in supporting exploit generation. For the rest cases, there are mainly two reasons behind the failure of our fuzz testing. First, our kernel fuzzing has to start after the occurrence of a dangling pointer. However, for the case tied to `CVE-2015-3636`, the invalid free operation cannot be triggered in 64-bit Linux kernel. Second, for the other 4 cases, the free and dereference are entangled in the same system call. As is mentioned in Section 2.4.4, this practice leaves a short time frame for kernel fuzzing, and `FUZE` performs only symbolic execution.

To perform kernel fuzzing in a more efficient manner, `syzkaller` customizes these system calls and extends their amount to 1,203. As is mentioned in Section 2.4.2, we trim the set of system calls that `FUZE` has to explore for the purpose of improving the efficiency of `FUZE`. In Table 2.5, we show the amount of system calls that `FUZE` has to explore during 12-hour kernel fuzzing. For all the cases except for that tied to `CVE-2014-2851`, we can easily observe that `FUZE` cut more than 60% of system calls. Among them, there are approximately half of the cases, for which kernel fuzzing needs to explore only about 100 system calls. This implies the contribution to the efficiency in exploitation facilitation.

In addition to the efficiency of kernel fuzzing, Table 2.5 demonstrates the performance of symbolic execution. More specifically, the table shows the minimum, maximum and average length of the path from a dangling pointer dereference site to a control flow hijacking or an invalid write primitive. Across all cases except for `CVE-2015-3636` – which we cannot trigger a UAF vulnerability in a 64-bit Linux system – we observe that the maximum number of basic blocks on a path is 86. This indicates primitives usually occur at the site close to dangling pointer dereference. By setting symbolic execution to explore exploitable machine states within a maximum depth of 200 basic blocks, we could not only ensure the identification of exploitable states but also reduce the risk of experiencing path explosion.

2.7 Related Work

As is described above, our work could expedite the exploit generation for kernel UAF vulnerabilities as well as facilitate the ability of circumventing security mitigation in OS kernel. As a result, the works most relevant to ours include those facilitating the ability of bypassing widely-deployed security mechanisms as well as those automating the generation of exploits for a vulnerability known previously. In the following, we describe the existing works in these two types and discuss their limitations.

Bypassing mitigation. There is a body of work that investigates approaches of bypassing security mitigation in OS kernel with the goal of empowering exploitability of a kernel vulnerability. Typically, these work can be categorized into two major types – circumventing Kernel Address Space Layout Randomization (`KASLR`) and bypassing Supervisor Mode Execution / Access Prevention (`SMEP` / `SMAP`). It should be noticed that we do not discuss techniques for circumventing other kernel security mechanisms (e.g., `PaX` / `Grsecurity` [28]) simply because – for the performance concern – they are typically not widely deployed in modern OSes.

Regarding the approaches of bypassing `KASLR`, a majority of research works focus on leveraging side-channel to infer memory layout in OS kernel. For example, Hund et al. [29] demonstrate a timing side channel attack that infers kernel memory layout by exploiting the memory management system; Evtyushkin et al. [30] propose a side channel attack which identifies the locations of known branch instructions and thus infers kernel memory layout by creating branch target buffer collision; Gruss et al. [22] infer kernel address information by exploiting prefetch instructions; Lipp et al. [31] leak kernel memory layout by exploiting the speculative execution feature introduced by modern CPUs. In this work, we do not focus on expediting exploitation by facilitating bypassing `KASLR`. Rather, we facilitate exploitation from the aspects of crafting exploits and bypassing `SMEP` and `SMAP`.

With regards to circumventing `SMEP` and `SMAP`, there are two lines of approaches commonly used. One is to utilize Return-Oriented Programming (ROP) to disable `SMEP` [19, 32] or `SMAP` [21], while the other is to leverage implicit page frame sharing to project user-space data into kernel address space so that one could run shellcode residing in user memory without being interrupted by `SMEP` or `SMAP` [1]. In this work, we follow the first line of approach to facilitate the ability of bypassing `SMEP` and `SMAP`. Different from the existing approaches in this type, however, we focus on exploring various system calls to facilitate the construction of an ROP chain. This is because chaining disjoint gadgets

in OS kernel for bypassing `SMEP` and `SMAP` needs to explore the abilities of different system calls, which typically requires significant domain expertises and manual efforts.

Generating exploits. There is a rich collection of research works on facilitating exploit generation. To assist with the process of finding the right object to take over the memory region left behind by an invalid free operation, Xu et al. [17] propose two memory collision attacks – one employing the memory recycling mechanism residing in kernel allocator and the other taking advantage of the overlap between the physmap and the SLAB caches. To be able to control the data on a kernel stack and thus facilitate the exploitation of Use-Before- Initialization, Lu et al. [33] propose a targeted spraying mechanism which includes a deterministic stack spraying approach as well as an exhaustive memory spraying technique. To reduce the effort of crafting shellcode for exploitation, Bao et al. [14] develop ShellSwap which utilizes symbolic tracing along with a combination of shellcode layout remediation and path kneading to transplant shellcode from one exploit to another. To expedite the process of crafting an exploit to perform Data Oriented Programming (DOP) attacks, Hu et al. [34] introduce an automated technique to identify data oriented gadgets and chain those disjoint gadgets in an expected order.

In addition to the aforementioned techniques, the past research explores fully automated exploit generation techniques. In [13] and [35], Brumley et al. explore automatic exploit generation for stack overflow and format string vulnerabilities using preconditioned symbolic execution and concolic execution, respectively. In [36], Mothe et al. utilize forward and backward taint analysis to craft working exploits for simple vulnerabilities in user-mode applications. In [16], Repel et al. make use of symbolic execution to generate exploits for heap overflow vulnerabilities residing in user-mode applications. In [25,37,38], Shellphish team introduces two systems (`PovFuzzer` and `Rex`) to turn a crash to a working exploit. For `PovFuzzer`, it repeatedly subtly mutates input to a vulnerable binary and observes relationship between a crash and the input. For `Rex`, it symbolically executes the input with the goal of jumping to shellcode or performing an ROP attack.

In comparison with the exploit generation techniques mentioned above, the uniqueness of our work is mainly manifested in three aspects. First, our technique facilitates exploiting kernel UAF vulnerabilities which have higher complexity than other vulnerabilities. Second, our technique facilitates kernel UAF exploitation at the stage of exploit crafting and mitigation bypassing. Third, as is discussed in earlier sections, our proposed techniques could explore different running contexts, which is essential for the success of kernel UAF exploitation.

2.8 Conclusion

In this paper, we demonstrate that it is generally challenging to craft an exploit for a kernel UAF vulnerability. While there are a rich collection of works exploring automatic exploit generation, they can barely be useful for this task because of the complexity of UAF and scalability of kernel code. We proposed `FUZE`, an effective framework to facilitate exploitation of kernel UAF vulnerabilities. We show that `FUZE` could explore OS kernel and identify various system calls essential for exploiting an UAF vulnerability and bypassing security mitigation.

We demonstrated the utility of `FUZE`, using 15 real-world kernel UAF vulnerabilities. We showed that `FUZE` could provide security analysts with an ability to expedite exploit generation for kernel UAF vulnerabilities, and even facilitate the ability of bypassing widely deployed security mitigation mechanisms built in modern OSes. Following this finding, we safely conclude that, from the perspective of security analysts, `FUZE` can significantly facilitate the exploitability evaluation for kernel UAF vulnerabilities. As future work, we will extend this exploitation framework to perform end-to-end exploitation without the intervention of manual efforts. In addition, we will explore more primitives for exploitation facilitation.

Chapter 3
ELOISE: Identify Elastic Structure as An Exploitable Design Pattern

In the previous chapter, we developed `FUZE` to explore the corruption capability of vulnerabilities which can be further used to perform exploitation and bypass protections. For example, security researchers have demonstrated an exploitation method that utilizes the characteristic of elastic kernel objects to bypass KASLR, disclose stack/heap cookies, and even perform arbitrary read in the kernel. While this exploitation method is considered a commonly adopted approach to disclosing critical kernel information, there is no evidence indicating that elastic kernel object is an exploitable design patterns. It is because the effectiveness of this exploitation method is demonstrated only on anecdotal kernel vulnerabilities. It is unclear whether such a method is useful for a majority of kernel vulnerabilities.

To answer this question, we propose a systematic approach. It utilizes static/dynamic analysis methods to pinpoint elastic kernel objects and then employs constraint solving to pair them to corresponding kernel vulnerabilities. In this work, we implement our proposed method as a tool - `ELOISE`. Using this tool on three popular OSes (`Linux`, `FreeBSD`, and `XNU`), we discover that elastic object is a design pattern as it is pervasive in general caches. Evaluating the effectiveness of these elastic objects on 40 kernel vulnerabilities across three OSes, we observe that they can enable most of the vulnerabilities to bypass `KASLR` and heap cookie protector. Besides, we also observe that these elastic objects can even escalate the exploitability of some vulnerabilities allowing them to perform arbitrary read in the kernel. As such, we conclude that elastic object is an exploitable design pattern. Motivated by this, we introduce a new defense mechanism to mitigate the threat

of elastic kernel objects. We prototype our defense mechanism on Linux, showing this mechanism introduces negligible overhead. In Chapter 6, we will further developed this protection to make it more advanced and universal.

3.1 Introduction

Over the past years, security researchers have introduced many defense mechanisms to harden the kernel, preventing it from being exploited (e.g., [28, 39, 40]). Under the protection of these techniques, common exploitation methods are no longer useful. For example, the design of `KASLR` no longer allows an attacker to hijack the control flow of the kernel and thus reliably jump to a particular exploited function in memory.

Responding to the effort of kernel defense development, security researchers recently devote significant energy to developing methods to circumvent exploitation mitigation and kernel protection commonly adopted by OSes (e.g., [23, 41–53]). Among all these efforts, one commonly adopted approach is to leverage an overwriting primitive to manipulate an elastic kernel object and thus bypass `KASLR`. Technically, this method first leverages an overwriting capability to manipulate a length field in a kernel object. The length field indicates the boundary of an elastic buffer enclosed in the kernel object. By manipulating this field, the attacker can trick the kernel into authorizing him/her to read a memory region that he/she otherwise cannot be entitled to. As we elaborate in Section 3.2, by placing a pointer in the overread region referencing a global variable, the attacker could utilize a disclosure channel to uncover that pointer to the userspace and compute the kernel base address accordingly.

In the past, security researchers have utilized anecdotal kernel vulnerabilities to demonstrate the effectiveness of this exploitation practice in bypassing `KASLR`. They even show that this method can be extended, potentially helping an attacker disclose a stack/heap cookie and even perform arbitrary read. However, by far, it is unclear whether this exploitation approach is useful for a majority of kernel vulnerabilities[1]. As such, we have no clue whether this method should raise our serious concern and motivate us to develop a new kernel defense to mitigate the threat of elastic kernel objects.

To answer this question, one instinctive reaction is to demonstrate exploitability by manually crafting exploits of many kernel vulnerabilities. However, given the sophistication of the kernel code, this approach inevitably introduces a significant amount of

[1]Note that without further clarification, the kernel vulnerabilities we refer to are those that corrupt (or, in other words, manipulate) data on a heap area. The vulnerabilities with only a read capability are excluded.

manual effort, limiting the possibility of scaling this approach to various OSes. Moreover, given the complexity of kernel exploitation, the conclusion drawn through this manual approach might heavily rely upon the expertise of security researchers.

In this work, we design and develop a systematic method to explore the effectiveness of the exploitation method mentioned above. Our basic idea is to utilize static/dynamic analysis to identify elastic kernel objects and then employ constraint solving to pair elastic objects with corresponding kernel vulnerabilities. We implement this approach as a tool and name it after `ELOISE` standing for "**E**xploitab**L**e **O**bject d**IS**cov**E**ry". Using this tool, we show that elastic kernel objects are pervasive in the kernel implementation across three popular OSes (`Linux`, `XNU`, and `FreeBSD`). For many vulnerabilities identified in these OSes, our tool could track down at least one elastic kernel object (and sometimes more), which allows an attacker to disclose heap/stack cookie, bypass `KASLR`, or perform arbitrary read. Motivated by this observation, we further introduce a new defense mechanism to mitigate the threat of elastic kernel objects. Our basic idea is to isolate elastic kernel objects in independent caches. In this way, most of the kernel vulnerabilities are no longer able to manipulate the data in an elastic object and thus trick the kernel into disclosing the critical information to the userspace.

In summary, this paper makes the following contributions.

- We design and develop a systematic method, demonstrating that a commonly adopted exploitation method is a severe threat to existing kernel defenses. It allows a majority of kernel vulnerabilities to disclose heap/stack cookie or bypass `KASLR` (27 out of 40). Besides, it enables some vulnerabilities to perform arbitrary read in the kernel (8 out of 40).
- We implement our systematic method as a tool – `ELOISE` that facilitates the discovery of elastic kernel objects and their pairing with corresponding vulnerabilities. A user study shows that the tool could significantly expedite the development of kernel exploits.
- We design and prototype a new defense mechanism on Linux to mitigate the threat of elastic kernel objects. Experiments show that the new defense significantly mitigates the threat of elastic objects and, more importantly, introduces negligible overhead to an OS (0.19%).

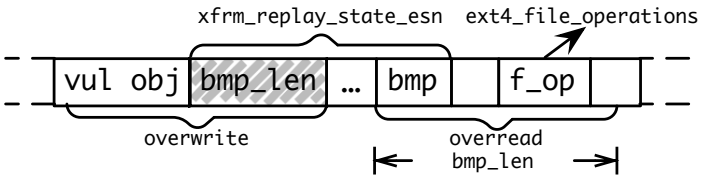


Figure 3.1. The illustration of the anecdotal exploit performing buffer overread and uncovering the function pointer `f_op` referencing the kernel function `ext4_file_operations`.

3.2 Background

In this section, we define elastic kernel objects. Then, we briefly describe how to use these objects to perform exploitation and thus bypass kernel mitigations, followed by the challenges of this exploitation method. Finally, we discuss the threat model.

What is an elastic object? An elastic object always contains a length field that controls the size of an elastic kernel buffer. When the kernel access the data in the buffer, the length field indicates the range of the data that the kernel can read or write. As is summarized in the Appendix 3.9.1, the implementation of an elastic structure/object is very diverse. For example, an elastic object could be a kernel object that encloses the elastic buffer as part of the object (see the 1st in Figure 3.5) or an object that contains a pointer referencing a buffer outside the object (see the 2nd in Figure 3.5). Using elastic objects, the kernel developers could minimize their need for manually managing allocated memory [54] and, more importantly, upgrade the performance of kernel execution by improving the cache hit rate [55].

How to use an elastic object to bypass exploit mitigation? We use a real-world example to illustrate how to leverage an elastic object to perform kernel exploitation and bypass mitigation. As is depicted in Figure 3.1, `xfrm_replay_state_esn` is an elastic kernel object that contains an elastic buffer `bmp` at the end of the kernel object. `bmp_len` is a length field that controls how many bytes the system call `recvmsg` could read data from `bmp` and return to the userland. To perform exploitation, an attacker could utilize the overwriting ability from the vulnerability to enlarge the value of `bmp_len` and thus obtain the ability to disclose the data in `bmp` buffer and the kernel object adjacent to `xfrm_replay_state_esn`. As is illustrated in Figure 3.1, the kernel object next to `xfrm_replay_state_esn` contains a function pointer referencing `ext4_file_operations`. Through the buffer overread, the attacker could disclose the address of the function, calculate the base address of kernel code, and eventually bypass `KASLR`.

In addition to bypassing `KASLR`, an elastic object can also facilitate the disclosure of stack/heap cookies and even enable arbitrary read. For example, if the elastic buffer is

located on the stack, an overread of this buffer can cause access to unauthorized data on the stack (such as stack cookie). If the buffer is located on the heap and its adjacent slot is in free status, the overread could unveil the freed slot's metadata and thus leak the encoded heap cookie accordingly. For some vulnerabilities, an attacker can tamper the value of the pointer arbitrarily. In this case, the attacker can access nearly any memory addresses, and an arbitrary read can be easily granted.

Challenges of using elastic objects for kernel exploitation. To perform the exploitation described above, an adversary first has to ensure a kernel implementation uses the elastic kernel objects. However, given the complexity of kernel code and the diversity of kernel versions, it is extremely labor-intensive to track down such kernel objects by auditing kernel code manually. Second, even if a kernel implementation relies upon elastic kernel objects, the adversary also has to guarantee the existence of leaking channel. Through this channel, he/she could pass the data stored in the elastic buffer (i.e., the buffer the size of which is indicated by the length fields in the kernel object) back to a userland process. However, there has not yet been a systematic approach to pinpointing the link between elastic kernel objects and the userland process. Third, after identifying the elastic kernel objects with the potential to leak data to userland, it does not imply the adversary could utilize that object to leak critical kernel information. Given a vulnerability corrupting data in a particular cache/zone, an attacker cannot guarantee he/she could allocate his desired kernel object to the same cache. Even if both the vulnerable and elastic objects share the same cache/zone, the attacker still needs to ensure the vulnerability gives him a sufficient ability to manipulate the length field tied to the elastic buffer.

Threat model & assumptions. In addition to the defense mechanisms that the exploitation method aims to bypass, this work first assumes that the kernel is armed with other exploitation mitigations and kernel protection mechanisms, such as `SMEP` and `SMAP` protection [56], `KPTI` protection [57], and W⊕R. These protections and mitigations are the kernel defenses most commonly enabled and adopted in `FreeBSD`, `Linux`, and `XNU`. Second, we assume the kernel heap freelist has been randomized on both Linux and `XNU` (`FreeBSD` has no such protection). However, since there have already been exploitation methods [58–60] decisively bypassing kernel freelist randomization, without further clarification, this research does not consider it the obstacle of the general exploitation method. Third, it is very typical that an attacker has only one zero-day vulnerability in hand. Therefore, we do not assume the attacker has additional vulnerabilities to facilitate the exploitation and thus the mitigation circumvention. Finally, the capability

of a vulnerability used in this work indicates at which memory region an adversary could overwrite data freely. Since an attacker can obtain a vulnerability capability from a PoC program which only panics kernel, we conservatively assume an attacker cannot find capabilities other than that manifested through the PoC program. For example, if a PoC program overwrites only four bytes of kernel memory on the heap at the time of kernel panic, we conservatively assume the attacker could obtain only the four-byte overwriting capability.

3.3 Technical Approach

To tackle the challenges mentioned above, we first introduce a method to identify a set of elastic object candidates in the kernel. Then, we specify how to filter out the elastic objects useful for bypassing exploitation mitigation. Finally, we introduce the method for pairing kernel vulnerabilities with corresponding elastic objects.

3.3.1 Identifying Elastic Object Candidates

Tracking down elastic structure candidates. Recall that an elastic object has to contain a length field. As a result, we first examine the existence of an integer variable in kernel structures. Due to the complexity of kernel implementation, the definition of a structure could involve other structural variables. To ease the integer variable identification and reduce possible mistakes, before looking for the integer field in structures, we go through each of the field members in the structure and flatten that structure as follows.

Given a structure S, if its field member f_i is a structural variable, we replace f_i with all its field members $[f_{i1}, f_{i2}, ..., f_{in}]$. If the field member f_i is an array with more than two dimensions, we compute its total size and replace it with a single-dimensional array accordingly. If the field member f_i is a union variable or a nested union, we duplicate the corresponding field member lists by copying the field members in each union and thus obtain a set of new structures $\{S_1, S_2, ..., S_u\}$ where u is the number of different definitions inside the union.

In this work, we repeat the process above recursively until no more operations above can be further applied. Intuition suggests that by following the recursive procedure to pre-process a structure, each of the field members in that structure can be turned into either an ordinary data type (e.g., `char`, `int*`) or a single-dimensional array. With

this, we can easily pinpoint integer variables in structures and deem a structure with an integer field as our candidate.

Pinpointing elastic object candidates at allocation sites. With the candidate structures in hand, our next step is to first track down all the sites of heap memory allocation. Then, we examine whether allocated objects at these sites are in the types of candidate structures. Finally, we examine whether reaching these allocation sites requires any root privilege. In this work, we preserve all the objects that survive the checks above, and treat them as our elastic object candidates.

To pinpoint the sites of heap memory allocation, we search for critical kernel functions in the kernel source code. In Linux, FreeBSD, and XNU, there are two types of kernel functions responsible for allocating memory on the heap. One is `kmalloc`, `kalloc`, and `malloc` series which are used for object allocation on the general cache/zone. The other is `kmem-cache`, `mcache_alloc` and `uma_zalloc` series which are designed for allocation on the special cache/zone. In our design, we deem the invocation of these functions in the kernel code as the sites of memory allocation.

To determine the type of objects allocated at these sites, we analyze the return values of these functions. The reasons are ❶ their return values are always pointers referencing the objects allocated, and ❷ by analyzing the type of the return value, we can easily point out whether the type of an allocated object is within the set of candidate structures. To be more specific, when analyzing the types of return values, we follow a use-def chain and resolve memory alias. As is stated in Section 3.4, we implement our use-def chain analysis by using LLVM. As a result, when performing use-def analysis, we keep track of those instructions relevant to type casting, pointer dereferencing, and argument passing. The operands of these instructions explicitly reveal the object type. By using this information, we can easily infer and conclude the type of each allocated object accordingly.

To determine the privilege required to allocate a kernel object, we check that, in between, whether there is at least one path that does not involve the kernel function call `capable(CAP_SYS_ADMIN)` for Linux, `priv_check`, `priv_check_cred` for FreeBSD, and `priv_check_cred()` for XNU. The reason is that a call to any of these functions on the paths toward an allocation site indicates root permission and the failure of exploitation[2]. Besides, we also check whether any of the callee functions along the path towards an allocation site accesses a device that only privileged users can visit. The reason behind this is that access to privileged devices implies a high privilege for corresponding object

[2]Note that system calls without the root permission requirement can also be in the privilege category. However, they do not tie to the highest permission. In this work, we, therefore, treat them as unprivileged ones.

allocation. In this work, we examine the function's access to privileged devices by using device operator structure (e.g., `struct cdevsw`). Such a structure contains a function pointer through which we can obtain the access control list of the corresponding devices. If the device's permission is root, we will exclude corresponding allocation sites and candidate objects.

Profiling elastic object candidates. Recall that our proposed method includes a component that pairs a vulnerability with corresponding elastic kernel objects for mitigation circumvention. To enable this component, as we will discuss in Section 3.3.3, we need to know the property of elastic objects (e.g., the cache/zone to which an elastic object belongs). As a result, we further perform analysis and profile kernel object as follows.

For allocation functions in `kmem-cache`, `mcache_alloc`, or `uma_zalloc` series, their first argument is always a static or global pointer referencing a special cache/zone. For example, to allocate a kernel object in the type of `struct seq_file`, the allocation function `kmem_cache_zalloc()` puts `seq_file_cache` as its first argument, explicitly specifying the cache/zone the object belongs to. For a kernel object allocated in this manner, we can easily point out the corresponding cache/zone in which the object fits. Different from allocation functions designed for the special cache/zone, allocation functions in `kmalloc`, `kalloc`, and `malloc` series use constant or sometimes constant plus a variable as its first argument, indicating the size of the allocated object. For the functions with a constant as their first argument, we can easily associate the kernel object with the corresponding cache/zone. For example, if the Linux kernel allocates an object with 132 bytes, we can associate the cache `kmalloc-192` with the object because a 132-byte kernel object is too large for `kmalloc-128` and overly small for `kmalloc-256`. For the allocation functions with the first argument in the form of a constant plus a variable, at this particular stage, we temporarily tie the corresponding object to all the general caches/zones with the size greater than the constant.

3.3.2 Filtering out Object Candidates

Recall that the length field in an elastic object indicates the size of a kernel buffer. Also, to use an elastic object to perform exploitation, one must ensure there is a channel to disclose the data in the buffer to the userspace. However, the object candidates identified above do not imply that their enclosed integer variable represents the size of a kernel buffer nor that they support data disclosure. As a result, we further narrow down objects with such properties. To do this, we summarize a set of critical kernel functions

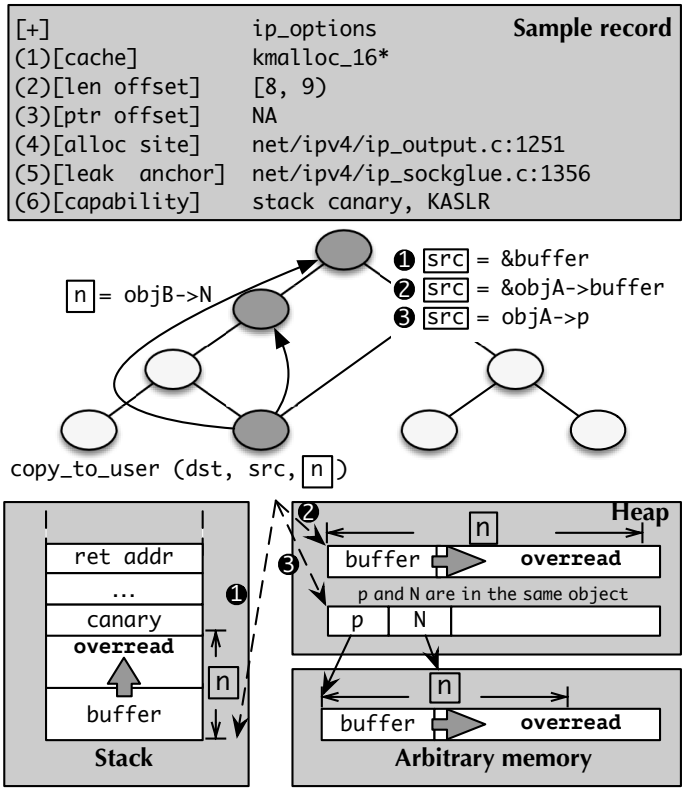


Figure 3.2. The illustration of backward taint analysis starting from `copy_to_user()`. Argument `n` originates from field `N` inside elastic object `objB`. From different paths, taint analysis concludes argument `src` could come from three different variables indicated by ❶∼❸.

(Table 3.3 in the Appendix) and deem the calling sites of these functions as the leaking anchors through which an attacker could potentially uncover data in a kernel buffer to the userspace. With these leaking anchors in hand, we then perform a backward data flow analysis, filtering out the candidate objects that satisfy the properties mentioned above.

As is shown in Table 3.3 (presented in the Appendix), all the critical kernel functions contain two important parameters. One indicates the length of kernel data to be disclosed to the userland (e.g., the second argument `attrlen` in the function `nla_put_nohdr()`). The other specifies the address from which the kernel data would be retrieved (e.g., the last argument `data` in the function `nla_put_nohdr()`). In this work, we take both of these arguments as the taint sources and perform interprocedural backward taint analysis for each of the taint sources individually.

Starting from the taint sources indicating the length of kernel data (e.g., the argument `n` in Figure 3.2), we keep track of the data flow reversely and examine the memory regions from which these arguments originate. If the value of the length argument

originates from a variable allocated on the stack or global memory region, we discard the invocation site of the corresponding kernel function because, as is mentioned earlier, the success of the exploitation relies upon the power of overwriting data on the kernel heap. A length argument originating from a stack, or global variable does not hold the requirement of launching the attack successfully. For the length argument tied to a variable on the kernel heap region (e.g., `n=objB->N` in Figure 3.2), we preserve the calling sites and further perform backward analysis for the taint source corresponding to the data argument mentioned above (e.g., the argument `src` in Figure 3.2). Slightly different from our backward taint analysis applied to the length argument, we first follow the data flow reversely and track down all the sites where the corresponding data argument is initialized (e.g., ❶ `src=&buffer`, ❷ `src=&objA->buffer`, and ❸ `src=objA->p`). Starting from these variable assignment sites, we then analyze the type of the variable accordingly.

For the variable referencing a memory region on the stack (e.g., the dotted line ❶ in Figure 3.2), we conclude that an adversary could potentially obtain an ability to overread data on the stack if an overwriting capability allows the adversary to manipulate the corresponding length argument through the variable identified on the heap (e.g., `objB->N` in Figure 3.2). By using this stack overread capability, we can further conclude the capability of disclosing the stack canary and the return address to bypass `KASLR`. Concerning the variables allocated on the non-stack region (i.e., the heap area[3]), they could be categorized into the following two types, providing an adversary with different exploitability.

The first type indicates the variable referencing the address of one field in a kernel object (e.g., the dotted line ❷ in Figure 3.2). For this type of variable, we conclude that an adversary could potentially obtain an ability to bypass `KASLR` or forge a legitimate heap cookie. The reason is that an adversary could utilize an overwriting capability to vary the corresponding length argument (e.g., `objB->N` in Figure 3.2), follow the corresponding path to trigger the critical function (e.g., `copy_to_user()`), and eventually obtain an overread primitive on the heap. Using a state-of-the-art technique [6, 61] to manipulate heap layout, the adversary could easily turn this overread ability on the heap into the ability to bypass `KASLR` and heap cookie protector.

The second type of variables are pointers enclosed in the kernel objects (e.g., the dotted line ❸ in Figure 3.2). For this type of variable, before drawing any conclusion, we take one additional step, which continues backward taint analysis, and examines whether

[3]Note that the kernel needs to determine the size of the buffer on the global area at the compilation time. Therefore, a variable tied to a data argument cannot be present on the global region if the data argument references an elastic buffer.

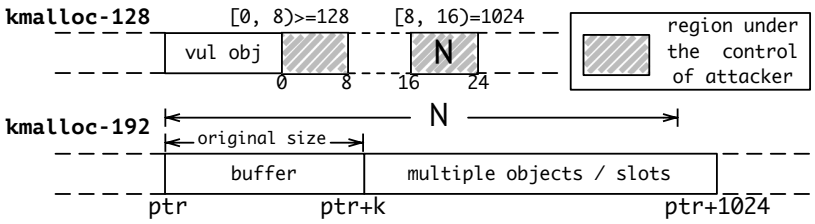


Figure 3.3. The summarization of vulnerability capability & demonstration of buffer overread through the capability.

the variable, as well as that tied to the length argument, are both in the same object. In this work, we conclude that the kernel objects with such a property can potentially provide an adversary with the ability to not only bypass `KASLR` but, more importantly, perform arbitrary kernel read. The reason behind this conclusion is as follows. For variables (associated with the length and data arguments) enclosed in the same kernel object, an adversary could potentially allocate the object in the cache/zone same as that of the vulnerable object and thus utilize the overwriting ability to manipulate both variables accordingly (e.g., manipulating `p` and `N` in Figure 3.2). Since the variable tied to the length argument indicates how many bytes of data one could read, and the other variable specifies from which memory region one could read the data, the manipulation capability turns the overwriting vulnerability into an arbitrary read primitive. With this primitive, the adversary could read the content in the interrupt descriptor table (IDT), compute the base address, and thus circumvent `KASLR`. Besides, as the previous research [41] has already demonstrated, the adversary could also use this arbitrary read primitive to dump memory and search for the string "root:!:" in the memory. Since this string is part of the file "/etc/shadow", the adversary could disclose a user's hashed password and potentially recover the password by using password cracker (e.g., John the Ripper password cracker [62]).

In this work, for each of leaking anchors surviving from the analysis above, we store the corresponding conclusive capabilities in a database for the consecutive analysis (e.g., "stack canary" and "KASLR" depicted in Figure 3.2). For the elastic kernel objects and corresponding structures included in the candidate set but not associated with any critical functions, they do not satisfy the definition of elastic kernel objects (i.e., having a channel to disclose data in the elastic kernel buffer to userspace). Therefore, we discard such objects and structures from the candidate set. By following the analysis above as well as the way to knock off unqualified elastic objects, we can eventually filter out all the elastic kernel objects useful for exploitation and, thus, mitigation circumvention.

3.3.3 Pairing Vulnerabilities with Objects

Through the analysis above, we could obtain the information regarding each of the elastic objects. As is shown in Figure 3.2, the information includes (1) the caches or zones tied to each object, (2) the offset of the length field corresponding to the head of the object, (3) the offset of the elastic buffer corresponding to the object head if the pointer referring the elastic buffer and the length field share the same object, (4) the sites where the kernel allocates the object, (5) the leaking anchor(s), and (6) conclusive capability inferred through the paths toward the corresponding leaking anchor(s).

With the information in hand, given a vulnerability, we could use the following approach to pair that vulnerability with elastic objects accordingly. First, we utilize a debugging tool (GDB [63]) to track the execution of a PoC program triggering the vulnerability but not necessarily performing actual exploitation. Based on our observation from the debugging tool, we then identify the cache where the vulnerability corrupts data. Besides, we manually summarize, to which specific memory locations in that cache, the vulnerability gives an attacker the ability to overwrite data. At each location, what value range could be under an attacker's control. For example, as is shown in Figure 3.3, through our manual analysis against an out-of-bound vulnerability, we can discover that the vulnerability overflows a vulnerable object in the cache `kmalloc-128` and corrupts data in it adjacent spot. Recall that we can allocate an elastic object at that adjacent spot by using the technique proposed in [6, 61]. Therefore, we can manually summarize the region under corruption is the first and the third 8 bytes of that elastic object, and the values put into these two regions have to be greater than 128 and equal to 1024, respectively.

In this work, we deem these summarized results as the capability of a vulnerability and utilize a list of 2-tuples `[(`$VCache_1$`, `Cap_1`), ..., (`$VCache_n$`, `Cap_m`)]` to model such a capability. Here, $VCache_i$ indicates the cache the vulnerability could corrupt, and Cap_j represents the range of unauthorized memory region at which vulnerability could overwrite data. Considering that, in one particular cache, a vulnerability might have the ability to modify data at multiple memory regions, we represent the Cap_j as a list of assertions `[(`$R_{j1}|Op_{j1}|V_{j1}$`), ..., (`$R_{jx}|Op_{jx}|V_{jx}$`)]`. In this assertion list, the notation R_{jk} indicates the unauthorized memory area where, through the corresponding vulnerability, an attacker could manipulate. The notations Op_{jk} and V_{jk} altogether specify the attacker's control over that region. To illustrate this, we again take, for example, the case shown in Figure 3.3. Using the representation above, we could write the capability of that vulnerability as `(kmalloc-128, [([0, 8)⩾128), ([8, 16)=1024)])`. Here,

`kmalloc-128` denotes the cache the vulnerability could corrupt. `[0, 8)`$\geqslant$`128` and `[8, 16)=1024` indicate the ranges of values that an adversary could put into the corresponding memory regions.

By using the modeling approach above to describe the capability of a vulnerability, we can automatically pair a vulnerability with those elastic objects useful for exploitation. To be specific, given a vulnerability, we first filter out all the elastic objects, if the caches or zones they tie to (indicated by the notation $OCache_1 \cdots OCache_h$) happen to have an overlap with the caches associated with the vulnerability (i.e., $\exists i \in [1, n],\ \exists j \in [1, h] \mid (VCache_i = OCache_j)$). For each elastic objects filtered out, we then examine whether the memory regions under manipulation cover its length field[4]. With this examination, we can preserve the elastic objects with their length field covered and thus narrow down the elastic objects useful for exploitation further. For the elastic kernel objects preserved, last but not least, we check if an adversary could use his overwriting ability to manipulate the length field and thus go over the boundary of the elastic buffer. Take the vulnerability capability mentioned above, for example. Assume the length field of an elastic object is at the third 8 bytes, and the object contains a pointer referencing a buffer in an object located at the cache `kmalloc-192`. Given part of the vulnerability capability `([16, 24)=1024)`, we can conclude the attacker could change the size of the elastic buffer to 1024 and thus overread the buffer and access the data in its entire adjacent slot (see Figure 3.3). With this ability, we can further conclude the attacker could potentially forge heap cookies and bypass `KASLR` because he/she could keep that slot adjacent to the buffer either unoccupied or occupied with an object enclosing a function pointer.

In this work, to determine the overread ability and thus conclude corresponding exploitability, we utilize the following strategies. For the elastic buffer on the heap, we check whether the newly manipulated buffer size is at least twice as large as the cache (at which the buffer is located). With this, regardless of the position of the buffer in an object, we can always guarantee to overread the entire adjacent slot or kernel object and thus potentially give an adversary the ability to bypass `KASLR` or forge a legitimate heap cookie. For the elastic buffer on the stack, we compute the stack frame where the elastic buffer is located and then examine whether the newly manipulated buffer size is larger than the frame size. With this, we can ensure an adversary could always have

[4]Note that we also examine the manipulable memory region covers the elastic buffer if both the pointer referring the elastic buffer and the length field share the same object. In this work, we record the elastic object with this property because this indicates the object could potentially offer the capability of arbitrary read.

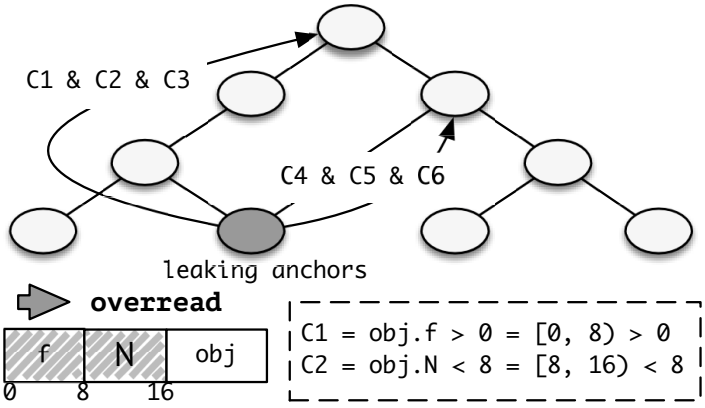


Figure 3.4. The illustration of constraint extraction from two paths. `ELOISE` preserves only the constraints pertaining to the manipulated fields in elastic object `obj` (i.e., `C1 & C2`).

access to the stack canary and the return address. It should be noted that the restriction we impose upon the process of determining possible exploitability is very tight. Even if some of them do not hold, it is still possible to bypass corresponding mitigation. In this work, we impose universal, tight restrictions. First, it is because the design eases our process in finding elastic kernel objects and concluding exploitability. Second, it is because the tight restriction represents the lower bound of a concluded exploitability.

Through the series of examinations above, for any individual vulnerability, we can easily track down the corresponding elastic objects friendly for exploitation. However, this does not imply that the vulnerability could automatically inherit the security implication tied to the elastic objects. On the paths toward the leakage sites after the manipulation of an elastic object, the kernel might use the manipulated fields as branch predictors. With an improper modification on some of the fields, the kernel execution might be detoured, and the kernel would no longer invoke the critical functions like `copy_to_user()`. In some situations, the kernel may even accidentally touch invalid or non-permitted memory regions and thus trigger general page fault (GPF) and even kernel panic. As a result, before concluding the process of pairing vulnerabilities with elastic objects, we perform further analysis as follows.

Given a vulnerability and one of its corresponding elastic objects identified through the method mentioned above, we first retrieve all its paths towards leaking anchors (see Figure 3.4). Along each path, we first extract all the pointer dereferences. Then, we refine the set of branching constraints that must hold in order to reach out to the leakage site. For each of the dereferenced pointers, we ensure the pointer references a legit memory area if memory manipulation inevitably touches its original value. For the constraint set, we pay particular attention to the constraints in which the manipulated

fields in the elastic object are enclosed or has data dependency with the variable involved. For example, as is illustrated in Figure 3.4, after the manipulation of an elastic kernel object, manipulated values fill the length field `N` and one of its adjacent fields `f`. By performing the analysis above, we can first identify two distinct paths through which an attacker could potentially disclose data through an elastic buffer. Then, we can filter out all the constraints pertaining to the length field and its adjacent field (e.g., `C1=obj.f>0`; `C2=obj.N<8` shown in Figure 3.4). In this case, the constraints filtered out indicate the conditions that an attacker has to satisfy when crafting manipulated values for corresponding fields. In this work, we, therefore, introduce these constraints as the additional restrictions to the process of pairing vulnerabilities with corresponding elastic objects (e.g., `C1&C2` in Figure 3.4).

As is depicted in Figure 3.4, similar to the way to model a vulnerability capability, we represent each manipulated field by using the offset to the head of the elastic object, and summarize the corresponding constraints in the format of `(range|op|value)`. Here, the notation `range` denotes the memory area tied to the manipulated field in the elastic object, and the notation `op|value` specifies the condition the manipulated field has to satisfy. For example, `"[8, 16)<8"` depicted in Figure 3.4 indicate that the length field locates at the second 8 bytes of the elastic kernel object. In order for being able to follow the path on the left, the value put into the length field has to be less than 8.

3.4 Implementation

In this research, we implemented our idea as a tool and named it after `ELOISE`. In the following, we present some important implementation details pertaining to the design mentioned above. We release our code and exploits at [5] to foster future work.

Bitcode generation. As is discussed in Section 3.3, our proposed method is based on static analysis. In our implementation, `ELOISE` utilizes default settings to generate `LLVM` bitcode files, compiling `Linux`, `FreeBSD`, and `XNU` kernels by using `defconfig`, `GENERIC` and `xnudeps` configuration, respectively. Then, it takes as input a list of unlinked `LLVM` bitcode files. During static analysis, both compilation optimization and the heavy usage of load/store instructions could increase the burden of alias analysis as well as control-flow graph construction. As a result, in order to minimize their indirect impact upon our elastic kernel object identification, improve static analysis efficiency, and reduce false negatives for elastic object identification, `ELOISE` dumps bitcode files by using `WriteBitcodeToFile()` provided by `LLVM`. To be specific, `ELOISE` invokes this method in between `mem2reg` pass

and optimization passes, which reduces redundant load/store instructions and constructs SSA form for variables.

Control-flow graph construction and alias analysis. In a recent work [64, 65], researchers extend `LLVM` and implement a new tool for building a call graph for the Linux kernel. Technically, the tool leverages a multi-layer type analysis method[5] to construct a field-sensitive call graph. In the implementation of `ELOISE`, we borrow the call graph construction pass from this tool and use it as a building block for our kernel control-flow graph construction. To better serve our purpose, we customize the call graph by cutting off nodes and associated edges in the call graph that represent functions in `.init.text` section. This is because these functions cannot be invoked after kernel booting and are not useful for exploitation. Based on our customized call graph, we further construct our context-sensitive control-flow graph. To be specific, we add intra edges between basic blocks based on the successors specified in the instruction `BranchInst`. Besides, we introduce inter edges by connecting `Callsite` to the corresponding function entry and linking `ReturnInst` back to the same `Callsite`. As is mentioned in Section 3.3.1, our proposed technique also relies upon alias analysis results to determine the type of allocated kernel objects and perform backward taint analysis. Therefore, `ELOISE` extracts alias analysis results by reusing the `AliasAnalysis` pass provided by `LLVM` project. In comparison with one-level context-sensitive Andersen's algorithm, the precision of this alias analysis is roughly the same. It should be noted that, in the process of elastic object identification, we propagate a tainted variable to its aliasing variable only if we confirm the two variables have a must-alias relationship. With this strategy, while we might introduce an under-tainting issue, this approach could significantly minimize false positives (i.e., avoiding from pinpointing the kernel objects that are not actually elastic).

Elastic object identification. To identify elastic object candidates, we track down memory allocation functions in kernel code. Then, we analyze the return values of these functions, determining their types. In our implementation, `ELOISE` utilizes the following three instructions to infer type information. For the instruction `Getelementpr`, its arguments contain a pointer referencing an object as well as the type information of that object. Through this instruction, `ELOISE` can easily obtain the type information pertaining to the object. For the instruction `BitCast`, one of its arguments is a pointer referencing an object. Through the other two arguments, which specify the types being cast before and after, `ELOISE` can also interpret the type information easily. With respect to the instruction `CallInst`, a pointer referencing an object could be enclosed as part

[5]The LLVM pass released on GitHub [66] has implemented only a 2-layer type analysis.

of its arguments. Similar to the two instructions above, ELOISE can easily infer the type information for that object based on the type information specified along with the corresponding argument.

Elastic object filtering. In Section 3.3.2, we filter out the elastic object candidates by performing backward taint analysis from the length argument in the critical kernel function (e.g., the variable `n` in the function `copy_to_user(void __user* to, const void* from, unsigned long n)`). During our analysis, the length argument can be obtained through multiple dereference. For example, in the multi-layer dereference `A->B->len`, `B` is the actual elastic object that encloses the length field. As a result, to avoid mistakenly taking `A` as the elastic object, ELOISE taints only one `LoadInst` backward if a multi-layer dereference involves at the leaking anchor. As is mentioned in Section 3.3.2, we also need to determine which memory region the corresponding function arguments refer to (e.g., stack, heap, or global). In our implementation, ELOISE distinguishes memory regions by following the rules below. If the memory region is from `AllocaInst` instructions which allocate memory on stack, we deem the region as part of the stack. If the memory region is represented by a global/static variable, we assign it to a global area. Otherwise, we conservatively treat it as a memory region on the heap.

Critical constraint set extraction. As is described in Section 3.3.3, before pairing a vulnerability with elastic objects, we need to collect the constraint sets from the paths that lead kernel execution to the leaking anchor but not detour it to other sites or trigger accidental execution termination. To do this, ELOISE analyzes LLVM IR to determine the usage of object fields in each path towards leaking anchors based on the semantics of instructions. For example, if a field in an object is used in the instruction `CmpInst`, ELOISE first interprets the comparison by its `Predicate` (e.g., `ICMP_ULT` means the greater relationship "$>$"). Then, it checks which branch can reach the corresponding anchor. In this example, we keep the "$>$" operator if only the `TRUE` branch can reach the anchor. We flip the the operator into "$\leq$" if only the `FALSE` branch reaches the anchor or the length argument can be updated in `TRUE` branch. Otherwise, we discard this comparison because the corresponding constraint is not relevant to the practice of restricting kernel execution from flowing towards the leaking anchor.

Object and vulnerability pairing. In our implementation, ELOISE utilizes the `Z3` solver [67] to pair a vulnerability with elastic objects. More specifically, we use `BitVec` to represent the layout of an object and machine arithmetic to describe the corresponding memory range. For example, given a constraint `"[8, 16)<8"` in an object with the size of `128` bytes, we create `x = BitVec('x', 128*8)` to represent the object layout, use

`x << (8*8)>> (112*8)` to depict its range `[8, 16)`, and finally employ the representation `(x << (8*8)>> (112*8)< 8)` to indicate the constraint. Given an elastic object, for each path towards a leaking anchor, `ELOISE` first conjuncts the corresponding constraints on that path with a set of constraints indicating the capability of the corresponding vulnerability. Then, `ELOISE` feeds those combined constraints to the `Z3` solver. If a solution is successfully identified, it means the elastic object could be paired with the capability of the vulnerability and can be used for exploitation. Otherwise, we conclude that the elastic object under our examination cannot facilitate the exploitation of that vulnerability. It should be noted that, in the process of pairing, if one of the constraints tied to a path involves a variable (e.g., [0, 8)<`cache_detail.20`), `ELOISE` conservatively skips that path simply because the lack of information about that variable introduces uncertainty for using that object for exploitation. Admittedly, this implementation inevitably influences the number of elastic objects available for exploitation. However, as we will show and discuss in Section 3.5, even with such a conservative implementation, for nearly all kernel vulnerabilities, `ELOISE` can point out at least one elastic kernel object for bypassing corresponding exploit mitigation.

3.5 Evaluation

We design three experiments to evaluate the effect of elastic objects on kernel protection circumvention across three open-sourced kernels (`Linux`, `FreeBSD`, and `XNU`).

3.5.1 Experiment Design

Experiment I: evaluating our tool and characterizing elastic objects. As is described and discussed above, `ELOISE` relies upon static analysis. However, the accuracy of static analysis heavily relies upon the correctness of the kernel's control flow graph and the level of optimization introduced at the compilation stage. As a result, `ELOISE` inevitably introduces false positives (i.e., mistakenly identifying an object as an elastic one) and false negatives (i.e., failing to pinpoint an elastic object or successfully identifying an elastic object but failing to associate it with a disclosure channel). In this work, we design an experiment to evaluate the false positives (FP) and false negatives (FN).

To evaluate false negatives, ideally, we should follow a two-step procedure. First, we go through all the object allocation sites in the kernel implementation and determine whether the type of the allocated objects is one of the elastic structures defined in

Section 3.3.1. Second, for every elastic object confirmed in the first step, we extract its corresponding elastic buffer and then examine whether the kernel can pass the data in the elastic buffer to the userland through the disclosure functions defined in Table 3.3 (e.g., `copy_to_user()` or `copyout()`). After each step, we compare the results obtained through the manual efforts with those reported by `ELOISE` and thus pinpoint the false negatives accordingly.

Given the complexity and huge codebase of kernel implementation, this manual approach, however, is infeasible. Take Linux kernel as an example. Its implementation (v5.5.3) contains about 14 million lines of C code, 53,775 allocation sites, and 75,100 sites that invokes one of the kernel functions listed in Table 3.3. Even after successfully confirming an allocated object is in the type of an elastic structure, a kernel expert still needs to manually walk through tens of thousands of C code lines to determine whether the allocated object has a connection with the corresponding disclosure kernel functions. As such, we study false negatives through a random sampling approach.

First, we randomly sampled 800 out of 6,383 elastic structures (Note that these structures have no FN for sure). Then, we manually identify all of their allocation sites in Linux, followed by checking their connection with the disclosure functions. In this work, we identify false negatives by comparing our manually analyzed results with those of `ELOISE`. We argue this false-negative measure is representative even if we do not perform this analysis on all the allocation sites nor apply this manual effort on other kernels (i.e., `FreeBSD` and `XNU`). It is because ❶ our allocation site sampling is random, covering 12.5% of the kernel allocation sites; ❷ the design of three kernels shares many common properties, and the false negatives observed on one kernel could be potentially generalized to other kernels; ❸ our manual analysis relies upon the workforce of three Linux kernel researchers who have extensive experience in reporting Linux kernel bugs and publishing kernel research at top tier system and security conferences.

To evaluate false positives, we leverage automated tools along with our manual effort. For Linux and FreeBSD kernels, we first instrument panic functions at the sites where `ELOISE` identifies the allocation of elastic objects, and the sites where `ELOISE` discovers the disclosure of the data from an elastic buffer. Then, for each instrumented site, we use `Syzkaller` [24] to assist us in checking whether there is indeed a concrete input that could reach these sites to allocate elastic objects and thus disclose data in the elastic buffers. Following the kernel fuzzing, we manually examine the sites which the fuzzing fails to reach because these unreachable sites could result from either our inaccurate static analysis or the limitation of our fuzzing technique. In this work, we take the

unreachable cases as the false positives of ELOISE only if our manual review still cannot find a concrete input to reach out to our interest sites. It should be noted that there are no fuzzing tools publicly available for the XNU kernel. As a result, we solely rely upon manual effort for the false-positive study on XNU. For all the true positives, we then conduct further experiment to understand the pervasiveness of elastic objects and study their characteristics.

Experiment II: evaluating the ability to bypass mitigation. Recall that one of our objectives is to study whether elastic kernel objects could enable many kernel vulnerabilities to bypass widely deployed exploit mitigation methods, such as KASLR, heap protector, and stack canary. To answer this question, we select 40 kernel vulnerabilities and summarize the capability of each kernel vulnerability. Then, we use ELOISE to examine whether one of the identified elastic objects could be paired with these vulnerabilities and hence enable kernel protection bypassing.

Due to the space limit, we list these 40 vulnerabilities in Table 3.7 and describe how we manually summarize vulnerability capability in the Appendix. We argue that the selected vulnerabilities are representative. First, this list includes all types of vulnerabilities that corrupt data on the kernel heap (i.e., out-of-bound write, use-after-free, and double-free). Second, it covers all the heap corruption vulnerabilities used in various Linux kernel exploitation research (e.g., [4, 6, 7, 68]). Third, it includes 10 Linux kernel vulnerabilities randomly sampled from syzbot [69] (a Linux kernel vulnerability dashboard). These randomly sampled vulnerabilities are identified by Syzkaller recently but have not yet assigned with an CVE ID. Finally, it encloses all XNU and FreeBSD vulnerabilities publicly disclosed in the past three years.

It should be noted that, although the National Vulnerability Database lists many CVEs relevant to FreeBSD and XNU, a majority of them cannot be used for our evaluation because the detail of the vulnerabilities is incomplete or completely missing or the trigger of the vulnerabilities requires a particular hardware. As such, we chose our FreeBSD and XNU test cases by following three criteria. ❶ A vulnerability has to demonstrate its capability through a PoC program that triggers the vulnerability but, not necessarily, has to perform actual exploitation. ❷ The vulnerability has to allow us to trigger it without requiring a particular hardware device nor root privilege. ❸ By running the publicly released PoC program to trigger the vulnerability, the kernel panic or failure (typically declared along with a released writeup) has to be reproducible. To ease our experiment, we migrate the Linux, FreeBSD, and XNU vulnerabilities above to one particular version of Linux kernel (v5.5.3), one specific FreeBSD (v12.1), and one specific XNU kernel

(XNU-4903.221.2), respectively. They are all the latest versions of the kernels at the time we conduct our experiment.

Experiment III: evaluating the utility of ELOISE in exploit development assistance. To evaluate the effectiveness of `ELOISE` in expediting exploit development, we conduct a user study under IRB permission (#STUDY00010080). In particular, we recruit our subjects from a pool of CTF players and a pool of security analysts who have extensive experience in debugging Linux kernel and writing kernel patches. Our primary recruitment method is to invite participants through emails. In our invitation email, we describe our study objective as a paid task that quantitatively measures the time spent on exploit development and qualitatively surveys the difficulty in writing an exploit. In our email, we also require all the participants to ❶ have experience in exploiting Linux kernel vulnerabilities, ❷ be familiar with the commonly adopted exploitation method discussed in this paper, and ❸ be comfortable for conducting this study online. To evaluate whether applicants qualify for this study and form groups, we asked all applicants to sign the participation agreement and complete a self-assessment form (see Figure 3.6 in Appendix 3.9.5). The collected self-assessment forms indicate that, among the 8 participants, 6 subjects meet with our requirements. Regarding the expertise level, 4 subjects have about three years of experience in exploiting kernel vulnerabilities (high expertise), and the rest 2 subjects have about one year of the corresponding experience (moderate expertise). Based on these responses, we randomly assign each group with 2 highly skillful subjects and 1 subject with moderate experience. From the highly skillful subjects, we randomly picked one from each group as the leader to fill in the short survey form during the experiment.

We only include Linux kernel vulnerabilities in the experiment because the Linux kernel is fully open-sourced and thus we avoid introducing the reverse engineering skillset which is noise to our experiment. While conducting the user study, it is possible that some participants have already established the prior knowledge for some Linux kernel vulnerabilities or already known some public exploits that leverage elastic objects to bypass `KASLR`. As a result, to prevent the measurement bias introduced by these prior knowledge, we design our kernel vulnerability sets without those kernel vulnerabilities that already covered by the participants' knowledge base (Question 6 in self-assessment form, Figure 3.6). In the the first row of Table 3.9, we list the 5 vulnerabilities satisfying our selection criteria.

For both groups, we first gave the 5 vulnerabilities and their corresponding PoCs that trigger the vulnerabilities but not perform exploitation. Then, we asked both groups to

develop exploits to bypass KASLR by using the exploitation method mentioned in this paper. We provided the kernel objects with function pointer for the two groups to help them leak the base address. For Group A, we also equipped them with our tool ELOISE, which assists them in searching elastic objects and pairing vulnerabilities with objects accordingly.

In this experiment, we kept track of the performance of two groups through short surveys every 30 minutes and thus evaluated the utility of ELOISE both quantitatively and qualitatively. As is shown in Figure 3.7, the survey partitions an exploit development into 3 critical stages – ❶ vulnerability capability exploration, ❷ elastic object identification, ❸ memory layout manipulation. While receiving the survey, the participants need to specify the stage of their exploit development and report their progress at that stage. This short survey could be completed in one minute without intervening the exploitation development too much. If completing one stage of exploit development, the participants also need to turn in their results. From the response to the survey inquiry, we roughly measured the time spent at each stage and thus quantify the utility of ELOISE. At the end of the user study, we also handed out a post-test survey (see Figure 3.8) through which we qualitatively evaluate the utility of ELOISE and understand the challenges in exploit development. Due to the space limit, we leave the three survey forms in Appendix 3.9.5.

3.5.2 Results of Experiment I

Falsely identified & missing elastic objects (FP/FN). By using ELOISE, we track down 97 elastic kernel objects tied to 98 disclosure functions on Linux, FreeBSD, and XNU. We compare these objects and disclosure functions with those randomly sampled and manually confirmed. We find our manually audited objects and disclosure functions are a subset of those pinpointed by ELOISE. This discovery cannot directly conclude zero false negatives because the kernel's scale limits our ability to manually audit all objects. However, it implies the false negatives of ELOISE are minimal.

For 97 kernel objects that ELOISE identifies, we confirm 74 as the true positives, which indicates the false positives are moderate, and the reporting results of our proposed method is valuable. For those falsely identified elastic objects, we further explore the root cause and discover the false positives root in the inaccurate kernel call graph construction. For example, for the kernel objects `probe_resp` and `ctl_table`, the allocation of which can only occur in the functions `ieee80211_set_probe_resp()` and `register_leaf_sysctl_tables ↪ ()` when hardware devices are plugged in, ELOISE mistakenly links these objects with unprivileged system calls.

Cache	Struct	Potential	Privilege	Constraints
FreeBSD				
kmem.32	i40e_nvm_access	H	∅	[12, 16) < 4097
Linux				
kmalloc-8	ipv6_opt_hdr	H	∅	[1, 2) < Arg
kmalloc-16	ldt_struct	H & A	∅	[8, 12) < 65536
	ip_options⋆	anchor1: H anchor2: S	∅	anchor1: [8, 9) < Arg anchor2: [8, 9) ≠ 0
kmalloc-32	fb_info	H	NET_ADMIN	[768, 776) = kaddr
	ip_sf_socklist⋆	H	∅	[4, 8) ≠ 0
	cache_reader †	H	∅	[0, 8) ≠ cache_detail.20
kmalloc-192	cfg80211_scan_request⋆	H & A	NET_ADMIN	[24, 32) ≠ null
XNU				
mbuf	mbuf	H	∅	∅

Table 3.1. Elastic kernel objects sampled from Table 3.4∼ 3.6. The "Potential" column specifies the potential that the object provides for a vulnerability. H and S indicate the potential of leaking data from the heap and stack region, respectively. A indicates the potential of performing arbitrary kernel read. In the "constraints" column, ∅ denotes data disclosure imposes no critical constraints. `Arg` represents a system call argument under a user's control; `kaddr` stands for any valid kernel address; `cache_detail.20` indicates the 20^{th} field of the variable in the type of `struct cache_detail`.

Objects' exploitability & pervasiveness. For the elastic kernel objects surviving from our examination (i.e., true positives), we sample a small amount from Table 3.4∼3.6 and list them in Table 3.1. The results in both tables specify the kind of exploitation the elastic object could potentially facilitate. As we can observe from Table 3.4∼3.6, for nearly all kernel objects identified (70 out of 74), they can potentially facilitate a vulnerability to disclose data from kernel heap and thus bypass exploitation mitigations such as `KASLR` and heap cookie protector. For 28 elastic kernel objects, we observe they can potentially perform arbitrary kernel read because these objects enclose both the length field and a pointer referencing the elastic buffer. For a small number of elastic objects (5 out of 74), they can provide a vulnerability with the potential to overread data from kernel stack and thus leak stack canary accordingly. By manually examining kernel code, we note that `Linux`, `FreeBSD`, and `XNU` use these kernel objects widely (with 39,483, 44,956, and 22,307 sites allocating or using one of these objects in `Linux`, `FreeBSD`, and `XNU`, respectively). Following all these observations, we safely conclude the elastic kernel objects and their usage are pervasive in three different kernels.

Elastic objects that require high privilege. In Table 3.1, 3.4, 3.5 and 3.6, we

also specify the privilege needed for reaching out to these objects. As we can observe, most of the elastic kernel objects (60 out of 74) require no permission for allocation and information disclosure (indicated by $\emptyset$ in the table). For the remaining kernel objects, either their allocation or consecutive information disclosure requires the privilege such as `CAP_NET_ADMIN` or `CAP_AUDIT_READ`. By default, the kernel does not grant both of these permissions to an ordinary user. However, this does not mean these objects are not helpful for exploitation because a recent research [70] has already demonstrated that an ordinary user can create a user namespace and thus naturally bypass the permission check accordingly.

Objects' cache/zone coverage. In Table 3.1, 3.4, 3.5 and 3.6, we also categorize the elastic kernel objects based on the cache or zone to which they belong. As we can observe from Table 3.4∼3.6, the identified objects cover most of the general and some special caches/zones (e.g., `kmalloc-16384` in Linux, `mbuf` in FreeBSD, and `pipe_zone` in XNU). For some objects that enclose the elastic buffer, their size can vary. Therefore, the kernel can allocate them to general caches/zones with the size greater than that specified in the table. In Table 3.1, 3.4, 3.5 and 3.6, we also highlight these objects with a star symbol. These cache/zone-flexible kernel objects (18 out of 74) could significantly enrich the availability of kernel objects for exploitation and thus potentially escalate the exploitability of a vulnerability.

Constraints tied to objects & their security implication. In Table 3.1, 3.4, 3.5, and 3.6, we finally specify the constraint set that one has to satisfy in order for disclosing kernel data to the userland successfully. As we can observe, there are only one kernel object `ip_options` that contains more than one set of constraints tied to different leaking anchors. It indicates that, except for this object, every elastic object discloses kernel data from only one leaking anchor. As such, as we can observe from Table 3.1, only the object `ip_options` provides a vulnerability with the potential to disclose data not only from the kernel heap but also from the kernel stack.

From the column "Constraints" in Table 3.1, 3.4, 3.5 and 3.6, we can also observe that there are a few kernel objects (marked with a dagger symbol), the constraint sets of which involve variables. As we specify in Section 3.4, when pairing objects with vulnerability, we conservatively discard the paths associated with these constraints and ignore the corresponding kernel objects accordingly. While this inevitably reduces the total number of elastic objects available for exploitation, their influence upon exploit mitigation bypassing is negligible because the elastic objects falling into this category are minimal (10 out of 74).

CVE-ID or Syzkaller-ID	Capability	Suitable objects #	Security Impact
FreeBSD			
2016-1887	zone_mbuf:[0, 256)=*	1	BA, AR [6]
Linux			
bf96... [71]	ip_dst_cache:[64, 68)=*	0	NA
2018-6555	kmalloc-96:[0, 8)=kaddr kmalloc-96:[8, 16)=kaddr	3	SC, HC, BA
2018-5703	NA	0	NA
2017-8890	kmalloc-64:[0, 8)=kaddr: [8, 16)=kaddr:[16, 18)<238:[18, 64)=*	12 + (1)	SC, HC BA, AR
2017-7533	kmalloc-96:[0, 11)=*:[11, 12)='\0'	2	HC, BA
2017-15649	kmalloc-4096:[2160, 2168)=*	0	NA
XNU			
2019-8605	kalloc.192:[0, 192)=*	4 + (1)	HC, BA, AR
2017-2370	kalloc.256:[0, 256)=*	3	HC, BA

Table 3.2. Exploitability summary sampled from Table 3.7. # in the third column indicates # of elastic objects useful for the exploitation of the corresponding vulnerability. # in the parentheses indicates # of elastic objects useful for exploitation, but the paths to their leaking anchors include variables. In the last column, SC, HC, and BA signify, the vulnerability could disclose stack canary, heap cookie, and base address, respectively. AR indicates it could perform arbitrary kernel read.

3.5.3 Results of Experiment II

Summary of effectiveness in bypassing mitigation. Due to the page limit, we only sample some vulnerabilities from Table 3.7 and show them in Table 3.2. The results in both tables indicate the exploitability of the vulnerabilities under the facilitation of elastic kernel objects. As we can observe from Table 3.7, about 67.5% (27 out of 40) vulnerabilities successfully demonstrate the ability to bypass not only KASLR but also heap cookie protector. Among these 27 vulnerabilities, 12 vulnerabilities also provide us with the ability to uncover stack canary and 8 vulnerabilities also exhibit the capability of performing arbitrary kernel read. These observations indicate that elastic kernel objects could generally make existing kernel protection futile.

Exploit diversity. From Table 3.2 & 3.7, we can also observe that for all exploitable

[6] FreeBSD has no heap cookie protection and thus no security impact on heap protector.

vulnerabilities (except for the one indicated by CVE-2017-2370), there are more than one elastic kernel objects useful for exploitation and mitigation circumvention. For some vulnerabilities, the number of useful kernel objects is even larger (e.g., the one indicated by CVE-2017-7184 listed in Table 3.7). From the column "Capability" in both tables, we can discover that this richness results from ❶ the ability to corrupt kernel heap data in various caches/zones and ❷ the ability to overwrite elastic objects with less restriction.

In this work, we argue that the richness of the elastic objects could also be very disconcerting. On the one hand, it is because more elastic objects offer more opportunities to bypass mitigations (e.g., the vulnerability tied to CVE-2017-8890 demonstrating the ability to bypass various mitigations). On the other hand, it is because the richness potentially diversifies the way to craft a working exploit, making the pattern-based exploitation detection more challenging.

Analysis of failure cases. For the 13 vulnerabilities that ELOISE fails to pinpoint a suitable object, we perform a manual diagnosis and have the following discovery. As of the vulnerabilities tied to CVE-2018-5703, CVE-2018-12233, CVE-2018-1000112, and 3d67 [72], their PoC programs only demonstrate the ability to overwrite the data inside the vulnerable object. These vulnerabilities naturally fall short of the power of manipulating any fields in elastic objects. For vulnerabilities corresponding to CVE-2018-18559, CVE-2017-15649, CVE-2017-10661, CVE-2019-6225, and 422a [73], while their PoC programs demonstrate the ability to corrupt some data in general caches/zones, ELOISE cannot track down any kernel object with its length field overlapping with the corrupted region. For the vulnerability indicated by CVE-2018-4243, although ELOISE identifies objects overlapping with the corrupted region, the vulnerability provides only the ability to overwrite the length field with all zeros. Regarding vulnerabilities associated with CVE-2019-5603, CVE-2019-5596, and bf96 [71], the corruption happens in special c
↪ aches/zones in which no elastic objects are available for further exploitation.

3.5.4 Results of Experiment III

The A/B test experiment results show that Group A, equipped with ELOISE, succeeded in disclosing the base address of kernel image and bypassing KASLR for all 5 vulnerabilities, whereas Group B failed all. To get insights on this significant difference, we reviewed the surveys gathered from both groups while they analyzed vulnerabilities and developed exploits. For more detailed results, readers could refer to Appendix 3.9.5.

From our collected survey results, we found that it took roughly 0.5∼2 hours for the two groups to finish exploring the capability of one vulnerability. On the one hand, it

indicates that the two groups have no apparent difference in expertise level. On the other hand, this result shows that the capability exploration is not a heavy workload for security analysts with Linux kernel debugging experience. This conclusion aligns with our post-test survey, in which all participants stated that the capability exploration is not a challenging task when the corresponding PoC program is provided. They reported that after the PoC program is given, they can rely on the built-in debugging features in the kernel (e.g., `KASAN`) to quickly learn which cache is corrupted and which part of the memory is corrupted. Then, they can disable `KASAN`, use `GDB` to trace kernel execution, and thus determine the value of overwritten data under their control.

From our collected survey (summarized in Table 3.9), we also discovered that under the guidance of our tool `ELOISE`, Group A could complete the identification of elastic objects in less than 0.5 hours. In contrast, Group B was stuck in this identification stage and made no progress for any of these 5 vulnerabilities in 24 hours. This significant difference implies the `ELOISE`'s benefits in identifying elastic objects in the kernel and facilitating the exploitation. Following this observation, we also reviewed our post-test survey results. We observed that Group B thought the most challenging part of the exploitation is searching for the elastic objects and "felt frustrated" when facing the large codebase. For Group A, the post-test survey indicates that the most challenging part for exploit development is how to stabilize exploitation. They stated that unexpected kernel activities could intervene in the heap layout manipulation, making the disclosed data useless for bypassing `KASLR`. For example, Group A reported that their exploits leak all zero or other trash value from time to time. They put lots of effort into taming noisy kernel activities.

Based on our A/B test experiment, we argue that `ELOISE` is beneficial for exploit development. Although it is not an end-to-end automation tool, it could save security analysts' efforts to search for the elastic objects in the kernel and match them with the vulnerabilities. Using `ELOISE`, analysts could craft a working exploit more efficiently.

3.6 Defense Mechanism

In this section, we first discuss existing defense mechanisms. Then, we describe the design, implement and evaluation of our defense approach. Due to the space limit, we leave alternative defense mechanisms and future research in Appendix 3.9.4.

3.6.1 Existing Defense Mechanisms

Recall that the exploitation method discussed in this paper requires manipulating the kernel heap layout. Intuition suggests that the existing defense most likely to mitigate this exploitation method is heap freelist randomization [74, 75]. However, this approach cannot be an effective solution to our problem. One is because research [74] has already demonstrated that freelist randomization has no effects on mitigating exploitation against use-after-free and double-free vulnerabilities. The other is because there have already been many techniques proposed for bypassing this mitigation effectively (e.g., [58–60]).

In addition to memory layout manipulation, the exploitation method also needs to accurately locate and modify the length field in an elastic object. As a result, another possible existing defense is structure layout randomization [76], which shuffles the fields in a data structure at the compilation phase for preventing attackers from predicting the offset of sensitive data within the structure. In this work, we argue this defense is also not likely to be useful nor practical for our problem because it relies upon a random seed to perform randomization, and the protection of this seed is not trivial. For example, Linux distros [77] have to expose the random seed to their users for building third-party kernel modules. Besides, there are intensive on-going discussions about how to prevent a random seed from being accessed by unprivileged users on the same machine [78].

Apart from above defenses, both `Linux` and `XNU` kernel import "USERCOPY" checking [79] from PaX/Grsecurity team. This hardening ensures that the length argument does not exceed the size of cache/zone slot or stack frame. While this technique can mitigate the threat of some elastic objects, it suffers from two problems. On the one hand, it only enforces the length checking for `copy_{from/to}_user()` and `copyout()`. Other critical kernel functions for data transferring are not included. On the other hand, the legit length range is not restricted enough. It is still possible to leak sensitive data residing in the cache/zone slot or stack frame.

3.6.2 Our Defense Approach

Design. To mitigate the threat of elastic objects, we propose a new defense mechanism. It isolates elastic objects that `ELOISE` identifies into individual shadow caches/zones. To be specific, we create an isolated shadow cache/zone (e.g., `kmalloc-isolated-16`) for each general cache/zone (e.g., `kmalloc-16`). Using shadow caches/zones, we store elastic objects with the corresponding sizes. For example, the elastic object `ldt_struct` ↪ originally allocated in `kmalloc-16` will be assigned in `kmalloc-isolated-16` after the

isolation mechanism is enabled.

With the isolation mechanism, an adversary has little chance to leverage the vulnerability tied to other objects to manipulate the length (and pointer) field in elastic objects. Besides, common heap spray objects and kernel objects with sensitive information like function pointers are also isolated from the elastic objects. They could not be used for heap Fengshui and spraying. Admittedly, an elastic object itself could also be potentially vulnerable, which provides attackers with the ability to overwrite other elastic objects sharing the same isolated shadow cache/zone. However, as showed in the following section, vulnerable elastic objects are relatively fewer than non-elastic vulnerable objects. Therefore, the cache-isolation-based defense dramatically raises the bar for launching the exploitation method and reduces leaked data's significance. Note that our defense approach is very different from a recently proposed isolation mechanism – xMP [40]. xMP provides page granularity isolation. With such isolation granularity, overwrite/overread still work, and the exploitation method is still effective.

Implementation. We implemented and prototyped the proposed isolation-based mitigation in the Linux kernel. To be specific, we added the support of our mitigation method by creating the shadow caches and other caches at the boot time. Further, we modified the kernel source code by adding one more flag (e.g., `__GFP_ISOLATE`). In our implementation, we used this flag as an additional argument for the functions that allocate kernel objects (e.g., `kmalloc()`). Using this flag as an indicator, our modified kernel could determine if the allocated objects should be placed in the shadow caches. To determine which allocation should happen at the shadow cache, we use the output of `ELOISE`. It indicates which calls indeed allocate elastic objects. In this way, we can ensure that the elastic objects can be isolated from other kernel objects physically.

Performance Evaluation. We evaluate the performance overhead of the proposed mitigation on a machine with a 1.6 GHz CPU, 16GB RAM, and 500 GB HDD. Our hardened kernel is modified from a plain Linux kernel, which is v5.5.3 (same as the kernel version used in our previous experiment). We conducted the measurements using three sets of benchmarks. The first set is micro-benchmarks from LMbench v3.0 [80], which tests the latency and bandwidth of common system calls and I/O operations. The second set is macro-benchmarks from Phoronix Test Suite 9.8 [81], which runs five real-world applications. To prevent the overhead from being hidden behind sophisticated kernel execution, we especially designed the third set benchmark to stress-test the impact of our mitigation approach. This set of customized benchmarks uses the system call sequences to reach elastic object allocation and corresponding data leakage intensively. In our

experiment setup, we ran the three sets of benchmarks for three rounds and calculated the average. Due to the space limit, we list the detailed results in Table 3.8 (in the Appendix). Overall, we could observe that the performance overhead is negligible, with the average 0.19% performance drop. From Table 3.8, we could also find that for TCP socket I/O throughput, the hardened kernel even performs better (6.29% improvement). This fluctuation is presumably because our mitigation approach changes the hit rate of hardware cache, and the deviation of benchmarks adds uncertainty to the measurement.

Security Evaluation. We also evaluated the security of our proposed mitigation approach by re-pairing the vulnerabilities with elastic objects. For all Linux vulnerabilities shown in Table 3.7 (except for `CVE-2017-7184` and `CVE-2017-17053`), `ELOISE` no longer reports elastic objects available for performing the exploitation after our isolation mechanism is applied. It is because the elastic objects and vulnerable objects are mostly different. They are isolated into two caches. There is no longer a possibility to use vulnerable objects to manipulate the length field of an elastic object. For vulnerable objects in `CVE-2017-7184` and `CVE-2017-17053`, they are also elastic objects allocated in the shadow caches. Technically, they can be leveraged to overwrite data in the isolated caches and thus manipulate the length field of an elastic object for data disclosure. However, we argue that, even if this situation exists, it does not dilute our proposed defense method because the disclosed data is not likely to be useful for bypassing kernel mitigation. Taking the practice of circumventing `KASLR` using `CVE-2017-17053` as an example, to use the vulnerable object to reveal a kernel base address, in addition to leveraging the elastic object, an attacker usually has to identify a general object that encloses a function pointer. Then, the attacker needs to place the object in the same isolated cache. However, due to general and elastic kernel object isolation, such an object is no longer available for this isolated cache.

3.7 Related Work

The works most relevant to ours include escalating exploitability for bypassing exploit mitigation and designing automated methods to facilitating exploit development. Here, we summarize and discuss them below.

Escalating exploitability. Side-channel based attack [50, 51] is one common approach for exploitability escalation. Technically, it leverages hardware features to disclose critical information from the kernel. For example, by taking advantage of Intel TSX, Jang et al present a highly stable timing attack against `KASLR` [23]. Gruss et al propose to utilize

pre-fetch instructions to circumvent KASLR without triggering SMEP/SMAP protection [52]. Lipp et al introduce a method to read arbitrary kernel memory from userland by exploiting out-of-order execution in modern processors [53].

Another exploitability escalation method is through new exploitation approaches. For example, ret2dir [68] injects exploitation payload to physmap instead of user space to circumvent SMEP and SMAP. To bypass a series of Linux kernel protection (except for KASLR), the technique KEPLER [7] first transforms a control-flow hijacking primitive into a stack overflow. Then, it utilizes that overflow to enable an ROP attack in the Linux kernel. To avoid being caught by CFI, Data-only attack [82] exploits the vulnerable software through corrupting data flow instead of triggering critical control flow examination.

Facilitating exploit development. Researchers have proposed many exploitation automation techniques, ranging from the works that assemble exploits fully automatically (e.g., [13, 15, 35, 37, 38, 83, 84]) to the works that partially facilitate exploit development (e.g., [14, 16, 85–89]). However, they can barely tackle the unique challenges in the kernel. Presumably, as such, we recently witnessed many research efforts on the kernel exploitation facilitation.

For example, Xu et al propose two memory collision attack mechanisms [17] to assist heap spray in kernel Use-After-Free exploitation. Lu et al introduce a deterministic stack spraying exploitation method and a reliable exhaustive memory spraying technique to facilitate the exploitation of Use-Before-Initialization vulnerabilities in the Linux kernel [33]. Following this, Cho et al further extend the stack spraying method in [90]. To expedite the exploration of useful primitives for kernel Use-After-free exploitation, FUZE [4] searches exploitable machine states by utilizing under-context fuzzing along with symbolic execution. Chen et al design a capability-guided fuzzing technique that extracts the capability for out-of-bound write vulnerabilities in the Linux kernel [91]. To obtain the desired heap layout for kernel exploitation, SLAKE [6] proposes a method to navigate kernel objects and then an algorithm to elastic kernel layout automatically.

Uniqueness of our work. First, rather than exploiting hardware features, we explore exploitability escalation by exploiting the capability demonstrated by kernel vulnerabilities as well as the nature of kernel objects. Second, instead of developing yet another method to circumventing exploit mitigation such as SMEP/SMAP, we focus on the exploitation method that could bypass KASLR and heap/stack cookies or perform arbitrary read in the kernel. Third, different from the works that facilitate exploit development without considering mitigation circumvent, our proposed techniques facilitate an attacker's ability to assemble a working exploit with the capability of bypassing widely deployed kernel

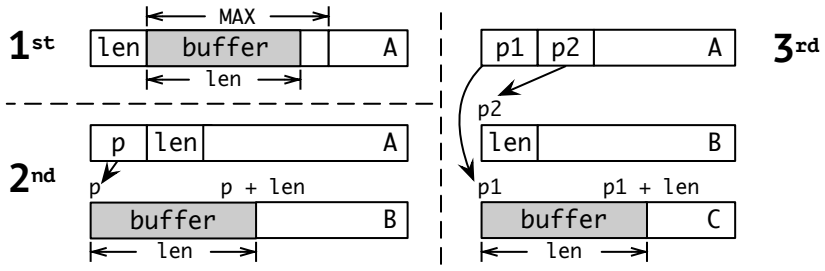


Figure 3.5. The alternative implementations of elastic kernel objects.

mitigation. Last but not least, rather than focusing on one particular type of vulnerability, this work targets exploitability escalation and exploitation facilitation for all types of vulnerabilities that could demonstrate data corruption on the kernel heap.

3.8 Conclusion

Using elastic objects to bypass kernel protection is a commonly adopted exploitation practice. However, no systematic research has been conducted to study the effectiveness of this exploitation method. As such, it has not yet raised sufficient awareness and motivates the development of a defense against such exploitation. In this work, we show that elastic objects could nearly always facilitate a kernel vulnerability to bypass exploitation mitigation such as `KASLR`, heap cookie protector, and stack canary. Taking a close look at existing kernel defense mechanisms, we discover that none can be useful or practical for hindering the threat of elastic kernel objects. Inspired by this finding, we introduce a new lightweight defense mechanism. We conclude that the threat of elastic objects can be, to some extent, mitigated if the kernel could place elastic objects into separated caches or zones.

3.9 Appendix

3.9.1 Implementation of Elastic Kernel Objects

The kernel object `xfrm_replay_state_esn` shown in Figure 3.1 is just one kind of implementation for an elastic kernel structure/object which encloses not only a length field but also the elastic buffer. Based on our manual analysis on `Linux`, `FreeBSD`, and `XNU`, we also discover three alternative implementations. Here, we summarize them below.

Types of Channel	Function Prototypes
User Space Memory Access APIs	unsigned long copy_to_user(void __user* to, const void* from, unsigned long n);
	int copyout(const void *kernel_addr, user_addr_t user_addr, vm_size_t nbytes);
Netlink	int nla_put(struct sk_buff* skb, int attrtype, int attrlen, const void* data);
	int nla_put_nohdr(struct sk_buff *skb, int attrlen, const void* data)
	int nla_put_64bit(struct sk_buff *skb, int attrtype, int attrlen, const void* data, int padattr)
	void* nlmsg_data(const struct nlmsghdr* nlh); void* memcpy(void* dest, const void* src, size_t count);
	void* nla_data(const structure nlattr *nla); void* memcpy(void* dest, const void* src, size_t count);
General Networking	void* skb_put_data(struct sk_buff* skb, const void* data, unsigned int len);
	void* skb_put(struct sk_buff* skb, unsigned int len); void* memcpy(void* dest, const void* src, size_t count);

Table 3.3. The summary of the FreeBSD/Linux/XNU critical kernel functions responsible for migrating data from kernel space to userspace. In the column of "function prototypes", the parameters in bold specify the addresses from which the kernel data originate. The parameters with wavy line indicate the amount of kernel data that an attacker can potentially disclose to the userland.

As is illustrated in the first example shown in Figure 3.5, the first alternative implementation is to have a large buffer defined in the middle of a data object and a field within that object indicating the actual buffer size or more precisely speaking the actual bytes used for storing data. At the time of defining the actual number of bytes used for storing data, kernel typically examines the length field and ensures it does not go beyond the boundary of the large buffer. However, we discover that the kernel does not always enforce this essential check at the time of reading data from that buffer. As such, it eases an attacker's ability to manipulate the length field and thus construct a buffer overread.

As is depicted in the second and third examples shown in Figure 3.5, the rest two alternative implementations do not enclose the length field and elastic buffer in the same kernel object. Instead, they place the length field and the elastic buffer in two individual kernel objects. The difference between the second and third implementation is that one implementation contains an explicit reference to the elastic buffer and, in contrast, the other implementation references the elastic buffer through a third intermediate kernel object.

3.9.2 Summary of Critical Kernel Functions

Modern OS kernels use virtual memory for isolation and provide separate address spaces for kernel and application processes. During kernel execution, however, userland processes need to inspect the on-going activities in the kernel and control kernel behavior from time to time. The kernel also needs to copy extra arguments from userland for processing and copy results back as a response to system call invocation. As a result, kernels design and implement various communication channels to facilitate data transfer between kernel space

and userspace. While many communication channels (e.g., `sysfs` [92] in Linux) require root privilege or high privilege (e.g., `CAP_SYS_PTRACE`) for data migration, unprivileged users in a local or remote machine can still obtain data from kernel space through unprivileged communication channels. In this work, the exploitation method takes advantage of these unprivileged channels to uncover kernel data to userland processes. In the following, we summarize these communication channels and categorize them into three types.

User Space Memory Access APIs [93]. User space memory access APIs are those functions like `copy_to_user()` and `copyout()`. For `copy_to_user()`, this API copies `n` bytes of data from kernel address `from` to user address `to`. When executing this function, the Linux kernel first ensures the destination memory region (i.e., (`to`, `to` + `n`]) is mapped in userspace. Then, the Linux kernel maps this region to kernel space and disables `SMEP/SMAP` protections (`PXN/PAN` on ARM) to avoid Oops or panic. For the function `copyout()`, it works similarly to `copy_to_user()`, coping `nbytes` data from `kernel_addr` to `user_addr`.

In addition to the two APIs mentioned above, kernels have other APIs or macros like `put_user_4` to transfer data from kernel space to userspace. These APIs are similar to `copy_to_user()` and `copyout()`. However, they determine the amount of transfer data at the compilation phase (e.g., 4 bytes in `put_user_4`). As such, the exploitation method cannot manipulate such APIs. In this work, we exclude such non-manipulable APIs and list only those manipulable ones in Table 3.3.

Netlink socket family. This channel uses the networking framework for kernel-user communication. Designed as a generic object model [94], `netlink` passes all kinds of status information about internal kernel activities to user processes (e.g., registration, removal of new devices, and hardware-related events). Although `CAP_NET_ADMIN` capability is generally required to build `netlink` socket, as is mentioned in Section 3.5, unprivileged users like containers can easily bypass this restriction by creating a user namespace (`CLONE_NEWUSER`). In Linux distributions (e.g., Ubuntu and Debian), the namespace is broadly deployed. Therefore, unprivileged users with `CAP_NET_ADMIN` capability can easily communicate with kernel through `netlink` message. In Table 3.3, we summarize all the kernel functions in this type. It should be noted that some communication channels are a combination of two sequential function calls.

General Networking. Different from the netlink socket family, which only establishes communication channels between kernel space and userspace in a local machine, APIs in general networking category enable remote data transfer. The Linux kernel sends and receives network packets by manipulating a `socket buffer`. Take the practice of sending

packets for an example. In the beginning, the kernel allocates and reserves a `socket buffer` to store user data. When protocol control and user data are passed through TCP/IP layers, the kernel prepends/appends them to `socket buffer` and, at the same time, performs data validation. After this entire procedure, the NIC driver copies the entire network data in the `socket buffer` to a hardware buffer. Besides, it capsules the kernel data (e.g., authentication associated data in the link layer) into network packets. As such, the APIs responsible for general networking operations provide adversaries with the ability to disclose data to remote hosts. In Table 3.3, we list all the functions in this type and specify the number of data that the kernel can capsule into network packets.

Linux					
Cache	**Struct**	**Offset (len/ptr)**	**Potential**	**Privilege**	**Constraints**
kmalloc-8	ipv6_opt_hdr	[1, 2)/NA	H	∅	[1, 2) < Arg , p ≠ null
kmalloc-16	sock_fprog_kern	[0, 2)/[2, 10)	H & A	NET_RAW	[0, 2) ≤ Arg
	policy_load_memory	[0, 4)/[4, 12)	H & A	∅	[0, 4) < Arg
	ldt_struct	[8, 12)/[0, 8)	H & A	∅	[8, 12) < 65536
	ip_options⋆	[8, 9)/NA	anchor1: H anchor2: S	∅	[8, 9) < Arg, anchor1 in put_cmsg() [8, 9) ≠ 0, anchor2 in do_ip_getsockopt()
	iovec	[8, 16)/[0, 8)	H & A	∅	∅
kmalloc-32	cfg80211_pkt_pattern	[16, 20)/[8, 16)	H & A	NET_ADMIN	∅
	user_key_payload⋆	[16, 18)/NA	H	∅	[16, 18) < Arg
	xfrm_replay_state_esn⋆	[0, 4)/NA	H	NET_ADMIN	∅
	ip_sf_socklist⋆	[4, 8)/NA	H	∅	[4, 8) < Arg, [4, 8) ≠ 0
	cache_reader †	[24, 28)/NA	H	∅	[0, 8) ≠ cache_detail.20
	tc_cookie	[8, 12)/[0, 8)	H & A	NET_ADMIN	[8, 12) ≠ 0
	cfg80211_bss_ies⋆	[24, 28)/NA	H	NET_ADMIN	[24, 28) ≠ 0, p ≠ null
kmalloc-64	sg_header	[4, 8)/NA	H	∅	∅
	inotify_event_info	[36, 40)/NA	H	∅	∅
	fb_cmap_user	[4, 8)/[8, 16), [16, 24), [24, 32)	S	∅	[4, 8) ≠ 0
	cache_request	[40, 44)/[32, 40)	H & A	∅	[20, 24) ≠ 0, [40, 44) ≠ 0
	msg_msg	[24, 32)/[32, 40)	H & A	∅	[24, 32) < Arg, [24, 32) ≤ 4048
	fname⋆ †	[44, 45)/NA	H	∅	[44, 45) ≤ compat_getdents_callback.3, p ≠ null, [32, 40) == null, [40, 44) < Arg
	ieee80211_mgd_auth_data⋆	[48, 52)/NA	H	∅	∅
	tcp_fastopen_context	[32, 36)/NA	S	∅	[32, 36) < Arg
kmalloc-96	request_key_auth	[48, 52)/[40, 48)	H & A	∅	[48, 52) < Arg, p ≠ null
	xfrm_algo_auth⋆	[64, 68)/NA	H	NET_ADMIN	∅
	cfg80211_wowlan_tcp	[28, 32)/[32, 40), [56, 62)/NA, [80, 84)/[84, 88)	H & A	NET_ADMIN	∅
	xfrm_algo⋆	[64, 68)/NA	H	NET_ADMIN	∅
	xfrm_algo_aead⋆	[64, 68)/NA	H	NET_ADMIN	∅
kmalloc-192	cfg80211_scan_request	[32, 36)/[24, 32)	H & A	NET_ADMIN	p ≠ null, [24, 32) ≠ null
	mon_reader_bin	[16, 20)/[24, 32)	H & A	∅	[16, 20) < 4096, [16, 20) ≠ 0, [16, 20) < Arg,
	cfg80211_sched_scan_request	[40, 44)/[32, 40)	H & A	NET_ADMIN	[8, 16) == kaddr, [48, 56) == kaddr, p ≠ null
kmalloc-256	mon_reader_text	[112, 116)/[116, 124)	H & A	∅	[112, 116) < Arg
	station_info	[120, 124)/[112, 120)	H & A	NET_ADMIN	∅
kmalloc-512	ext4_dir_entry_2⋆†	[6, 7)/NA	H	∅	[6, 7) ≤ compat_getdents_callback.3
kmalloc-1024	xfrm_policy	[372, 373)/NA	S	NET_ADMIN	∅
	fb_info	[816, 824)/[808, 816)	H & A	∅	[832, 836) == 0, [768, 776) == kaddr
kmalloc-2048	audit_rule_data⋆	[1036, 1040)/NA	S	AUDIT_CONTROL AUDIT_READ	∅
kmalloc-16384	n_tty_data	[8800, 8804)/NA	H	∅	[8800, 8804) < 4096
proc_dir_entry_cache △	proc_dir_entry †	[177, 178)/NA	H	∅	p ≠ null, [177, 178) ≤ compat_getdents_callback.3
seq_file_cache △	seq_file †	[24, 28)/NA	H	∅	[24, 28) ≠ 0, [24, 28) < Arg, [24, 28) < seq_file.1, [96, 104) == kaddr

Table 3.4. Elastic kernel objects identified and confirmed in Linux. For a detailed explanation of the listed results, readers could refer to the corresponding text in the Appendix. For the discussion of the results, readers should refer to the text in Section 3.5.

[7]FreeBSD has no heap cookie protection and thus no impact on heap protector.

XNU					
Cache	**Struct**	**Offset (len/ptr)**	**Potential**	**Privilege**	**Constraints**
kalloc.16	user_ldt	[4, 8)/NA	H	∅	[4, 8) ≤ 8192, [4, 8) ≤ Arg
	sockaddr⋆	[0, 1)/NA	H	∅	[0, 1) ≤ 255
	accessx_descriptor⋆	[0, 4)/NA	H	∅	[0, 4) ≠ 0
kalloc.32	msg †	[16, 18)/NA	H	∅	[16, 18) < msgrcv_nocancel_args.7, [16, 18) > 0
	audit_sdev_entry	[8, 16)/[0, 8)	H & A	∅	∅
kalloc.64	uio⋆	[16, 24)/NA	H	∅	[16, 24) ≤ 4096
	user_msghdr_x	[40, 44)/[32, 40)	H & A	∅	[40, 44) ≠ 0
kalloc.80	kauth_filesec⋆	[36, 40)/NA	H	∅	∅
	vm_map_copy	[16, 24)/NA	H	∅	[16, 24) ≠ 0
kalloc.192	nfsbuf	[112, 116)/[136, 144)	H & A	∅	[112, 116)>0
kalloc.224	audit_sdev	[140, 144)/NA	H	∅	∅
kalloc.1024	necp_client⋆	[800, 808)/NA	H	∅	[800, 808) < Arg
mbuf	mbuf	[24, 28)/NA	H	∅	∅
pipe_zone	pipe	[0, 4)/[16, 24)	H & A	∅	[104, 108) ≠ 0
NFS.mount	nfsmount	[196, 200)	H	∅	∅
mecache.necp.flow	necp_client_flow	[120, 128)/[128, 136)	H	∅	[120, 128) ≠ 0, [128, 136) ≠ null

Table 3.5. Elastic kernel objects identified and confirmed in `XNU`. For a detailed explanation of the listed results, readers could refer to the corresponding text in the Appendix. For the discussion of the results, readers should refer to the text in Section 3.5

FreeBSD					
Cache	**Struct**	**Offset (len/ptr)**	**Potential**	**Privilege**	**Constraints**
kmem.16	TWE_Param⋆†	[3, 4)/NA	H	∅	p ≠ null, [3,4) ≤ twe_paramcommand.3
	iovec	[8, 16)/[0, 8)	H & A	∅	∅
	sockaddr	[0, 1)/NA	H	∅	∅
kmem.32	i40e_nvm_access	[12, 16)/NA	H	∅	[12, 16) > 0, [12, 16) < 4097
	vpd_readonly	[16, 20)/[8,16)	H & A	∅	∅
	vpd_writeonly	[20, 24)/[8,16)	H & A	∅	∅
kmem.64	uio	[24, 32)/NA	H	∅	[32, 36) ≠ 2
	gctl_req_arg	[28, 32)/[40, 48)	H & A	∅	[24, 28) == 32
	ips_ioctl	[20, 24)/[8, 16)	H & A	∅	∅
kmem.128	usb_symlink	[80, 82)/NA	H	∅	p ≠ null, [80, 81)+[81, 82) ≤ 252
kmem.256	ucred	[52, 56)/[176, 184)	H & A	∅	∅
	shmfd	[0, 8)/NA	H	∅	∅
	iso_node†	[56, 64)/NA	H	∅	[56, 64) ≤ uio.2
	iso_mnt†	[48, 52)/NA	H	∅	[48, 52) ≤ uio.3
kmem.512	acc_fib†	[8, 10)/NA	H	∅	[8, 10)≤ aac_softc.61
	dirent	[20, 22)/NA	H	∅	[20, 22) ≤ Arg
kmem.1024	buf	[48, 52)/[24, 32)	H & A	∅	∅
kmem.2048	fw_device	[16, 20)/NA	H	∅	p ≠ null,[16, 20) ≥ 1024
mbuf	mbuf	[24, 28)/[8, 16)	H & A	∅	∅
TMPFS node	tmpfs_node	[40, 48)/NA	H	∅	∅

Table 3.6. Elastic kernel objects identified and confirmed in `FreeBSD`. For a detailed explanation of the listed results, readers could refer to the corresponding text in the Appendix. For the discussion of the results, readers should refer to the text in Section 3.5.

CVE-ID or Syzkaller-ID	Type	Capability	Suitable objects #	Security Impact
FreeBSD				
2019-5603	UAF	file_zone:[40, 44)=*	0	NA
2019-5596	UAF	file_zone:[40, 44)=*	0	NA
2016-1887	OOB	zone_mbuf:[0, 256)=*	1	BA & AR [7]
Linux				
1379... [95]	OOB	kmalloc-512:[0, 512)=*	10 + (1)	SC, HC, BA
3d67... [72]	OOB	NA	0	NA
422a... [73]	OOB	kmalloc-64:[0, 4)=0x8	0	NA
5bb0... [96]	UAF	kmalloc-192:[16, 24)=0, kmalloc-192:[48, 56)=kaddr	1	HC, BA
6a6f... [97]	UAF	kmalloc-1024:[0, 8)=kaddr	3	HC, BA
a84d... [98]	OOB	kmalloc-32:[0, 4)=*	1	HC, BA
bf96... [71]	UAF	ip_dst_cache:[64, 68)=*	0	NA
e4be... [99]	OOB	kmalloc-64:[0, 16)=*, [16, 24)=192, [24, 64)=0	6	SC, HC, BA
e928... [100]	UAF	kmalloc-256:[120, 128)=kaddr	1	HC, BA, AR
ebeb... [101]	UAF	kmalloc-1024:[15, 24)=kaddr	1	HC, BA
2018-6555	UAF	kmalloc-96:[0, 8)=kaddr, kmalloc-96:[8, 16)=kaddr	3	SC, HC, BA
2018-5703	OOB	NA	0	NA
2018-18559	UAF	kmalloc-2048:[1328, 1336)=*	0	NA
2018-12233	OOB	NA	0	NA
2017-8890	DF	kmalloc-64:[0, 8)=kaddr:[8, 16)=kaddr:[16, 18)<46:[18, 64)=*	12 + (1)	SC, HC, BA, AR
2017-7533	OOB	kmalloc-96:[0, 11)=*:[11, 12)='\0'	2	HC, BA
2017-7308	OOB	kmalloc-1024:[0, 1024)=*, kmalloc-2048:[0, 2048)=*	12 + (1)	SC, HC, BA
2017-7184	OOB	kmalloc-32:[0, 32)=*, kmalloc-64:[0, 64)=*, kmalloc-96:[0, 96)=*, kmalloc-128:[0, 128)=* kmalloc-196:[0, 192)=*, kmalloc-256:[0, 256)=*, kmalloc-512:[0, 512)=*	22 + (2)	SC, HC, BA, AR
2017-6074	DF	kmalloc-256:[0, 8)=kaddr:[8, 16)=kaddr:[16, 18)<238:[18, 256)=*	11 + (1)	SC, HC, BA
2017-2636	DF	kmalloc-8192:[0, 8)=kaddr:[8, 16)=kaddr:[16, 18)<8174:[18, 8192)=*	10 + (1)	HC, BA
2017-17053	DF	kmalloc-16:[0, 8)=*	4	SC, HC, BA
2017-17052	UAF	kmalloc-256:[0, 8)=kaddr, kmalloc-256:[8, 16)=kaddr	3	SC, HC, BA
2017-15649	UAF	kmalloc-4096:[2160, 2168)=*	0	NA
2017-10661	UAF	kmalloc-256:[192, 200)=kaddr, kmalloc-256:[200, 208)=kaddr	0	NA
2017-1000112	OOB	NA	0	NA
2016-6187	OOB	kmalloc-8:[0, 8)=*, kmalloc-16:[0, 16)=*, kmalloc-32:[0, 32)=* kmalloc-64:[0, 64)=*, kmalloc-128:[0, 128)=*	24 + (2)	SC, HC, BA, AR
2016-4557	UAF	kmalloc-256:[56, 64)=*, kmalloc-256:[64, 72)=*	3	HC, BA
2016-10150	UAF	kmalloc-64:[24, 32)=*, kmalloc-64:[32, 40)=*	3	SC, HC, BA, AR
2016-0728	UAF	kmalloc-256:[0, 8)=*	2	HC, BA
2014-2851	UAF	kmalloc-192:[0, 8)=*	2	HC, BA
2010-2959	OOB	kmalloc-256:[0, 256)=*	11 + (1)	SC, HC, BA
XNU				
2019-8605	UAF	kalloc.192:[0, 192)=*	4 + (1)	HC, BA, AR
2019-6225	UAF	kalloc.96:[8, 16)=*	0	NA
2018-4243	OOB	kalloc.16:[0, 8)=0	0	NA
2018-4241	OOB	kalloc.2048:[0, 2048)=*	5	HC, BA
2017-2370	OOB	kalloc.256:[0, 256)=*	3	HC, BA
2017-13861	DF	kalloc.192:[0, 192)=*	4 + (1)	HC, BA, AR

Table 3.7. The exploitability summary of kernel vulnerabilities. For a detailed explanation of the listed results, readers could refer to the corresponding text in the Appendix. For the discussion of the results, readers should refer to the text in Section 3.5.

3.9.3 Detail of Evaluation

Elastic object identification. Table 3.4∼3.6 list all the elastic kernel objects that we identify and confirm on both Linux and XNU kernels. To be specific, the results in the

table (from the column on the left to the right) indicate (1) the caches/zones to which each elastic object belongs, (2) the structural type of each elastic object, (3) the offset of the length field in each elastic object and, if applicable, that of the pointer referencing the elastic buffer, (4) the security capabilities that each object could potentially provide, (5) the privilege needed for using each elastic object to perform exploitation, (6) the constraints that an adversary has to satisfy in order for using each object for disclosing critical kernel data successfully.

In Table 3.4∼3.6, for clarification, we mark all the special caches/zones with a triangle symbol △ and highlight with a star symbol ⋆ the objects that could belong to all the caches/zones greater than they are specified in the table. Besides, we utilize the symbol ∅ to signify no privilege is required if an attacker performs exploitation with the corresponding object. Similarly, the same symbol ∅ in the constraint column indicates no restriction is imposed on the manipulated elastic object. In the struct column, we use the dagger symbol † to indicate the objects the constraint sets of which involve variables.

For the mathematical notations depicted in the constraint column, `Arg` signifies the argument of a system call, the value of which is under the attacker's control. `p`≠`null` indicates that a pointer `p` referencing the elastic object should not equal to a null value. The notations `compat_getdents_callback.3`, `msgrcv_nocancel_args.7`, and `cache_detail.20` ↪ all represent variables, the values of which are undecidable through static analysis. For example, `cache_detail.20` indicates the 20^{th} field of the object in the structual type `cache_detail`. The notation `kaddr` represents a valid kernel address. The formula `[768,776]== kaddr`, for example, indicates the value at the memory range `[768,776]` has to be a valid kernel address.

Last but not least, in the potential column, we use three different characters to represent the potential capability of an elastic object. The characters H and S indicate the object could potentially allow an adversary to leak data from heap and stack, respectively. The character A denotes the potential of performing an arbitrary kernel read.

Kernel vulnerabilities and their exploitability. Table 3.7 lists all the kernel vulnerabilities used for our evaluation. From the column on the left to the right, the results shown in the table indicate (1) the CVE-ID associated with the kernel vulnerability, (2) the vulnerability type into which the vulnerability was categorized, (3) the capability of the vulnerability summarized manually, (4) the total number of elastic kernel objects useful for the exploitation of the vulnerability, (5) the security implication tied to the exploitation.

In the capability column of Table 3.7, as is specified in Section 3.3.3, the capability of each vulnerability is represented in the format of `cache`:[`range|operator|value`, $\cdots$]. For example, the formula `kmalloc-96:[0,8)=kaddr` indicates the vulnerability offers the ability to manipulate the first byte of an elastic kernel object, and the manipulated value is a valid kernel address. As we can observe from the table, in addition to using the notation *kaddr* to denote a valid kernel address, we introduce the symbol $*$ indicating an arbitrary value. For example, `kmalloc-2048:[1328,1336)=*` signifies the vulnerability allows an attacker to assign an arbitrary value to the memory region `[1328,1336)`.

In the column marked with "Suitable object #", we specify the total number of elastic kernel objects useful for exploitation and mitigation circumvention. It should be noted that the number without parentheses denotes the total amount of objects associated with constraints involving no variables. The number with parentheses indicates the amount of those associated with constraints involving variables. As we discuss in Section 3.4, when pairing a vulnerability with elastic objects, `ELOISE` ignores the paths involving variables and discard corresponding kernel objects (if for that elastic object `ELOISE` identifies no other paths without variable involvement). This conservative design inevitably reduces the elastic objects available for exploitation. However, as we can observe in Table 3.7, even without using objects in this type, vulnerabilities can still find alternative objects for performing successful exploitation.

As is shown in Table 3.7, the vulnerabilities selected for our evaluation cover all types such as out-of-bound write (OOB), use-after-free (UAF), and double free (DF). In the last column of the table, for each vulnerability, we also specify all their security impacts. The notations we use for indicating these impacts are SC, HC, BA, and AR. They denote the capabilities of performing arbitrary kernel read (AR) as well as bypassing stack canary (SC), heap cookie protector (HC), and leaking base address or, in other words, KASLR (BA).

Vulnerability capability summarization. Table 3.7 also summarizes the capability of each vulnerability. In our evaluation, we extract the capability of each vulnerability from its PoC program based on the criteria below. If a vulnerability is in the type of out-of-bound write, we take its capability as the range of its overflow region and the corresponding value under its control. If a vulnerability is in the use-after-free category, we depict its capability based on how the vulnerability manipulates the freed object via the corresponding dangling pointer. If a vulnerability is in the category of double free, we treat its capability as the value under the control of the corresponding spray objects (e.g., `msg_msg` used in many publicly released exploits [102–104]). We will make all these

vulnerabilities available in virtual machines and release the exploits crafted by using elastic kernel objects.

Performance overhead of our proposed defense. Table 3.8 lists the performance of the Linux kernel with and without our proposed defense mechanism. The results are observed from three different benchmarks discussed in Section 3.6.2. We present the performance change in the column of "overhead". For latency measure, a negative percentage indicates performance improvement, whereas the positive percentage represents the performance degradation. For the throughput measure, the negative and positive rates mean performance degradation and increase, respectively.

3.9.4 Alternative Defense Methods

In addition to the defense proposed in Section 3.6, there are alternative defense solutions that might be useful for mitigating the threat of elastic kernel objects. In the following, we discuss these methods and analyze the challenges of their implementation.

The first possible solution is to build shadow memory for each of the elastic objects allocated in the kernel. In that shadow memory, we can record the actual size of the corresponding object. When the kernel discloses data in an elastic buffer at any leaking anchor, we could check whether the amount of the data migrating to the userspace is within a legitimate range. Since the construction of shadow memory inevitably introduces memory and performance overhead, the key challenge of this solution is to develop a lightweight method to minimize overhead in a systematic method.

Another possible solution is to design a mechanism to enable the integrity check for the data in the length field. For example, we could first expand each of the elastic structures and introduce a checksum field. Then, when the kernel allocates the corresponding object and initializes its length field, we could encrypt the length value and store it in the checksum field accordingly. With this design, at the time of disclosing data in the elastic buffer to the userland, the kernel could easily retrieve and scrutinize the checksum. However, the key challenge of implementing this idea is to ensure the addition of the checksum will not influence the usability of the kernel. For example, some elastic data structures designed for protocols have specific formats. After allocating objects in these types, the kernel references the data through corresponding offsets. If introducing additional field into such objects, one has to ensure the newly added checksum field does not incur incorrect data reference.

Benchmark	w/o defense	w/ defense	Overhead
LMbench - latency (ms)			
syscall()	0.3813	0.3796	-0.46%
open()/close()	1.5282	1.5290	0.05%
read()	0.4596	0.4529	-0.94%
write()	0.4125	0.4127	0.05%
select() (10 fds)	0.5114	0.5043	-1.39%
select() (100 fds)	1.1805	1.1774	-0.26%
stat()	0.7590	0.7600	0.14%
fstat()	0.4576	0.4584	0.19%
fork() + exit()	90.37	91.71	1.46%
fork() + execve()	255.18	257.85	1.05%
fork() + /bin/sh	858.86	863.77	0.57%
sigaction()	0.4182	0.4192	0.25%
Signal delivery	0.9337	0.9309	-0.30%
Protection fault	0.6914	0.7093	2.58%
Pipe I/O	3.7497	3.7951	1.87%
UNIX socket I/O	5.9786	5.882	-1.62%
TCP socket I/O	9.7846	9.6776	-1.09%
UDP socket I/O	6.5358	6.2251	-4.75%
LMbench - throughput (MB/s)			
Pipe I/O	4755.49	4753.89	0.03%
UNIX socket I/O	10385.07	10307.40	0.75%
TCP socket I/O	6327.32	6725.17	-6.29%
mmap() I/O	13559.20	13511.95	0.35%
File I/O	7707.81	7702.82	0.06%
Phoronix - latency (s)			
FFmpeg	14.01	14.46	3.22%
GnuPG	17.39	17.35	-0.22%
Phoronix - throughput			
Apache (request/s)	16700.23	16088.00	3.67%
OpenSSL (signs/s)	272.00	272.00	0
7-Zip (MIPS)	9970.00	9374.00	5.98%
Customized bench - latency (ms)			
sock_fprog_kern	28.54	28.30	0.09%
ldt_struct	33.81	31.48	-2.52%
ip_options	29.29	30.67	2.40%
user_key_payload	34.04	35.33	-2.87%
xfrm_replay_state_esn	29.69	30.06	1.67%
ip_sf_socklist	29.13	28.05	-3.78%
sg_header	31.84	30.75	-2.99%
inotify_event_info	32.68	31.77	0.42%
msg_msg	27.75	26.83	0.66%
tcp_fastopen_context	28.79	28.65	-1.04%
request_key_auth	81.23	79.88	2.98%
xfrm_algo_auth	30.32	29.50	-0.28%
xfrm_algo	28.64	28.43	-0.11%
xfrm_algo_aead	31.36	31.39	0.13%
xfrm_policy	31.07	30.53	-1.43%
Average			0.19%

Table 3.8. The performance of the Linux with and without our proposed defense. For latency measure, the smaller the number is, the better the performance is. For throughput, the larger the number is, the better the performance is.

1. Did you ever debug Linux kernel vulnerabilities before?
   a. Yes
   b. No
2. How long is your experience in Linux kernel security?
3. Did you ever write an exploit for Linux kernel vulnerability?
   a. Yes
   b. No
4. If you answer yes to 3, how do you usually bypass KASLR
   a. I assume no KASLR
   b. I use hardware side-channel
   c. I use information disclosure in `dmesg`
   d. I bypass KASLR if the vulnerability provides read primitive
   e. Others
5. Do you know or ever use elastic structures for KASLR bypassing?
   a. Yes
   b. No
6. Please list the CVE-ID or other ID or links of vulnerabilities you have debugged or drafted exploits for.

Figure 3.6. Self-assessment form.

1. Which vulnerability are your group working on?
2. For this vulnerability, which stage are your group on?
   a. Vulnerability capability exploration
   b. Elastic object identification
   c. Memory layout manipulation.

Figure 3.7. Short probing survey form.

3.9.5 More Details of User Study

Section 3.5.1 describes the setup of our user study. Here, we provide the three survey forms (i.e., Figure 3.6, 3.7, and 3.8) we used during the experiment. In Table 3.9, we display the time took for each group spent in solving the five vulnerabilities. As the short probing survey is collected every 30 minutes, the time is accumulated according to the survey results.

1. Do you think exploring the capability of vulnerability is difficult?
   a. Yes
   b. No
2. What methods do you usually use to explore the capability of vulnerability?
3. What do you think is the most challenging part of drafting the exploits?

Figure 3.8. Post-test survey form.

	2010-2959		2017-7184		2017-7308		2017-8890		ebeb... [101]	
Stage	A	B	A	B	A	B	A	B	A	B
1	1	0.5	2	2	2	2	1	1	1	1
2	0	4.5	0	2	0	3.5	0	4.5	0	3
3	1	NA	2	NA	2	NA	1	NA	1	NA

Table 3.9. Summary of our probing survey results. The # in the table indicates how many hours reported through the short probing survey that the two groups spent on the stage of a specific vulnerability. NA means the corresponding group participant did not enter the stage while they develop the exploit for a particular vulnerability.

Chapter 4
SLAKE: Leverage Exploitable Design pattern via Layout Manipulation for Exploitable Primitives

To evaluate the exploitability of investigated design patterns, a security analyst usually has to manipulate slab and thus demonstrate the capability of obtaining exploitable primitive (e.g., the control over a program counter or performing arbitrary read/write). 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 the evaluation. 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.

4.1 Introduction

Despite extensive code review, a Linux kernel, like all other software, still inevitably contains a large number of bugs and vulnerabilities [105]. 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 [106] (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., [4, 14–17, 33, 68, 88, 107]), and a recent research work indicates an analyst can use various test cases to explore the desired memory layout for vulnerability exploitation [85]. 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 4.6, even merely using `Syzkaller` (a kernel fuzzing tool) [24] 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., [4, 7, 33]), 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 4.6, 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.

[1] As we will clarify in Section 4.2.1, this work focuses on the ability to hijack control flow but not that to escalate privilege for a userland process.

4.2 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.

4.2.1 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` [18] and other debugging tools (e.g., `GDB` [63]), 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 [7, 43–48, 108–111]. 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 [112].

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., [15, 107]), 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.

4.2.2 Technical Background

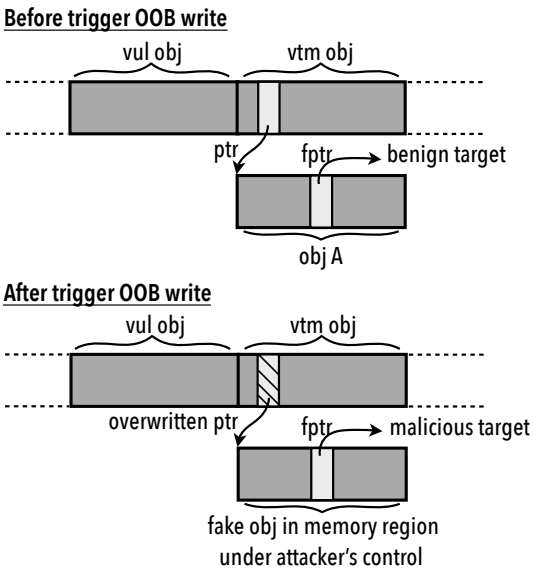
Here, we briefly introduce how SLAB/SLUB allocator works in Linux kernel, followed by the kernel exploitation techniques commonly adopted in the real world.

4.2.2.1 SLAB/SLUB Allocator

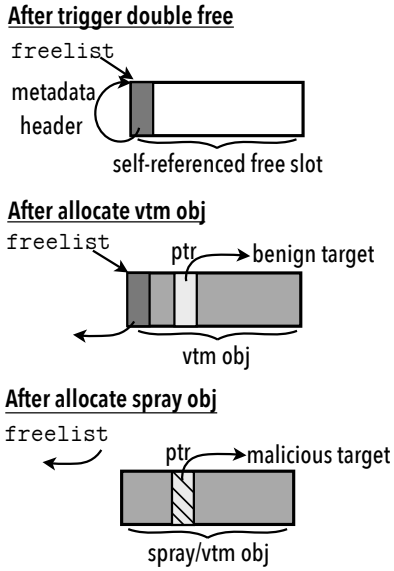
SLAB/SLUB allocator organizes physical memory in a unit of `cache`. Kernel objects in the same `cache` share the same type or have similar sizes. Inside each `cache` are a set of `slab`s which are contiguous pages. For each newly created slab, SLAB/SLUB allocator partitions it into multiple individual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

4.2.2.2 Kernel Exploitation Approaches

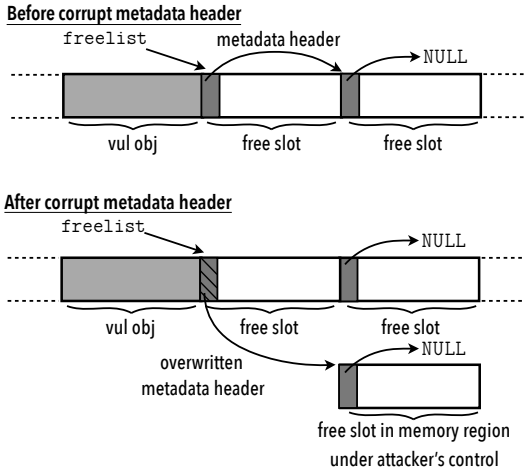
The kernel exploitation can be viewed as a three-step procedure. ❶ an adversary summarizes the type of a target vulnerability as well as at which memory addresses he could manipulate data freely (i.e., the capability of corrupting memory regions). ❷ The adversary determines the specific exploitation approaches to obtaining the ability to



Exploitation through OOB write



Exploitation through double free



Exploitation through corrupting metadata

Figure 4.1. The illustration of some kernel exploitation approaches.

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 4.2.1, 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 4.2.2.1 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 [42, 68] 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 4.2.2.1, 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 4.2.2.1 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.

4.2.3 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.

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.

4.3 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 [85].

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 4.6, 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.

4.3.1 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).

4.3.1.1 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 4.2.2.2, 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()`.

4.3.1.2 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.

4.3.1.3 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 4.2.2.1). 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 4.3.2.2. 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.

4.3.2 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 4.4, 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.

4.3.2.1 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 4.3.2.2, along with under-context fuzzing, these addresses could help us pinpoint the system calls that truly free or dereference the objects we intentionally allocate.

4.3.2.2 Syscall Identification

As is mentioned above and evaluated in Section 4.6, 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.

4.4 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 4.2.3). 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.

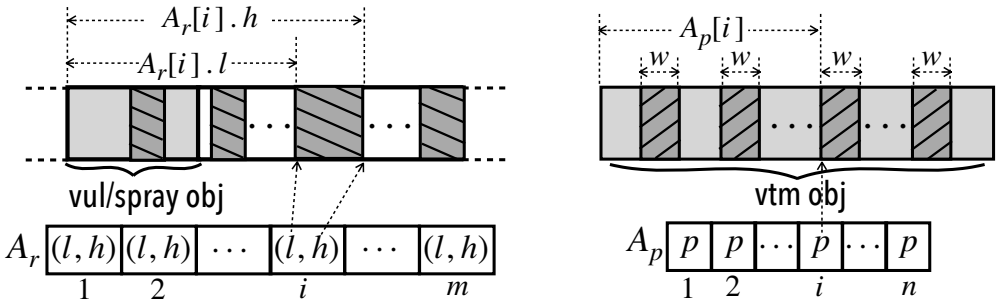
4.4.1 Matching Vulnerability Capability

Recall that Section 4.2.2.2 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.

4.4.1.1 Modeling Vulnerability and Object

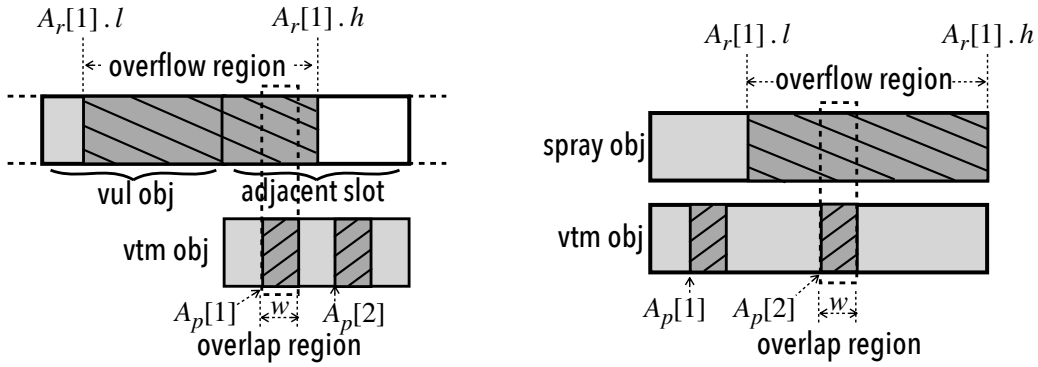
Given a vulnerability, we describe the capability of that vulnerability by using the array $A_r[m]$. This array contains m elements, each of which specifies a memory region under an attacker's control. As is depicted in Figure 4.4.1, each element contains a pair (l, h). It indicates the start and end 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., $\forall i, j,\ 1 \leq i, j \leq m, i \neq j$, $A_r[i].h < A_r[j].l$ 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



Vulnerability capability Kernel object property

Figure 4.2. Modeling vulnerability & object.



Pairing OOB write Pairing UAF/double free

Figure 4.3. Pairing vulnerability with objects.

located. As is illustrated in Figure 4.4.1, 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.

4.4.1.2 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

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

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 4.4.1).

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 4.4.1).

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.

4.4.2 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 4.2.3, 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

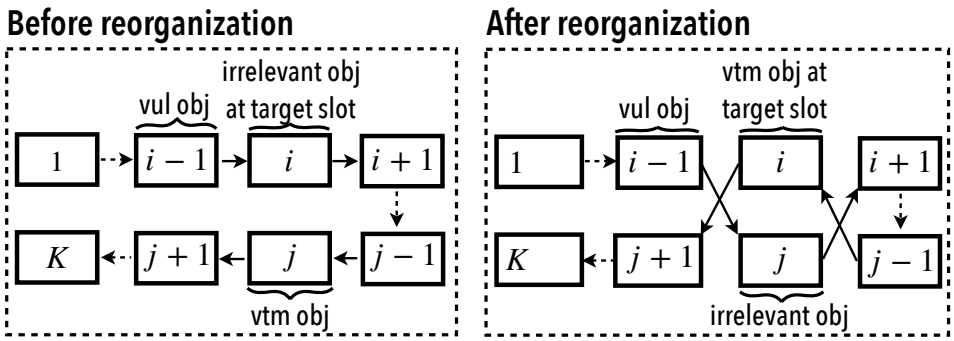


Figure 4.4. Reorganizing occupied slots.

to deallocate the object without side effects.

It should be noted that we perform defragmentation [17,113] 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 4.4, 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 4.2.2.2, we should allocate a victim object to the i^{th} slot. However, as is shown in Figure 4.4, 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` [114] 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 4.4, 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 4.4, 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.

4.5 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 [115]. 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 4.3.1, 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` [116], 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 on` and `select`. In total, the two LLVM passes contain about 2,000 lines of C++ code, capable of running on LLVM 6.0 [117].

Dynamic analysis. As is mentioned in Section 4.3.2, 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` [118] 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 4.3.2, 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 [24] with the integration of Moonshine [119] – 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 4.3.1, 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.

4.6 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.

4.6.1 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

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 4.1. 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.

system, network, time management, interprocess 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 [4,7,17,68] 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

[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.

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 4.2. 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.

searched the CVE list [121] 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 [6] to foster future work.

4.6.2 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 [116], 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 [122]. In Table 4.1, 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

CVE-ID	Type	Exploitation Methods			
		I	II	III	IV
N/A [120]	OOB	5 (1⋆)	-	-	5 (0)
2010-2959	OOB	13 (1⋆)	-	-	13 (0)
2018-6555	UAF	-	1(1⋆)	-	-
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 4.3. 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 ⋆ denotes the objects used in the public exploit are not enclosed in the exploits developed through `SLAKE`. I $\sim$ IV indicate the aforementioned exploitation methods pertaining to OOB, UAF, Double Free and metadata manipulation, respectively.

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 4.1, 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 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

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

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 4.2, we show the comparison results. As we can observe, SLAKE successfully identifies not only all the 10 objects from the public 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.

4.6.3 Evaluation of Exploit Development

Comparison between public and SLAKE-generated exploits. Table 4.3 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.

In addition, we can observe that, for those vulnerabilities that have not yet demon-

[5]As we mentioned in Section 4.2.1, 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.

strated 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 unexploitability 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 4.2, 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 5.9, we will explore technical approaches to tackle this issue in the future.

4.7 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 [123] and Android [124]). 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 [125], buddy system [126]) or those used in userland (e.g., ptmalloc [127]), 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 4.3.1, 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 inter-procedural 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 4.2, security researchers have developed other approaches for exploiting some special cases (e.g., [48, 49]) 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 4.2, 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 4.6, 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.

4.8 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 [17]. 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 [33]. 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 [4] 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 [7] 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 [15, 107]. 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 [14]. 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 [36]. 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 [16, 87]. 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 [37, 38, 83]. 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 [85]. Recently, Ispoglou et al. proposed to automate data-only attack to bypass control-flow integrity in userland [88]. 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.

4.9 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.

4.10 Appendix

4.10.1 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.

4.10.2 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 4.5.

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 4.4.1. 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 4.4.2. 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 4.5. An example illustrating the database and its usage

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, $-10 \leq i \leq 10, i \neq 0$ **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, $1 \leq i \leq m$, $1 \leq j \leq n$ **do**
    if $A_r[i].l \leq A_p[j] < A_p[j] + w \leq A_r[i].h$ **then**
      return True
  return False

Chapter 5
KEPLER: Evaluate Exploitable Primitives against Mitigations

In the previous chapter, we developed SLAKE which leverages the design pattern to obtain exploitable primitives such as control-flow hijacking primitives. However, obtaining primitives is insufficient to evaluate exploitability because of widely deployed exploit mitigations in an off-the-shelf modern Linux kernel. We therefore propose KEPLER to facilitate exploit generation by automatically generating a "single-shot" exploitation chain. KEPLER accepts as input a control-flow hijacking primitive and bootstraps any kernel ROP payload by symbolically stitching an exploit chain taking advantage of prevalent kernel coding style and corresponding gadgets. Comparisons with previous automatic exploit generation techniques and previous kernel exploit techniques show KEPLER effectively facilitates evaluation of control-flow hijacking primitives in the Linux kernel.

5.1 Introduction

Software bugs may have extremely serious consequences, especially for the OS kernel, where they could be fatal to the reliability and security of the entire OS because of the higher privilege that the kernel resides in and the abundance of hardware resources that the kernel has direct control of. Kernel bugs can lead to data leakage, privilege escalation and even persistant attacks [112].

One straightforward solution to minimize consequences of kernel bugs is to immediately patch all of the kernel bugs reported via mail lists and kernel fuzzers [24, 119, 128, 129]. In practice, considering the lack of manpower to patch all bugs timely, vendors typically prioritize their efforts to patch the bugs with severer security implications after assessing

their exploitability.

Existing solutions to assess exploitability efficiently are either applying simple heuristics to inspect crashing state of the bug (e.g. crash state and the output of address sanitizer [130]) or *automatic exploit generation* techniques [4, 14–16, 83, 85, 87, 107, 131], which could automatically generate a working exploit from PoC. The former approach lacks *accuracy* because it is simply based on prior knowledge regardless of the context constraints brought by the uniqueness of each bug. Capable of proving exploitability, automatic exploit generation is the ultimate choice of exploitability assessment.

The common workflow of automatic exploit generation systems are similar. In general we can divide them into the following two steps: ❶*exploit primitive identification* and ❷*exploit primitive evaluation*. In the first step, they search for pre-defined exploit primitive ("exploitable" states) based on the crashing path triggered by a PoC input. In the second step, after pinpointing an exploit primitive, they add exploit constraints and perform *constraint solving* to generate a concrete input to exercise a predefined exploit technique (e.g. `ret2libc`).

However, in order to generate working exploit for a control-flow hijacking primitive, there remains to be the following three challenges in the process of *exploit primitive evaluation* which limits the capability of automatic exploit generation techniques to target a complex real-world system such as the Linux kernel.

Challenge 1, exploit mitigation. Exploit mitigations are designed and introduced to reduce attack surface and raise the bar of exploitation. For a modern Linux kernel, many new hardware features [32,56], compiler-assisted instrumentation [132,133], sensitive data objects protection [57, 134–136] and virtualization-based hypervisor [137, 138] have been introduced as exploit mitigations. As a consequence, many kernel exploit techniques are no longer effective [1, 17, 139–141], despite the fact that heavier enforcement such as control-flow integrity (`CFI`) [142–145] is still *not* widely-adopted by major Linux releases due to performance concerns.

Challenge 2, exploit path pitfall. Side effects of exploit primitives could terminate exploitation in middle. Memory corruption, occurred along with an exploit primitive, can frustrate the attempt to trigger the primitive the second time, because an *exploit path* could unavoidably contain instructions triggering an unexpected termination of exploitation. Such termination can be a kernel panic of invalid memory access or an infinite loop in a kernel thread.

Challenge 3, ill-suited exploit primitive. Lack of pivoting gadget [146] and insufficient control over general registers can make an exploit primitive ill-suited. Although

some strong primitives have been proven exploitable and even Turing complete [88], there is still a gap between ill-suited exploit primitive and the requirement of mounting a certain exploit technique.

Considering the three challenges, it could be quite difficult to even manually generating an exploit for an exploit primitive. To address the above challenges, we propose KEPLER, an automatic exploit primitive evaluation framework for real-world Linux kernel vulnerabilities. KEPLER employs a novel and generic exploit technique designed for Linux kernel. Starting from an ill-suited *control-flow hijacking primitive* (`CFHP`), KEPLER overcomes the lack of stack pivoting gadget and manages to build an "one-shot" exploitation chain to bootstrap existing *Turing complete* exploit techniques such as `ROP` [147], with the ability to bypass mainstream kernel mitigations as well as detouring exploit path pitfalls.

To achieve this, KEPLER first enhances the exploit primitive to satisfy a minimal requirement of controlling dual registers (e.g. `rip` and `rdi` for `x86-64`) at the same time. Starting from the primitive, KEPLER generates an exploit which sequentially executes a chain of five gadgets. Specifically, KEPLER reuses those stack-based invocations of kernel I/O channel functions to leak and smash kernel stack of current process and execute arbitrary user-supplied `ROP` payload. KEPLER leverages *blooming gadget* to enhance register control for a control-flow hijacking primitive. KEPLER uses a *bridging gadget* to prevent unexpected kernel panic.

To generate exploits for each `CFHP` against arbitrary kernel binary, KEPLER operates in the following two phase: first, KEPLER statically analyzes the kernel binary for five categories of candidate gadgets. Then KEPLER starts kernel symbolic execution from a `CFHP`, and performs a *Depth First Search* (DFS) based gadget stitching algorithm over candidate gadgets.

Our evaluation of KEPLER shows that it is a lightweight and sharp blade for exploit primitive evaluation. To highlight the effectiveness of KEPLER, we compare KEPLER with existing AEG tools (e.g. `Q` [148], `FUZE` [4]) and KEPLER outperforms all of them in terms of generating effective kernel exploits. We argue that KEPLER as an automated tool has at least reached the same exploitation skill level of a professional kernel hacker by comparing to a set of state-of-art kernel exploitation methods.

This research work makes the following contribution:

- **Kernel One-shot exploitation**. We present a generic kernel exploit technique which transforms an ill-suited control-flow hijacking primitive into arbitrary payload execution under various constraints posed by modern Linux kernel mitigations. The

proposed technique exploits prevalent kernel coding style and corresponding gadgets and thus is hard to defeat. In addition, the one-shot nature of this technique makes it suitable for the vulnerabilities prone to unexpected termination.

- **Robust kernel automatic exploit generator**. We implement KEPLER on a set of tools including `IDA SDK`, `KVM/QEMU` and `angr`. It analyzes the Linux kernel binary, tracks down the useful kernel gadgets and automatically generates many exploitation chains for launching "one-shot" exploitation and bypassing kernel mitigations. It requires no kernel source code and can be applied to stripped kernel images. It can be applied to the latest Linux kernel release without further modifications.

- **Practical impacts**. We systematically evaluate the effectiveness and efficiency of KEPLER using 16 real-world kernel vulnerabilities and 3 recently-released CTF challenges. Given a kernel control-flow hijacking primitive, we show that KEPLER generally could generate tens of thousands of distinct exploitation chains with the ability to bypass kernel mitigations and perform successful exploitation. We show that KEPLER can output the first working exploit for a kernel vulnerability in less than about 50 minutes.

5.2 Background and Related Work

Exploit primitives indicate capabilities attackers could get according to different machine states. A control-flow hijacking primitive (`CFHP`) is a machine state that potentially deviates the legal control-flow graph. In the context of symbolic analysis, a control-flow hijacking primitive is a state with unconstrained or partial-constrained instruction pointer. Other common exploit primitives include arbitrary/restricted memory write primitive (`AMWP`/`RMWP`), and arbitrary/restricted memory leak primitive (`AMLP`/`RMLP`).

As is described above, this research work mainly focuses on two aspects – ❶ facilitating exploit primitive evaluation for even ill-suited exploit primitives and bypassing widely-deployed kernel mitigation mechanisms by designing a new exploit technique. ❷ developing a tool to automate the newly proposed kernel exploitation approach. As a result, the works most relevant to ours include those pertaining to exploit primitive identification, exploit primitive evaluation and kernel exploit techniques/mitigations. In the following, we describe the existing works in these three directions and discuss their difference from ours.

5.2.1 Exploit Primitive Identification

To assist with the process of finding a useful exploit primitive (e.g.control-flow hijacking primitive and memory write/leak primitive), there is a rich collection of research works. For example, using preconditioned symbolic execution and concolic execution techniques, Brumley et al develop `AEG` as well as `mayhem` to identify control-flow hijacking primitives for further exploitation. [15, 107, 131]. Shoshitaishvili et al develop a cyber reasoning system `Mechanical Phish` [149]. It is built on angr [37, 38, 83] and performs fuzzing and symbolic tracing for the PoC to identify exploit primitives. To efficiently explore state space for exploit primitives in Linux kernel, Wu et al propose an automatic technique that utilizes under-context fuzzing along with partial symbolic execution to explore `CFHP` and memory write primitive for UAF bugs [4]. To construct better exploit primitives with the capability of out-of-bound access, Heelan et al utilize regression tests to obtain the knowledge of how to perform heap layout manipulation [85]. To obtain better exploit primitives for stack Use-Before-Initialization vulnerabilities, Lu et al propose a deterministic stack spraying approach as well as an exhaustive memory spraying technique [33].

In this work, we do not focus on facilitating primitive identification. Rather, we assume an exploit primitive is already identified and our research endeavor centers around subsequent primitive evaluation phase.

5.2.2 Exploit Primitive Evaluation

In the primitive evaluation phase, a security analyst or an automatic exploit generation system tries suitable exploit techniques for the seemingly exploitable states identified before.

Initially, without considering Data Execution Prevention, such systems use straightforward techniques such as `ret2stack-shellcode` and `ret2libc` with a `CFHP` [107, 131, 149]. Taking N⊗X into account, Schwartz et al propose Q [148] to facilitate exploitation with a `CFHP` by automatically constructing a `ROP` chain. Our work addresses the problem of `ROP` bootstrapping without stack pivoting gadget and is orthogonal to automatic `ROP` chaining techniques [112, 148, 150–152] because we do not tackle the problem of `ROP` payload construction.

With the facilitation of forward and backward taint analysis, Mothe et al devise a technical approach to craft working exploits for simple vulnerabilities in user-mode applications [36]. Utilizing symbolic execution, Repel et al craft exploits with single

memory write primitive (e.g. unsafe unlink and lookaside list corruption) for those heap overflow vulnerabilities residing in the userland applications [16]. To facilitate primitive evaluation of kernel Use-After-Free exploitation, Xu et al propose two memory collision mechanisms [17] to unleash CFHPs.

Recently, some research works take CFI into consideration for primitive evaluation [88, 153–158]. For example, Ispoglou et al propose BOP [88] to facilitate evaluation of repeatable arbitrary memory write primitives (AMWP) by proving the Turing completeness under CFI along with common userspace mitigations. BOP assumes the existence of a dispatcher gadget. BOP also automates exploit generation. As a repeatable AMWP is almost "god-mode" of kernel exploitation, our work facilitates primitive evaluation for those weaker exploit primitives (e.g. non-repeatable and ill-suited CFHP) in real-world .

In this work, we also develop a tool for facilitating primitive evaluation. However, this research work is fundamentally different from the aforementioned works in at least one of the following aspects. First, without assuming a perfect exploit primitive (e.g. unlimited invocation of AMWP), we can faciliate exploit primitive evaluation for those usually ignored exploit primitive (e.g. ill-suited primitives and primitive that can only be triggered once). Second, rather than dealing with applications in the user space, our tool targets the Linux kernel where exploitation typically involves complicated operations and sophisticated security mitigation mechanisms are generally enabled. Third, rather than generating one single exploit for a target vulnerability, our tool automatically explores many possible exploitation chains and output various working exploits.

5.2.3 Kernel Exploit Techniques/mitigations

Initially, a CFHP in the kernel can directly execute shellcode in user-space because there is no isolation between user and kernel space (e.g. `ret2usr`). Supervisor Mode Execution Prevention (SMEP) [32] prevents kernel from executing userspace code. An attacker can use code-reuse attack. To set stack pointer to controlled payload, she uses the prevalent "pivot-to-userspace" gadget to pivot stack to userspace [159]. With adoption of Supervisor Mode Access Prevention (SMAP [56]), an attacker can no longer rely on a fake stack in userspace because userspace memory access is forbidden except during I/O channel functions. Because there is usually none intra-kernel stack pivoting gadget for a Linux kernel, an attacker usually chooses to disable SMAP by flipping corresponding bits in the `cr4` register [141]. However, the "`cr4`-flipping" attack typically rely on double CFHPs [141] and not suitable for a *none re-triggerable* CFHP. In addition, a virtualization-based hypervisor can detect `cr4` register modification by inspecting a `vmexit` and thus

mitigate such exploitation [137, 138]. `Ret2dir` [1] attack sprays the `physmap` region by calling syscall `mmap`, as the direct mapped physical memory is marked as executable previously, diverting a `CFHP` to land on physmap led to arbitary shellcode exectuion. However, with a kernel patch applied, these `physmap` pages are no longer executable.

To enforce CFI policy for the kernel, several kernel `CFI` solutions [143, 160, 161] have been proposed, however, these mitigations are not broadly adopted by major Linux release version such as CentOS, Ubuntu and Debian.

Data-only kernel exploit techniques directly use a memory write primitive (`MWP`) to modify sensitive kernel data objects such as process credentials, page tables and virtual dynamic shared object (vdso) [139]. However, mitigations have been proposed [162, 163] and deployed [133] to prevent these low-hanging fruits.

Note some memory write primitive can be transformed to a control-flow hijacking primitive by overwriting and invoking a code pointer in kernel's data section or in the heap [164].

Kernel address space layout randomization (`KASLR`) is widely deployed in order to present the attacker an unpredictable attack surface. However, due to its lack of timely re-randomization and coarse-grained nature (only randomizing section base address), an attacker do not even need an arbitrary read primitive [31, 165] to bypass `KASLR`. With a hardware side channel [22, 29], he can infer the coarse memory layout without leaking any kernel memory content. Despite kernel Page Table Isolation (`KPTI` [57]) removes some side channels with extra overhead, he can also uses a restricted memory leak primitive to infer the coarse memory layout [166]. Knowing coarse memory layout is enough for exploitations.

5.3 Assumptions and Threat Model

Our threat model consists of a modern Linux kernel protected by widely-deployed mitigations with a known vulnerability.

Mitigation setting. Similar to recent major Linux release versions, we make the following assumptions of kernel mitigations. A kernel has enabled `SMEP/SMAP` [56] protection. A kernel has enabled `KPTI` [57] protection. A kernel has enabled stack canary for all functions containing local variable [132]. A kernel has enabled protection to prevent direct modification over sensitive kernel data objects including process credential [133] and page table [162]. A kernel has enabled `KASLR`. However, A kernel does `not` enable a `CFI` enforcement such as `RAP` [160] because of performance concerns.

Available Exploit Primitives. We assume there is a PoC which triggers the vulnerability and leads to a control-flow hijacking primitive (CFHP). We assume the CFHP is already identified with the PoC though either manual analysis or dynamic analysis such as symbolic tracing, thus finding exploit primitives is orthogonal to our work and we can focus on evaluating the CFHP. We assume a restricted memory leak primitive (RMLP) to help infer coarse kernel memory layout (e.g. getting the base address of `.text` and `physmap` section). We do `not` assume the existence of an arbitrary memory write primitive (AMWP) which could write to arbitrary kernel memory address. We do *not* assume the existence of an arbitrary memory leak primitive (AMLP) that can dump arbitrary kernel memory content. Under the threat model, the content in arbitrary memory address such as the stack canary value of arbitrary kernel thread usually remains secret because the coarse kernel memory layout information does not reveal the stack canary value of a kernel stack. We also assume the location of current kernel stack remains secret although a RMLP might help leak a pointer to current stack under some specific vulnerability context.

5.4 Motivating Example

We illustrate the challenges in evaluating a CFHP on x86-64 with CVE-2017-8890 [167], a recent vulnerability in the Linux kernel.

5.4.1 Vulnerability and Exploit Primitive

The root cause of the bug is an Use-After-Free over an object `ip_mc_socklist` defined in Table 5.1. As is shown in Table 5.2, the UAF bug results in a dangling pointer `inet->mc_list` and `*iml` become an alias of the dangling pointer in line 5. In line 7, there is an UAF access by dereferencing `iml->next_rcu`. In line 9, `kfree_rcu(iml)` queues a callback denoted by `iml->rcu`. Function `rcu_do_batch` handles the previously queued callback when the CPU gets a chance to process the `rcu` callback list, thus triggers another UAF access to the `ip_mc_socklist` object in `rcu_reclaim()`. The loop from line 5 to line 10 will continue and add another callback if `iml->next_rcu` is not `NULL`.

The CFHP is due to an asynchronous `kfree_rcu` call over the dangling pointer `*iml`. To be specific, by manipulating the value in `iml->rcu_head` through a proper heap spray, a security analyst can get a CFHP lator in `rcu_reclaim()` because function `kfree_rcu()` (line 9) is designed to queue a `rcu` callback denoted by `iml->rcu_head`. Function `rcu_do_batch` will be executed in the future, it will iterate over a list of `rcu_head` added

by kfree_rcu(). The statement pertaining to the CFHP (line 13) allows the attacker to control rip (head->func) through heap spraying. At the the time of the control flow hijack, register rdi (head) points to a controllable memory region.

5.4.2 Challenges of Crafting Working Exploit

However, developing an exploit with the aforementioned CFHP is quite difficult because of the following challenges.

5.4.2.1 Challenge 1: Exploit Mitigations

Widely deployed kernel mitigations frustrate a large amount of straight forward exploit techniques. With the presence of SMEP/SMAP protection, it is impossible to directly launch a traditional ret2user/pivot2usr attack. The ret2dir attack [1] is also not suitable because the physmap region is no longer executable [168]. With the write-protection over sensitive data such as process credentail and page table, it is not possible to direct overwrite these data and escalate priviledge by converting the CFHP into a MWP.

A security analyst may think of the "cr4-flipping" attack which usually requires two CFHPs: one for flipping the cr4 register and the other to launch ret2user/pivot2usr. However, this is often infeasible because virtualization based hypervisor could easily detect the behavior of flipping cr4 register by inspecting a vmexit [137] [138]. Even if there is not protection over cr4 register, leveraging the "cr4-flipping" attack or a similar exploit technique relying multiple CFHP with this CFHP still faces the following challenge.

5.4.2.2 Challenge 2: Exploit Path Pitfall

Attempting to leverage the "cr4-flipping" exploit technique, a security analyst would expect two CFHPs to disable SMEP/SMAP and pivoting to userspace respectively. However, such a exploit technique could be imfeasible because of the exploit path pitfall after the first CFHP.

As is shown in Figure 5.1, after using the CFHP in rcu_reclaim to disable SMAP protection by invoking native_write_cr4(), an attacker encounters an unexpected termination: in previous execution, the loop from line 5 to line 10 in Table 5.2 queues another rcu callback denoted by head' (iml->next_rcu->rcu) which is not under our control and causes a kernel panic.

This kernel panic attributes to an invalid memory access. The heap spray technique used by the PoC does not allow an attacker to control first QWORD of iml (an ip_mc_socklist

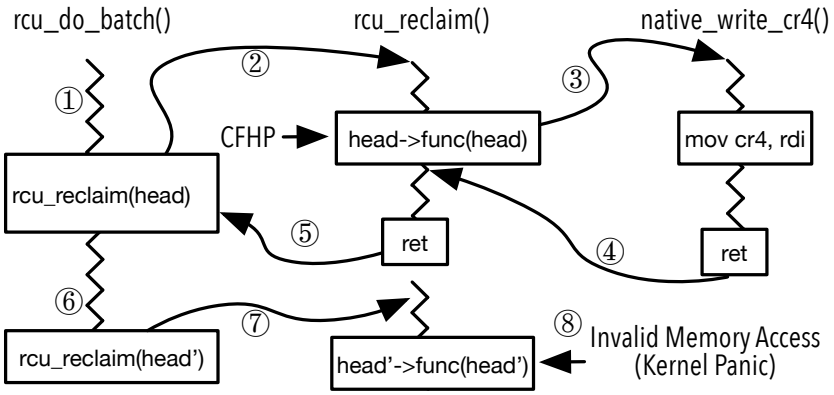


Figure 5.1. Demonstration of an exploit path pitfall in the motivating example. After applying the first `CFHP` to overwrite `cr4` register with `native_write_cr4()`, `rcu_reclaim(head')` is invoked but `head'` is unmanageable and panics the kernel.

object) because the first `QWORD` of an freed chunk becomes heap-metadata, thus `iml->next_rcu` becomes unmanageable.

A straight-forward solution to tackle the invalid memory access is adding extra constraints to ensure all related memory accesses are valid. However, as is pointed in [169], constraint solving for complex program usually incurs high cost (both cpu time and memory) and could fail because of constraint conflicts [14]. Instead of devoting extra resource to handle these pitfalls, an ideal exploit path should effectively detour them. As a result of lacking control of `iml->next_rcu`, the exploit path pitfall can not be simply prevented by techniques like adding constraints over the sprayed data and thus unavoidably panic the kernel. Although it is possible to tackle the problem by further tuning the PoC [4, 16, 87] and obtain a better exploit primitive, the showcased exploit path pitfall already hinders the evaluation of the `CFHP`.

In addition, an exploit path pitfall could also be attributed to un-successful heap spray. Due to the undeterminacy of kernel execution, there is not any heap spray technique with 100% success ratio. To get multiple `CFHP`s for a UAF vulnerability, an attacker may need to do multiple heap sprays and trigger a vulnerability multiple times, which could dramastically decrease the success ratio of the entire exploitation.

5.4.2.3 Challenge 3: Ill-suited Exploit Primitive

Facing the infeasibility of popular kernel exploit techniques, it is nature for a security analyst to use generic code-reuse technique such as `ROP` [147] as a second resort.

As pointed in [146], *stack pivoting* is a vital step for a `ROP` attack especially when

an attacker does not control content on the stack (e.g.the CFHP does not result from a stack overflow). In userspace, many heap exploits relying on a stack pivoting gadget (e.g. `swapcontext` in libc) to bootstrap a ROP attack.

It is however difficult in the target kernel to pivot stack pointer to a memory region under our control with this CFHP. The reason behind is two-fold. First, as is mentioned before, we can not simply pivot to a userspace with a traditional gadget such as `xchg eax,esp; ret` because of SMAP. Second, there is not a suitable stack pivoting gadget in a Linux kernel binary for this CFHP. Considering register `rdi` points to controllable area with this primitive, it would be great to have a gadget to overwrite `rsp` with a controllable memory address, such as gadget with form `xchg r**,rsp; ret`, `mov rsp,[r**]; jmp rxx` and `mov rsp,r**; ret` for consecutive payload [1]. Unfortunately, similar traditional stack-pivoting gadget does not exist or contains unavoidable exploit path pitfall (e.g. gadget `xchg rsp, r14 ; jmp rsp` could successfully pivot the stack pointer but inevitably panics the kernel) in our investigation across various modern linux kernel images.

Although previous works have demonstrated the power of code-reuse attack, mounting a traditional ROP attack for the CFHP seems surprisingly difficult because of the lack of stack pivoting gadget.

Without the capability of performing ROP attack, one may think of reusing other kernel functions. Unfortunately, there is also a problem of insufficient control over general registers because only `rdi` points to a memory region under control and other registers are not in control initially. We need to enhance this CFHP by controlling more general registers and perform subsequent exploitation.

5.5 Overview

To tackle the three challenges exposed by the motivating example, a security analyst needs to design a new exploit technique to turn a CFHP into a more exploit-friendly machine state based on the vulnerability context and his prior experience. Due to the lack of a ready-to-use exploit technique, he could expect a lot of debugging and manual efforts to explore possible exploit paths and improve his prototype exploit during the exploit development process and such practice could be extremely time-consuming and even fruitless.

In the following, we discuss the considerations that go into the design of KEPLER as well as the high level design of this framework to facilitate CFHP evaluation.

5.5.1 Requirements for Design

To support evaluation of a CFHP with working exploits, KEPLER should receive as input a state representing a CFHP and it should be able to output working exploits automatically under our threat model against arbitrary Linux kernel binary. To achieves the above ultimate goal, KEPLER should adopt an exploit technique which satisfies the following requirements.

First, an exploit technique is able to bypass all the widely-deployed mitigations enabled in the threat model. *Second*, taking potential exploit path pitfalls into consideration, an exploit technique should depend on only one control-flow hijacking primitive and detour these pitfalls to prevent an unexpected termination and make exploitation more reliable. The form of an exploit technique would be ideally similar to a "magic gadget[1]" which is previously mentioned in user-space exploitation [170], especially in the context of adversarial scenarios like Capture-The-Flag cyber competition. *Third*, an exploit technique should benefit from time-tested exploit technique such as ROP by efficiently bootstrapping traditional code-reuse attack in absence of stack pivoting gadget with an ill-suited CFHP. *Last but not least*, an exploit technique should be suitable for the automation framework. On one hand, it should be generic and not depend on any special code or feature which could be easily eliminated. On the other hand, the exploit generation phase should be easily automated - it should be a well-defined search problem over a search space of reasonable size.

5.5.2 High Level Design

To satisfy the requirements mentioned above, we design KEPLER to facilitate exploit primitive evaluation. KEPLER automatically generates an exploitation chain to *bootstrap* any kernel ROP chain with a single CFHP through "one-shot" exploitation.

Figure 5.2 shows how KEPLER automates the analysis task necessary to leverage a CFHP to produce an exploit in the presence of aforementioned challenges. Given a kernel state snapshot representing the CFHP, KEPLER enhances its power to construct a kernel stack-overflow by symbolically stitching several types of candidate gadget identified by static analysis on the kernel binary image.

As is mentioned before, our basic idea is to *bootstrap* a traditional ROP attack with a

[1]The term "magic gadget" means given a CFHP, one can instantly succeed in exploitation (e.g. getting a shell) by diverting control-flow to such gadget, thus boost exploit development and gain advantages in a game.

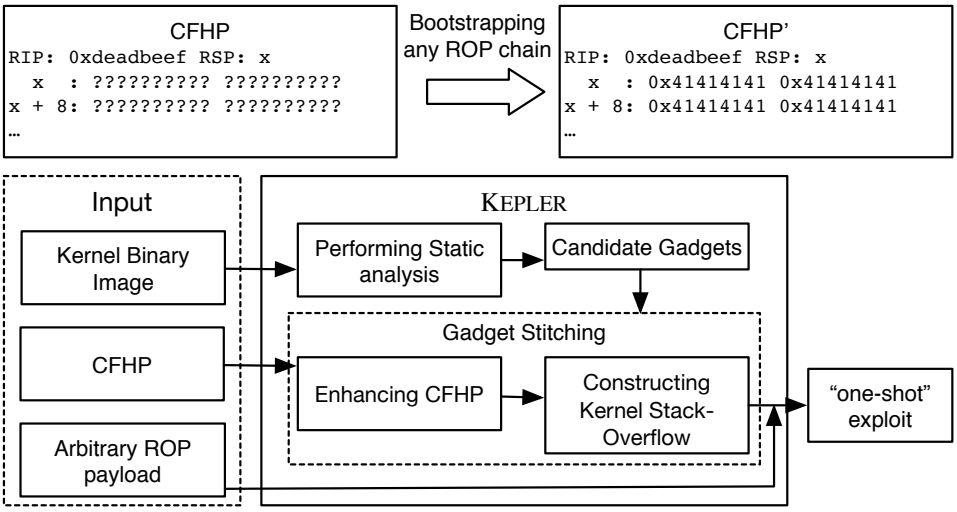


Figure 5.2. Overall of KEPLER's design.

CFHP in Linux kernel. At the high level, we achieve this by an "one-shot" exploitation chain which transforms a function pointer corruption based primitive (CFHP) into a stack-overflow based primitive (CFHP') as is shown in Figure 5.2.

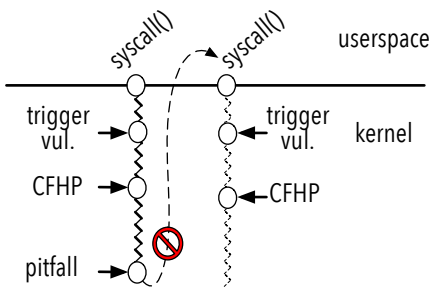


Figure 5.3. Conventional approach.

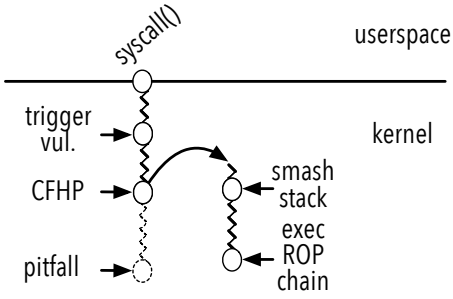


Figure 5.4. "One-shot" exploitation chain.

Figure 5.5. A comparison of two exploitation approaches; the conventional approach triggers a vulnerability twice but blocked by an exploit path pitfall and the other triggers the vulnerability only once.

Although there is not any gadget in Linux kernel which allows an attacker to directly escalate priviledge, The proposed "one-shot" exploitation is similar to "magic gadget" mentioned above in a sense that it only requires one CFHP and could reliably achieve the goal of arbitrary code execution in kernel context. To be specific, as is shown in Figure 5.4, the "one-shot" exploitation chain could finish exploitation with a single CFHP and thus is able to circumvent an exploit path pitfall after the return of CFHP which could cause an unexpected termination otherwise. We can benefit from a stack-overflow based

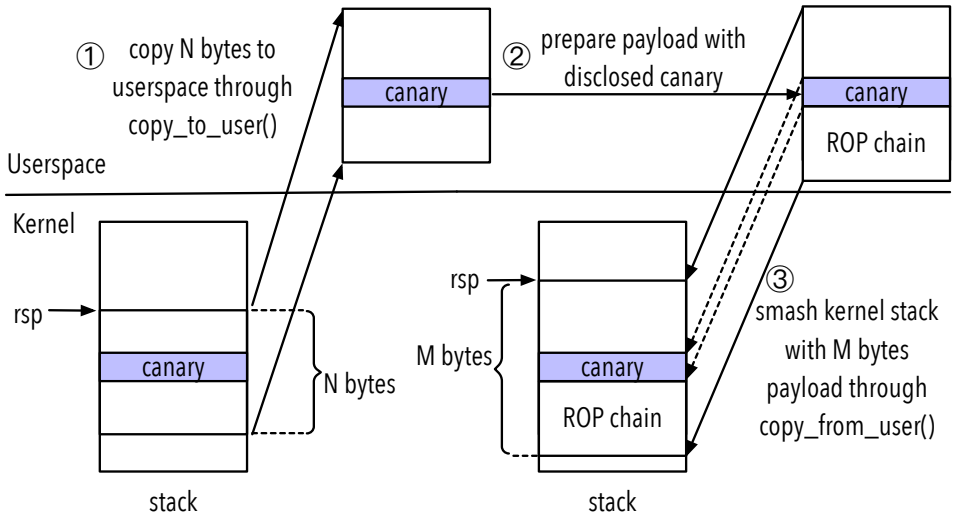


Figure 5.6. An overview of "one-shot" exploitation which discloses kernel stack canary and then smashes the kernel stack with arbitrary user-supplied ROP payload.

CFHP because it allows us to place arbitrary ROP payload on current kernel stack without any stack pivoting gadget. Given the scarcity (or non-existence) of intra-kernel stack pivoting gadget, we argue that constructing a kernel stack overflow is the most generic approach to perform a kernel ROP attack.

5.6 Design

In this section, we describe the exploit technique adopted by KEPLER and the insights behind the exploit tehcnique. As is mentioned before, KEPLER uses a CFHP to construct a stack overflow and bootstraps arbitrary ROP payload.

Our "One-shot" exploitation technique builds on two key ideas of *breaking isolation with I/O functions* and *improving exploit success ratio with a single CFHP*.

First, we can break isolation between kernel-space and user-space by abusing kernel I/O functions. The insight behind is such data channels are born to bypass SMAP which prohibits user-space access because SMAP is explicitly and temporarily disabled during execution of these functions. Figure 5.6 illustates a practice of reusing I/O functions to construct kernel stack overflow by first leaking kernel stack canary with `copy_to_user` and then smashing kernel stack with `copy_from_user`.

Second, "one-shot" exploitation can be achieved through stitching various kernel function gadgets. We can enhance register control for a CFHP with *blooming gadget* - a prevalent family of function gadgets in Linux kernel. We can detour exploit path pitfalls

with a *bridging gadget*.

5.6.1 Constructing Stack Overflow

There is a family of prevalent stack smashing gadgets inside Linux kernel, we observe they could greatly aid constructing stack-overflow via taking intrinsic *short return path* triggered by a page fault. However, such gadgets can not be directly used because initial CFHP does not have enough register control and the presence of a stack canary. We address the two problem in Section 5.6.2 and 5.6.3.

5.6.1.1 Looking into Stack-Smashing Gadget

We present stack-smashing gadgets which relies on functions that serve as data channel between user-space and kernel-space. The gadget could aid exploitation by transporting payload of arbitrary length from user-space to kernel stack, without assuming the stack location is already known.

As is named after, `copy_from_user(void* dst, void* src, unsigned long length)`[2] is a heavily used I/O kernel function which migrates data from the user to kernel space. Recall that SMAP prevents kernel code from accessing user-space address, and to temporarily bypass the restriction, `copy_from_user` uses a special instruction STAC to set AC flag in EFLAGS register before accessing user-space memory, thus allows the subsequent instructions to explicitly access user-space memory. Once the copy is finished, instruction CLAC is executed to re-enable SMAP.

As is specified in Linux kernel implementation, the function `copy_from_user()` takes as input three arguments - `dst`, `src` and length - which indicate the destination, source and length of the data that need to be copied from the user to kernel space.

From perspective of an attacker, a kernel stack overflow could be caused if he lets the CFHP jump to the site right before the invocation of `copy_from_user` (line 2 in Table 5.5) and the machine state satisfies the following three requirements: ❶ parameter `dst` (e.g. `rdi`) points to current kernel stack, ❷ parameter `src` (e.g. `rsi`) points to any user-space address so that its content is controllable by an attacker, ❸ parameter `length` (e.g. `rdx`) is greater than the size of current stack frame to cause a kernel stack overflow.

An interesting observation is that most (91% in Linux 4.15) invocations of this function set destination `dst` to address of a variable on kernel stack and thus will copy user data

[2]The security of `copy_from_user` has been improved by adding extra checks during the development of Linux Kernel. For example, upon failure during copy, set the `dst` memory region after the successfully copied bytes to zero to prevent uninitialized use.

into a kernel stack. If control-flow is hijacked to a invoking site of this function (e.g. line 2 in Table 5.5), an attacker could abuse such coding style to satisfy the requirement ❶ above because the code snippet help set `rdi` to current stack frame. However requirements ❷ and ❸ are still waiting to be satisfied given the initial `CFHP` does not implies any control over register `rdi` and register `rsi`. We will address this issue with blooming gadget in Section 5.6.3.

5.6.1.2 Choosing Short Return Path

The prevalent error handling code which is introduced to make kernel code more robust also provides a short return path for an attacker. An attacker could benefit from such a short return path after overflowing current stack frame.

As is depicted in line 4-5 in Table 5.4, function `bsg_ioctl` will directly return if return value of `copy_from_user` is *not* zero. Our statistic indicates the function `copy_from_user()` has been invoked at 671 sites. Among all these invocation sites, more than 99% contain the fault handling implementation

The insight of prioritizing short return path after stack smash is to prevent unexpected kernel panic as well as avoid the complexity of resolving extra data dependency in an error-prone and long normal return path of the function containing tens of basic blocks.

To take a short return path, `copy_from_user` must return a non-zero value as is shown in Table 5.4. Reviewing the source code of kernel function `copy_from_user`, we have identified 3 different situations which will force the function to return a non-zero value, 1) incurring an integer overflow when calculating *src+length*, 2) neither `src+length` nor `src` residing in user-space, 3) encountering an unresolvable page fault during copy. For the first two situations, the function `copy_from_user()` performs sanity check and returns a non-zero value without actually copying data to the kernel. For the last situation, the function migrates data to the kernel and pads with zeros the bytes failing to be copied.

5.6.1.3 Triggering Page Fault during `copy_from_user`

To force `copy_from_user` returning a non-zero value as well as successfully copying the `ROP` payload from user-space to kernel stack, we trigger page fault after copying enough payload according to the last condition described above.

We illustrate a representative example in Figure 5.7. We map two adjacent pages ($p1$ and $p2$) in the user-space and then unmap the second one. We fill the end of the page $p1$ with the actual payload including a stack canary and a ROP chain. We will

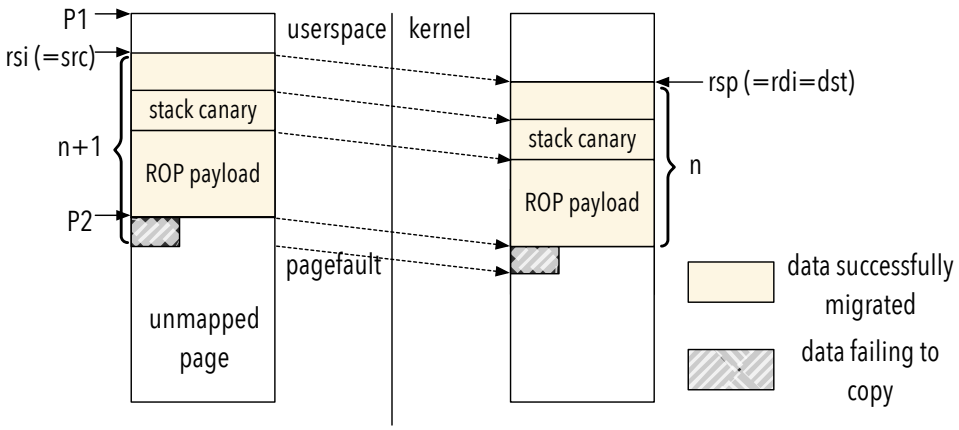


Figure 5.7. An example where `copy_from_user` triggers a page fault when copying user data to kernel stack.

discuss how to leak stack canary in Section 5.6.2. Through the technical approaches mentioned in Section 5.6.3, assume we have already obtained the control over registers `rsi` and `rdx` pertaining to the second and third parameters of `copy_from_user` respectively. Leveraging the control over registers, we manipulate the values in these registers. More specifically, we set `rsi` and `rdx` to $p1 + \text{PAGE\_SIZE} - n$ and $n + 1$ respectively, with n representing the length of payload actual copied. When the function attempts to copy the last byte, it failes to access the content at $p2$ and triggers a page fault because the page $p2$ is not mapped into the memory. Eventually `copy_from_user` returns a positive number 1 because one byte is not successfully copied.

5.6.2 Bypassing Stack Canary

As is mentioned earlier, to prepare a working payload for stack smash, an attacker has to know the value of kernel stack canary which remains secret in our threat model. We consider the presence of a strong kernel stack canary setting where stack canary is enabled for all functions containing a local variable.

We will first introduce two kinds of prevalent gadgets, then we discuss how to pair them to dump kernel stack memory.

5.6.2.1 Exposing Stack-disclosure Gadget and Auxiliary Function

To leak stack canary, an intuitive way is to construct an info leak of its value to user-space through an official data channel such that `SMAP` is not violated. In the following we introduce stack-disclosure gadget which is twin gadget of stack-smash gadget as well as auxiliary function prologue gadget.

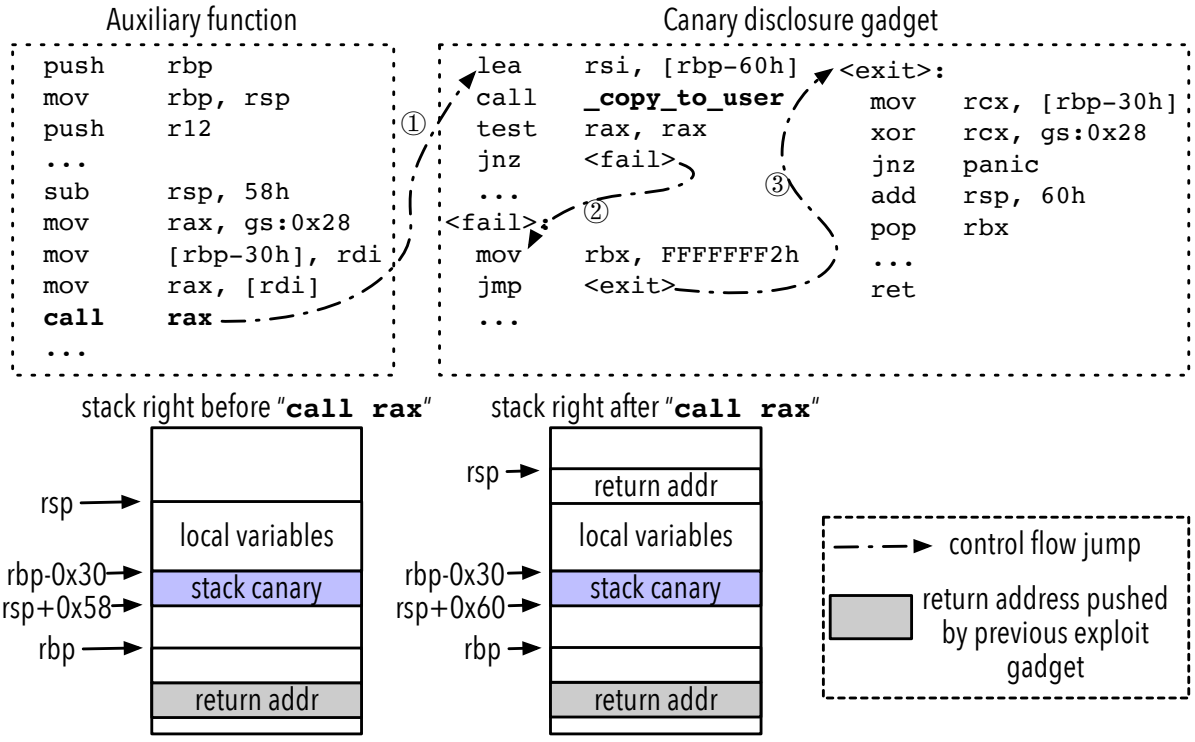


Figure 5.8. An example of canary disclosure gadget and its corresponding auxiliary gadget.

Stack-disclosure gadget. Function `copy_to_user()` is widely used in the Linux kernel codebase to copy kernel memory into user-space. In Linux kernel 4.15, our statistic indicates this function has been invoked at 594 sites. Of all these invocations, 82% are used for copying data from kernel stack to user-space. Since this naturally establish a channel to migrate data from kernel stack to user-space, we could exploit the characteristic of this kernel function to disclose the canary on kernel stack.

Auxiliary function. To successfully return from a stack-disclosure gadget and continue exploitation, we use auxiliary function to create a similar stack frame as the stack-disclosure gadget and transfer the control-flow to stack-disclosure gadget with an indirect call after the function prologue.

Auxiliary function should have stack canary protection and contain a controllable indirect call after its own function prologue which establishes a stack frame. Its layout of stack frame could be paired with a stack-disclosure gadget to form a "complete" stack frame and pass the stack canary check by putting a valid stack canary on the stack.

5.6.2.2 Disclosing Canary on Kernel Stack

By diverting the control-flow to a call site of `copy_to_user()`, we are closer to successfully disclose stack contents by satisfying the following four requirements. ❶, the registers should be set properly as parameters for `copy_to_user`, ❷, the kernel should not panic during the path caller function returns, ❸, the caller function of `copy_to_user` checks stack canary before return, ❹, the return address on stack must be set properly to continue the rest of the exploitation.

To tackle the first requirement, we leverage *blooming gadget* described in Section 5.6.3. For the second requirement, we could *trigger a page fault* and take a *short return path* similar to the technique described in Section 5.6.1.2. For the last two requirements, we pair stack-disclosure gadget with auxiliary function to generate a valid stack frame.

The key insight behind using auxiliary function to pair with stack-disclosure gadget is reusing the canary generated by the prolugue of auxiliary function. A pair of them should have the same number of saved registers, the same canary location and stack size of 8 bytes difference.

To elucidate the rationale behind the pairing, we take for example the routine of canary disclosure in Figure 5.8. We re-direct a `CFHP` to auxiliary function, After the prologue of auxiliary function which saves registers and establishes a stack frame, the target of indirect call `call rax` is set to the stack disclosure gadget in ①. Then content of current stack frame is copied to user-space by `copy_to_user`, a page fault is triggered to force non-zero return value of `copy_to_user`, as result *short return path* is taken in ②. Before the function returns, stack canary sanity check is performed ③, because the auxiliary function put a valid stack canary in current stack frame, the canary check is successfully passed and return to the caller of auxiliary function.

5.6.3 Putting them together: "One-shot" Exploit

It remains challenging to use an ill-suited `CFHP` to first disclose stack canary and then smashing kernel stack. The reason behind is a `CFHP` in practice may have limitations in the following two aspects. First, difficulty in combining aforementioned two building blocks with a single `CFHP`, second, lack of register control. "One-shot" exploitation uses a *blooming gadget* to amplify control over other registers and a *bridging gadget* to combine the two actions sequentially.

5.6.3.1 Augmenting `CFHP` with Blooming Gadget

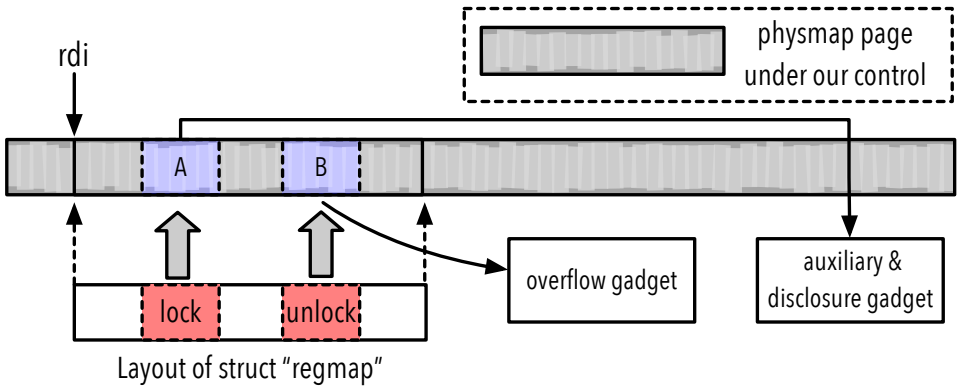


Figure 5.9. Memory layout after using `physmap` spray [1] to allocate physmap pages with data under our control.

To enhance a `CFHP` with the ability to control more registers, we introduce a family of *blooming gadgets*. The use of blooming gadget is inspired by `COOP` [155] which exploits a series of type confusions C++ program. Although Linux is written in C, its code heavily exhibits feature of object-oriented programming. The "self" object is usually passed as the first argument of function through `rdi`. And oftentimes the function containing indirect call using function pointer that resides in the object passed as parameter. We could let the `CFHP` to land at these functions to abuse type confusion.

We illustrate one such blooming gadget in Table 5.7. Kernel function `aliasing_gtt_unbind_vma` ↪ `()` contains an indirect call with three parameters calculated by dereferencing the function's first parameter `*vma`. Assume we have a `CFHP` with control over `rdi`, we can get an augmented `CFHP` which controls `rdi`, `rsi`, and `rdx` at the same time at line 3 in Table 5.7.

Note a blooming gadget works only if `rdi` is controllable at beginning. We found this requirement is easy to fullfil in practice. Our insight is that a `CFHP` usually has one register potentially controllable - either the register is fully controllable or the register points to a heap area under control. Such primitive can be turned into a `CFHP` with `rdi` control easily through a single gadget which ends with an indirect call. A worst case happens where a `CFHP` implies none of potentially controllable registers. Fortunately, we are still able to leverage uninitialized or controllable data on kernel stack as well as a common `ROP` gadget. For example, we could use `add rsp, 0x68; pop rdi; ret` to gain control over `rdi` if `rsp+0x68` and `rsp+0x70` is under our control.

5.6.3.2 Spawning Multiple `CFHP`s with Bridging Gadget

As is demonstrated in the motivating example in Section 5.4, an exploitation practice

depending on re-triggerable CFHP is not reliable because of exploit path pitfalls. We use bridging gadget - a family of kernel functions with multiple controllable indirect calls - to spawns two CFHPs and combine canary leak and stack smash into a single shot.

For example, function regcache_mark_dirty shown in Table 5.8 is such a bridging gadget which contains two indirect calls, map->lock in line 2 and map->unlock in line 5. As we can observe from the kernel gadget shown in Table 5.8, the function pointers tied to these two indirect calls are enclosed in a data object referred by the first argument of the function regcache_mark_dirty(). Recall that the first argument of a function is specified by the general register rdi, and we can usually obtain the control over that register using technique described in Section 5.6.3.1. As a result, in order to obtain control over both function pointers, we could first employ ret2dir [1] to allocate physmap pages and carefully crafted a data object accordingly. Then, we could refer the register rdi to a proper spot and set rip to the entry site of the gadget shown in Table 5.8. As is shown in Figure 5.9, assume the data carefully crafted in the spots of A and B represent the address of the auxiliary gadget together with a disclosure gadget responsible for leaking stack canary as well as the entry address of the gadget pertaining to stack smashing respectively. Then, by executing the bridging gadget shown in Table 5.8, we could first leak canary using the first indirect call. After the return of the call to copy_to_user(), there is no operations between the consecutive indirect calls that impose additional constraint to the second function pointer. Therefore, we could perform the stack smashing using the second function pointer without involving unexpected termination.

5.7 Implementation

Using IDA Pro SDK [171] and angr [83], we implemented KEPLER with about 8,000 lines of Python code. KEPLER is an automated tool that tracks down the aforementioned exploitation gadgets and chains them for exploitability assessment. Figure 5.10 depicts how these gadgets are concatenated. While previous sections have discussed the basic building blocks to perform an "one-shot" exploitation, the exploit chain could not be determined once and for all with static analysis because uniqueness of each CFHP and different gadget combinations bring about the variation of the exploitation context. For example, the consecutive exploitation gadgets might no longer obtain the control over related registers with a different initial CFHP.

To address this problem, we developed our tool to assess each of the gadget chains potentially useful for kernel exploitation. More specifically, we follow the guidance of the

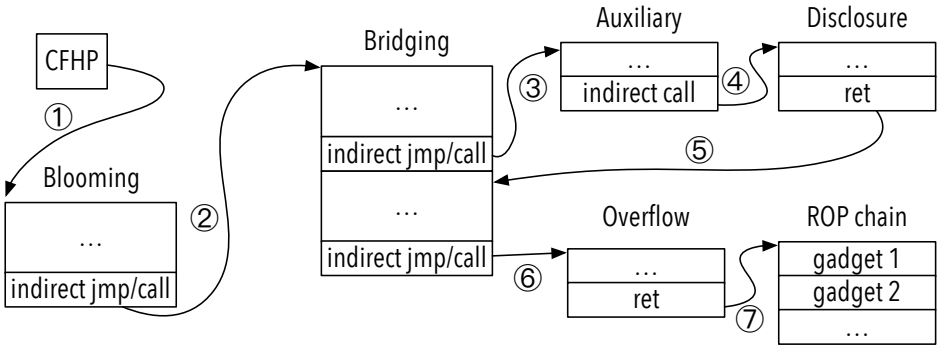


Figure 5.10. An illustration of how KEPLER stitches various kernel gadgets for ultimate exploitation.

exploitation chain construction shown in Figure 5.10, and design our tool to perform a depth-first exhaustive search which explores all the possible combinations of exploitation gadgets. When performing the depth-first search: starting from the `rip` hijack site, KEPLER symbolically executes the gadget chain that the search algorithm explores. To determine whether a gadget chain is useful for exploitation, our tool checks the memory access and deems a gadget chain useless if that exploitation chain attempts to access the user space or an unmapped kernel memory region. In addition, KEPLER examines the control over the registers at two critical sites – one at the entry of the disclosure gadget and the other at the entry of the overflow gadget. We implemented KEPLER to deem a gadget chain useless and terminate symbolic execution earlier if it has no control over the registers `rdi` and `rdx` at the first checking site or has no control over `rsi` and `rdx` at the second checking site. The reason behind this implementation is that, after executing bridging and auxiliary gadgets, we might lose the control over the registers needed for disclosure and overflow gadgets. With the check right before symbolically executing the two gadgets, we can quickly determine the usefulness of the gadget chain in exploitation, terminate unnecessary symbolic execution and thus save the computation resources.

In the process of the assessment of the exploitation chain, KEPLER symbolically executes each exploitation gadget. For some of them, they might carry a large number of basic blocks and even infinite loops. This could significantly influence the efficiency our tool and even incur the state explosion problem. To avoid these issues, for each path in an exploitation gadget, we set KEPLER to explore at most 20 basic blocks. In addition, we developed KEPLER to concretize each symbolic address. To be more specific, we set up a kernel page under our control in the physmap region and then concretize each symbolic address with a non-overlapping address of that memory page. In this work, we implemented this concretization mechanism by simply extending `ControlledData` – one

of the symbolic address concretization strategies of `angr`.

For each gadget chain that passes the assessment, KEPLER further performs constraint solving to generate payload accordingly. Technically, this can be easily done by using `angr`. However, the `Z3` solver used in `angr` consumes memory exhaustively and generally does not release the memory used for constraint solving even after the completion of computation. To address this problem, KEPLER partitions symbolic execution and constraint solving into two different processes. In this way, KEPLER could terminate the memory-intensive process every time the constraint solving is completed and thus free the memory for consecutive computation.

As is described in Section 5.6.1, the payload smashed to the stack contains the stack canary disclosed as well as a sequence of addresses indicating an ROP chain that performs actual exploitation. With respect to the stack canary, we employ a separated thread in the user-space to rapidly retrieve the canary – whenever it is disclosed to the user-space – and then make it ready for stack smashing. Regarding the ROP chain used in this work, we simply choose the ROP payload commonly used for privilege escalation. In Appendix, we specify the ROP payload used in this work. It invokes kernel functions `commit_creds()` and `prepare_kernel_cred()` to obtain the root privilege. Note that the construction of an ROP payload is out of scope of this paper. There are many commonly-adopted ROP payloads, which can be naturally hooked with our new kernel exploitation technique.

5.8 Case Study and Evaluation

In this section, we demonstrate our new exploit technique and evaluate our automated tool KEPLER using real-world kernel vulnerabilities and some recently-released CTF challenges. To be specific, we compare KEPLER with various kernel exploitation techniques to show it is a generic exploit technique, we also compare KEPLER with automatic exploit generation systems to highlight its power in evaluating exploitability with a `CFHP`. In addition, we show the efficacy and efficiency of KEPLER in facilitating exploitation chain construction.

5.8.1 Setup

We first randomly selected 3 recently released CTF challenges as well as 16 real-world kernel vulnerabilities archived between 2016 and 2017. Then, we successfully assembled these vulnerabilities in a mainstream Linux kernel 4.15.0 by inserting them into the kernel code or reverting their patch accordingly. In this work, we evaluate our tool KEPLER by

using this single Linux kernel, and demonstrate the effectiveness of "one-shot" exploit by launching exploitations against the inserted vulnerabilities.

As is summarized in Table 5.9, the vulnerabilities inserted cover various types such as Use-After-Free and Out-Of-Bound (OOB) read/write etc. It should be noted that the CVEs selected are a little bit unbalanced – with more in 2017 and less in 2016. On the one hand, this is because there are more than 2× of kernel vulnerabilities reported in 2017 than those in 2016 [172]. On the other hand, this is because some components in Linux kernel experience significant overhaul since 2016 and we have difficulty of re-enabling the corresponding vulnerabilities in a new kernel image.

In order to run and evaluate KEPLER, we also assembled and configured a testbed which has a 32-core Intel(R) Xeon(R) Platinum 8124M CPU and 256GB of memory. For each vulnerability, we then used this testbed to run 28 concurrent workers which simbolically explore the kernel code space and track down useful exploitation chains in parallel.

5.8.2 Effectiveness of "One-shot" exploitation

By searching the Internet, we gathered 10 exploits pertaining to the vulnerabilities inserted. As is shown in Table 5.9, these publicly available exploits perform exploitation through various approaches and therefore demonstrate different capability in bypassing kernel mitigations. Among these exploits, we found there are only 5 of them demonstrating the ability to perform exploitation under our aforementioned threat model. In comparison with the working exploits generated by KEPLER, publicly available exploits demonstrate much weaker exploitability (with 5 vs 17 cases). To some extent, this implies existing exploitation approaches highly rely upon the quality of the target vulnerability and corresponding `CFHP`, whereas our approach KEPLER could utilize prevalent kernel function and gadgets to explore exploitable machine states and thus escalate the exploitability for a `CFHP`. However, previous exploit technique `Q` [148] could not generate working exploit because `Q` rely on a stack pivoting gadget while its gadget discovery phase return none of working pivoting gadget. `FUZE` could only generate exploit for 3 cases because it evaluate exploitability of a `CFHP` simply with two straight forward exploit technique: `pivot-2-usr` and "`cr4`-flipping". The former does not bypass `SMAP` and the latter only works when at least two `CHFP`s is available.

Even for the vulnerabilities against which both public exploits and ours demonstrate the same capabilities in bypassing mitigations, we argue that our approach still exhibits stronger exploitability. This is because the public exploits circumvent mitigations by

manipulating control registers with two CFHPs, as is discussed in Section 5.2.3, this practice can be easily restricted by virtualization extension. For the two vulnerabilities CVE-2017-17053 and CVE-2016-9793 for which our approach fails to derive working exploits, we manually examine their execution traces leading to the kernel panic. We find that the failure results from the following fact. In order to take the control over rip prior to exploitation, both of these vulnerabilities require an exploit to access the data in the user space. This violates the protection of SMAP. KEPLER restricts any operations that violate our threat model and output a failure if none of the exploitation chains could avoid such violation.

5.8.3 Effectiveness and Efficiency of Our Tool

Our experiment utilizes KEPLER to explore the aforementioned kernel image with the vulnerabilities inserted. In this process, we exhaustively search gadget chains useful for exploitation and mitigation circumvention. In Table 5.9, we show the total number of useful exploitation chains identified as well as the total amount of time spent on finding these gadget chains. As we can observe, KEPLER could automatically pinpoint tens of thousands of unique kernel gadget chains to perform exploitation without triggering kernel protections. Since we implement KEPLER to perform gadget chain exploration in parallel, we also discover that these gadget chains could typically be identified within 50 hours. These observations together imply that KEPLER could diversify the ways of performing kernel exploitation in an efficient fashion. Given that some commercial security products pinpoint kernel exploitation by using the patterns of exploits, the ability to diversify exploitation has the potential to assist an adversary to bypass the detection of commercial security products.

From Table 5.9, we also observe that, for different vulnerabilities, KEPLER generates different number of gadget chains useful for exploitation. This can be attributed to the following fact. In the process of gadget chain identification and assessment, KEPLER starts gadget assessment from different machine states and contexts. For some vulnerabilities, the machine states and contexts do not provide us with sufficient control over some registers and memory regions. Under this circumstance, the availability of useful kernel gadgets would vary and thus influence the total number of generated exploits.

In Table 5.9, we also depict the time spent on finding the first kernel gadget chain useful for exploitation. As we can observe, KEPLER could quickly output an useful exploitation chain in less than about 50 minutes. This implies KEPLER has the potential to be used as a tool to quickly derive a working exploit without too many human efforts.

Last but not least, Table 5.9 also shows the total number kernel gadgets in different categories. As we can observe, there are typically tens of gadgets in each categories. This means that one cannot simply block our exploitation approach by eliminating a small number of kernel gadgets. In Section 5.9, we will further discuss the defense of our exploitation approach.

5.9 Discussion and Future Research

In this section, we discuss some plausible defence mechanisms against our "one-shot" exploit chain. Also, we elaborate why they are not effective nor suitable for preventing the proposed attack. Following our discussion and analysis, we then provide some suggestions for the future research.

Plausible Defense Mechanisms. To defend against the exploit chain mentioned above, one straightforward reaction is to eliminate the gadgets that must be used in kernel exploitation. However, as we have already demonstrated and discussed in Section 5.8, the tool we develop could enrich the choices of the gadgets needed for exploitation. This means that, following this potential solution, Linux developers would inevitably introduce significant amount of kernel code changes and it is difficult to guarantee these changes would not bring about negative influence upon Linux kernel execution. Other security mitigation could also be used as potential defense mechanisms. For example, there have already been a rich collection of research works on control and data flow integrity protection (e.g., [142–145, 161, 173]). Integrating and enabling them in Linux kernel, they could easily fail the attack mentioned above. Unfortunately, these techniques usually incur unacceptable overhead (e.g., [143] has an average overhead of 13%) or sometimes rely upon hardware features to reduce their overhead (e.g., [173]). As a result, they are barely used as a practical, general defense solution in release version.

Possible Future Research. Looking ahead, we suggest the future research could be conducted from two aspects. From the perspective of automatic exploit primitive evaluation, we believe there is an emerging need to invent technique to systematically evaluate various exploit primitives, expecially for those weak exploit primitives. In practice, theory and techniques should be proposed to facilitate deriving better exploit primitive with a initially weak exploit primitive. From the perspective of defense, on one hand, we believe there remains the need to design lightweight control-flow enforcements for Linux kernel. On the other hand, instead of manually overhauling kernel code, one could augment `GCC` with the ability to eliminate the exploitation gadgets at compilation

time.

5.10 Conclusion

We show it is generally challenging to generate exploits with a control-flow hijacking primitive in the Linux kernel under a realistic threat model, while there are a lot of research efforts in identifying exploit primitives and facilitating exploit generation with various exploit primitives. We propose KEPLER, a framework to facilitate evaluation of control-flow hijacking primitives which leverages a novel "one-shot" exploitation to convert a control-flow hijacking primitive into a classic stack overflow and thus bootstrap traditional code-reuse attack against modern Linux kernel. In comparison with previous automatic exploit generation and exploit hardening techniques, we showed that KEPLER outperforms other exploit techniques and could automatically generate thousands of working exploit for a control-flow hijacking primitive in Linux kernel despite the challenges of widely-deployed security mitigations, exploit path pitfall and ill-suited exploit primitive. Following this finding, we safely conclude that KEPLER can significantly facilitate evaluating control-flow hijacking primitive in the Linux kernel.

type=table

```
struct ip_mc_socklist {
  struct ip_mc_socklist *next_rcu;
  struct ip_mreqn multi;
  unsigned int sfmode;
  struct ip_sf_socklist *sflist;
  struct rcu_head rcu;
};
```

Table 5.1. Definition of struct `ip_mc_socklist`. Its first member `next_rcu` is unmanageable because the PoC uses a heap spray technique which does not allow us to control first `QWORD` of struct `ip_mc_socklist`.

```
void ip_mc_drop_socket(struct sock *sk){
  struct inet_sock *inet = inet_sk(sk);
  struct ip_mc_socklist *iml;
  // inet->mc_list is a dangling pointer
  while ((iml = inet->mc_list) != NULL) {
    // iml is alias of the dangling pointer
    inet->mc_list = iml->next_rcu;
    // queuing a rcu_head for execution in the future
    kfree_rcu(iml, rcu);
  }
}

void rcu_reclaim(struct rcu_head *head){
  head->func(head); // control-flow hijack
}

void rcu_do_batch(...){
  struct rcu_head *next, *list;
  while (list) {
    next = list->next; // next rcu is unmanageable
    rcu_reclaim(list);
    list = next;
  }
}
```

Table 5.2. Tailored source code pertaining to the `CFHP`. function `ip_mc_drop_socket` repeat invoking `kfree_rcu()` which queues a `rcu` task for asynchronous execution until `inet->mc_list` is `NULL`. The site pertaining to the `CFHP` is in function `rcu_reclaim`.

Table 5.3. A control-flow hijacking primitive in kernel UAF vulnerability CVE-2017-8890.

type=table

```
static long bsg_ioctl(struct file *file, unsigned int cmd, unsigned long
    ↪ arg){
  struct sg_io_v4 hdr;
  ...
  if (copy_from_user(&hdr, uarg, sizeof(hdr)))
    return -EFAULT; // short return
  ...
}
```

Table 5.4. Source code.

```
...
  mov rdi,rsp
  call <_copy_from_user>
  test rax,rax
  je 0xffffffff813d6ce4
  mov rax,0xfffffffffffffff2
  jmp <epilogue>
...
<epilogue>:
  mov rcx,QWORD PTR [rsp+0xa0]
  xor rcx,QWORD PTR gs:0x28
  jne <__stack_chk_fail>
  add rsp,0xa8
  pop rbx
  ret
```

Table 5.5. Assembly code.

Table 5.6. The kernel code fragment of an stack-smashing gadget that could smash kernel stack with carefully crafted payload.

type=table

```
static void aliasing_gtt_unbind_vma(struct i915_vma *vma) {
    ...
    vma->vm->clear_range(vma->vm, vma->node.start, vma->size);
    ...
```

Table 5.7. The kernel code fragment (a blooming gadget) that could enhance the control over multiple general registers.

type=table

```
void regcache_mark_dirty(struct regmap *map) {
    map->lock(map->lock_arg); // the 1st control-flow hijack
    map->cache_dirty = true;
    map->no_sync_defaults = true;
    map->unlock(map->lock_arg); // the 2nd control-flow hijack
}
```

Table 5.8. The source code of a bridging gadget – the kernel code fragment that could spawn multiple CFHPs.

ID	Vulnerability type	Public exploit	Q	FUZE	KEPLER	G1	G2	G3	G4	First chain (min)	Total time (hour)	Total # of exploitation chains
CVE-2017-16995	OOB readwrite	✓†	✗	✗	✓	41	114	27	201	45	37	29788
CVE-2017-15649	use-after-free	✓	✗	✓	✓	29	79	25	280	16	28	60207
CVE-2017-10661	use-after-free	✗	✗	✗	✓	28	78	30	301	17	25	49070
CVE-2017-8890	use-after-free	✗	✗	✗	✓	21	88	23	304	17	18	50471
CVE-2017-8824	use-after-free	✓	✗	✓	✓	63	101	35	306	50	70	164898
CVE-2017-7308	heap overflow	✓	✗	✗	✓	31	91	30	241	14	47	110176
CVE-2017-7184	heap overflow	✓	✗	✗	✓	31	95	31	254	24	37	93752
CVE-2017-6074	double-free	✓	✗	✗	✓	18	79	31	308	16	15	31436
CVE-2017-5123	OOB write	✓†	✗	✗	✓	40	86	27	311	14	39	113466
CVE-2017-2636	double-free	✗	✗	✗	✓	18	89	29	289	29	19	26372
CVE-2016-10150	use-after-free	✗	✗	✗	✓	34	84	25	293	52	34	88499
CVE-2016-8655	use-after-free	✓†	✗	✓†	✓	18	109	32	260	15	17	47413
CVE-2016-6187	heap overflow	✗	✗	✗	✓	22	85	32	301	17	21	51954
CVE-2016-4557	use-after-free	✗	✗	✗	✓	21	80	21	295	16	37	40889
CVE-2017-17053	use-after-free	✗	✗	✗	✗	-	-	-	-	-	-	-
CVE-2016-9793	integer overflow	✗	✗	✗	✗	-	-	-	-	-	-	-
TCTF-credjar	use-after-free	✓†	✗	✗	✓	35	89	25	292	25	14	82913
0CTF-knote	uninitialized use	✗	✗	✗	✓	21	89	33	318	17	36	40923
CSAW-stringIPC	OOB read&write	✓†	✗	✗	✓	35	88	25	289	17	33	84414

Table 5.9. The comparison of exploitability as well as performance of KEPLER. G1, G2, G3 and G4 represent the blooming gadget, bridging gadget, auxiliary and disclosure gadget pair, and stack-smash gadget. The "first chain" column indicates the time spent on pinpointing the first exploitation chain. The "total time" column specifies the total amount of time spent on finding all useful exploitation chains. † symbol represents the cases where the exploits could only bypass major mitigations (e.g. SMAP and SMEP) and fail to bypass others under our threat model. ✓and ✗ symbols indicate the existence and non-existence of a working exploit.

Chapter 6
HotBPF: Eliminate Exploitable Design pattern via Advanced Protection Design

In previous chapters, we developed tools to investigate exploitable design patterns and demonstrate the limitations of existing protection mechanisms. More specifically, we developed `FUZE` to explore the capability of vulnerabilities which facilitates exploitation, `ELOISE` to identify elastic structure as a design pattern and demonstrate that its exploitability given different vulnerability capabilities, `SLAKE` to facilitate exploitability evaluation by manipulating memory layout and obtaining exploitable primitives. Finally, we proposed `KEPLER` to evaluate existing mitigations using the obtained primitives. The results, unfortunately, show that existing protections fail to prevent exploitation in front of various potential exploitable design patterns.

Although in Chapter 3, we designed an isolation-based mitigation to protect the integrity of elastic structures, it is not general enough to be effective for other exploitable design patterns. In this chapter, we introduce a new defense named after `HotBPF` to isolate corruption and thus protect the Linux kernel from attacks that start from memory corruption and take advantage of a variety of exploitable design patterns.

6.1 Introduction

The Operating System kernel is the infrastructure of cyberspace. Despite extensive code auditing, the Linux kernel, like all other OS kernels, inevitably contains tremendous amount of vulnerabilities. The memory corruption introduced by these vulnerabilities can be misused by attackers to launch exploitation and achieve malicious goals such as

privilege escalation and information leakage. The attackers typically utilize the corruption to overwrite or overread sensitive data in the Linux kernel and thus violate security polices. For example, as demonstrated in `ELOISE`, the attackers can use the corruption to tamper the length field in elastic structures, misleading the kernel to overread flexible buffer.

To protect these sensitive data, researchers have developed a large number of isolation-based defenses. These defenses aim to place the sensitive data and objects to a dedicated domain so that memory corruption cannot overlap with them. In xMP [40], the authors leverage hypervisor visualization to isolate selected kernel objects. In AutoSlab [174], Grsecurity developers create a dedicated cache for every allocation site in the kernel. In kalloc_type used in Apple's product, the engineers rely on human expertise to select sensitive objects and allocate them to special zones. These isolation mechanisms raise the bar of exploitation to some extend but still suffer from the following shortcomings. First, they are only able to provide proactive protection - designed and implemented after sensitive objects are pervasively corrupted in the wild. Second, to enable these protections, the system administrators need to recompile and reboot the system which inevitably disrupts critical services runnng over the machine. Third, most defenses cannot prevent cross-cache exploitation which has been widely used [174]. Finally, the implementation of these isolation mechanism either introduce non-negligible overhead or rely on special hardware features or hypervisor to avoid performance degradation, which restricts their application in scenarios such as embedded systems and desktops.

In this work, we propose a new defense that isolates memory corruption instead of sensitive data and object. Since memory corruption is separated, the attackers can hardly create memory overlapping and thus launch exploitation. Technically speaking, this defense consists of two components. The first component takes as input a vulnerability report and statically analyze it to identify the vulnerable object (i.e., objects where memory corruption happens). Then, the first component further calculates all kernel program sites that can allocate the vulnerable object and feed them to the second component. The second component leverages eBPF mechanism to intercept all allocation sites of interest on-the-fly and divert the allocation of vulnerable object to `vmalloc` region. By separating memory corruption from other kernel objects, this defense can prevent attackers from exploiting the reported vulnerability. We name this defense as `HotBPF` because it plays a role similar to hot patch by remediating vulnerabilities without stopping the machine.

Compared with previous isolation-based defenses, `HotBPF` has the following advantages.

First, it can provide protection immediately after a severe vulnerability is reported, bridging the time window between vulnerability disclosure and vulnerability remediation. Second, this protection can be enabled on-the-fly which means there is no need to disrupt critical services to recompile and reboot the system. Third, compared with existing mitigations in the Linux kernel, this defense can provide stronger security protection by preventing cross-cache exploitation. Fourth, this defense is independent of hardware features and hypervisor virtualization. So it can be widely deployed in a variety of scenarios (e.g., embedded systems, desktops, and cloud servers). Last but not least, this defense is lightweight as it introduces an overall 2% - 3% performance overhead.

In summary, this work makes the following contributions.

- We design a new defense `HotBPF` that can separate memory corruption immediately after the vulnerability report. This separation can be enabled without recompiling and rebooting the system.
- We integrate `HotBPF` to the Linux kernel and demonstrate its outstanding effectiveness against diverse vulnerability types and exploitation techniques using real-world attacks.
- By measuring the performance overhead of `HotBPF` using benchmarks of different granularity levels, we highlight the lightweight feature of `HotBPF`. We expect to see that `HotBPF` can be merged to the upstream kernel in the future.

6.2 Background

In this section, we first describe memory management in the Linux kernel, followed by summary of kernel heap-based exploitation techniques. Then, we introduce the eBPF mechanism used to implement `HotBPF`.

6.2.1 Kernel Memory Management

Generally speaking, the Linux kernel replies on three allocators for memory management: buddy allocator, SLUB/SLAB allocator, and vmalloc allocator.

Buddy allocator. As the interface to physical memory, Buddy allocator implements page allocation by dviding the physical memory into chunks the size of which is an order of a page. At the moment of allocation, if the allocation size (e.g., 1 page) is not available, a larger chunk (e.g., 4 pages) is halved continuously until a chunk of the exact size is

produced (e.g., 4 pages to 2 pages to 1 page). The two halves are called buddies. One buddy is used to satisfy the allocation request and the other buddy will remain free. Later at the moment of free, the two buddies, if both are free, will coalesce into a larger chunk of free memory.

SLUB/SLAB allocator. Built on top of buddy allocator, SLUB/SLAB allocator requests memory from buddy allocator and takes care of memory management within pages. More specifically, SLUB/SLAB allcoator further partitions continuous pages into multiple individual 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 head- ers 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).

vmalloc allocator. Also built on top of buddy allocator, vmalloc allocator is different from SLUB/SLAB allocator. SLUB/SLAB allocator returns memory that is physically contiguous while vmalloc allocator returns memory that is virtually contiguous. Thus, the address translation of by SLUB/SLAB allocator memory can be done through direct calculation while the vmalloc memory relys on page tables to record the mapping relationship between virtual address and physical address. Although both rely on buddy allocator, the memory regions managed by the two allocators are separated and never overlap with each other. This feature makes it possible for `HotBPF` to isolate memory corruption and prevent it from tampering sensitive data and object in kernel. We will elaborate on this in Section 6.4.2.

6.2.2 Nature of Kernel Heap-based Exploitation

In Section 4.2, we presented commonly heap-based exploitation techniques. In other chapters, the readers can also learn the typical procedure of exploiting heap corruption vulnerabilities. Here, we summarize the key idea behind these exploitation approaches to illustrate the design of `HotBPF`.

The two elements of heap-based exploitation are “overlaping” and “sensitve data”.

More specifically, the attackers need to overlap the corruption introduced by the vulnerability to overwrite or overread sensitive data. In use-after-free and double free, the corruption range is the freed object referenced by the dangling pointer. In heap out-of-bound write/read, the corruption range is where overflow happens. The attackers want to place sensitive data within the corruption range so that tampering and leaking can succeed. The best way to deal with the spatial overlap is strict isolation and the best way to deal with the temporal overlap is one-time allocation (OTA, i.e., never reclaim freed memory).

Most existing defense mechanisms focus on preventing spatial overlap by isolating them to a dedicated domain. None of them puts efforts on isolating corruption because of the following challenges. First, the corruption source varies vulnerability by vulnerability and there lacks a systematic and automatic approach to identify corruption source given a vulnerability. Second, unlike sensitive data which can be marked at compilation time, the corruption can happen to any kernel variables. Without prior knowledge of vulnerable object (i.e., where corruption happens), the isolation needs to be enforced dynamically. Third, the on-the-fly isolation inevitably modify the allocation, access, and free operations of vulnerable objects, it's challenging to reduce the impact of these modification and thus lower the performance overhead of isolation. Recently, there are a few works propose to eliminate temporal overlap (e.g., [175]). however, there is no OTA solution for OS kernel because straightforward OTA solution will quickly exhaust memory which is not acceptable for OS as an infrastructure. In this work, we will illustrate how we design `HotBPF` to overcome these challenges by extending eBPF mechanisms.

6.2.3 eBPF Mechanism

As an in-kernel interpreter, the Berkeley Packet Filter (BPF) [176] is originally designed to run packet filters from userspace using a reduced instruction set. On the basis of BPF, extended BPF (eBPF) introduces a new bytecode and just-in-time compilation for improved performance. Loaded from userspace, an eBPF program can be triggered by a specific kernel event.

Thanks to kernel data structure named BPF maps, eBPF programs can maintain and access persistent memory. Maps are designed to store arbitrary data structures the size of which must be specified by the user application at creation time. They can be used for communicating between different eBPF programs or between eBPF programs and user applications. Furthermore, the kernel provides a restricted set of helper functions which can be called by eBPF programs to interact with the system and access specific

kernel data (e.g.map data, time since boot up). To allow using the C language to write eBPF programs, the eBPF bytecode backend is supported by the Clang/LLVM compiler toolchain.

Because the code of eBPF is provided by userspace, running it inside the kernel can impact the system's security and stability. To check if the program can be safely attached and executed, the Linux kernel calls the in-kernel eBPF verifier every time it loads an eBPF program. The verifier guarantees that the program meets two properties: safety, i.e., the program neither accesses unauthorized memory, nor leaks kernel information, nor executes arbitrary code, and liveness, i.e., the execution of the program will always terminate.

This verification process must be guaranteed to terminate. Therefore, a complexity limit is enforced by the kernel. Whenever the number of verified instructions reaches this limit, an eBPF program is rejected. Since Linux 5.3, the verifier supports bounded loops in eBPF programs by analyzing the state of every iteration of a loop. The verifier must be able to check every loop iteration before hitting the previously-mentioned instruction complexity limit. This limits the number of loop iterations as well as the complexity of the instructions in the loop body.

Because of all these limitations, eBPF is currently mostly used to monitor a running kernel or to process low-layer protocols of network packets (i.e.L2-L4). In this work, we will demonstrate how to leverage eBPF to provide on-demand protection without violating these limitations.

6.3 Motivating Example & Design Overview

In this section, we first describe a real-world use-after-free vulnerability reported by Syzkaller to illustrate the concept of corruption and vulnerable object. Then, we present the design overview of `HotBPF` which can separate corruption and prevent attackers from misusing the corruption to tamper sensitive kernel data and object.

As is depicted in the list, the function `tun_attach` is responsible for configuring the network interface. Its argument `tun` refers to a global variable shared by all the tun files in the opened state. As is shown in line 3, if `IFF_NAPI` is set in `tun->flags`, the kernel will initialize a timer and link the corresponding `napi` to the list of the network device `napi_list`. In line 12, another function `tun_detach` is responsible for cleaning up the data enclosed in `tun_file` as well as closing the file. If `IFF_NAPI` is set, the kernel will cancel the timer and remove the `napi` from `napi_list` of the device. In line 23, the function

`free_netdev` will go through the `napi_list` to delete `napi` in the list.

```
1  static void tun_attach(struct tun_struct *tun, ...)
2  {
3      if (tun->flags & IFF_NAPI) {
4          // initialize a timer
5          hrtimer_init(&napi->timer, CLOCK_MONOTONIC,
6              HRTIMER_MODE_REL_PINNED);
7          // link current napi to the device's napi list
8          list_add(&napi->dev_list, &dev->napi_list);
9      }
10 }
11
12 static void tun_detach(struct tun_file *tfile, ...)
13 {
14     struct tun_struct *tun = rtnl_dereference(tfile->tun);
15     if (tun->flags & IFF_NAPI) {
16         hrtimer_cancel(&tfile->napi->timer);
17         // remove the current napi from the list
18         netif_napi_del(&tfile->napi);
19     }
20     destroy(tfile); // free napi
21 }
22
23 void free_netdev(struct net_device *dev) {
24     list_for_each_entry_safe(p, n,
25                 &dev->napi_list, dev_list)
26         netif_napi_del(p); // use-after-free
27 }
```

Listing 6.1. The code snippet of the Linux kernel with a bug which demonstrates a use-after-free error.

Before invoking the function `tun_attach`, we can set `tun->flags` as `IFF_NAPI`. In this way, after `tun_attach` is called, the kernel could add the corresponding `tun_file` to the device list `napi_list`. Following this setup, we can further invoke `ioctl` to clear `tun_flags` and then call `tun_detach`. As is shown in Listing 6.1, the function `tun_detach` does not remove the corresponding `napi` from the list in line 17 $\sim$ 18, but frees it in line 20. Therefore, when traversing the device list, the KASAN-instrumented kernel will throw the use-after-free error. Based on this use-after-free error, many analysts may consider the bug is probably exploitable. This is a real-world vulnerability reported by Syzkaller in [177]. As described in Section 4.2, to exploit a use-after-free vulnerability, the attackers

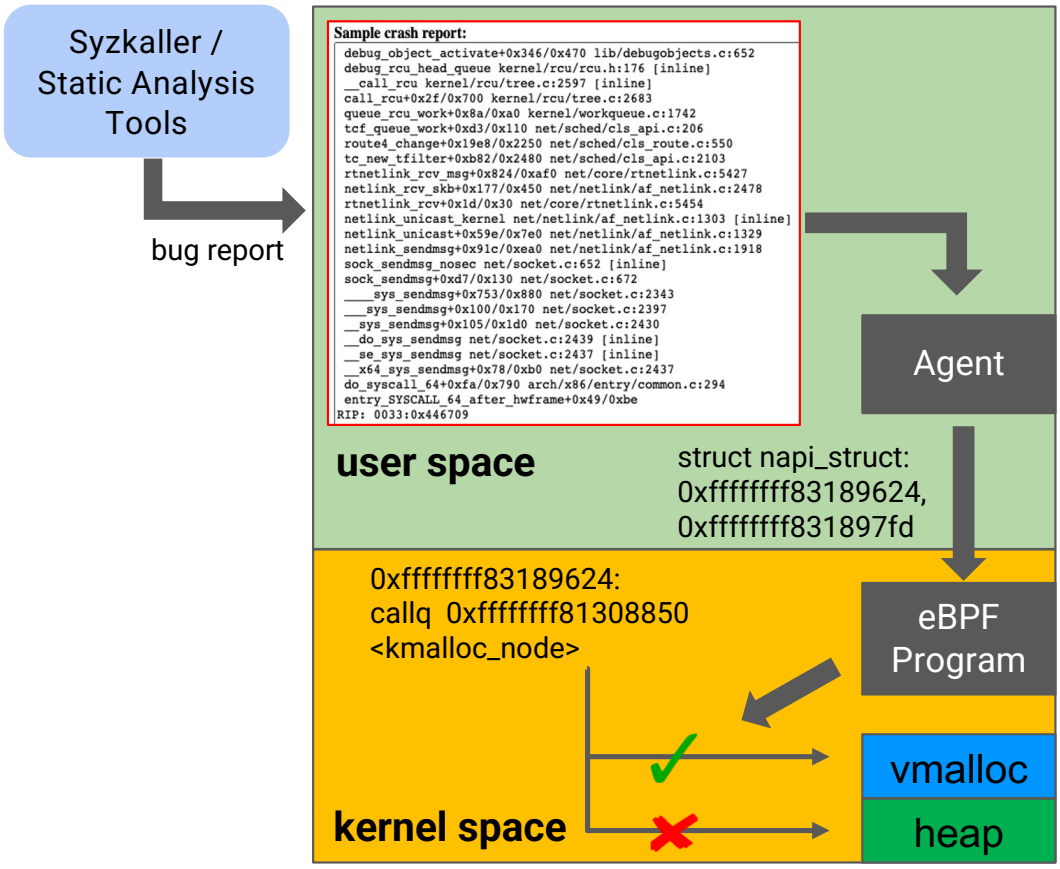


Figure 6.1. Overview of `HotBPF` which consists of two components. The "Agent" component in the userspace takes as input bug reports generated by Syzkaller or other static analysis tools and outputs the type of vulnerable structures (i.e., `struct napi_struct` as well as the addresses of its allocations sites). The "eBPF Program" intercepts the allocation sites of vulnerable structures and diverts it to the vmalloc region so that associated corruption is separated from sensitive kernel data and objects.

need to perform heap spray and overwrite sensitive data (e.g., function pointer) in the vulnerable object (i.e., `struct napi_struct` which is referenced by dangling pointer). To prevent the exploitation, `HotBPF` is designed to prohibit overlapping between vulnerable object and spray object.

As illustrated in Figure 6.1, the design of `HotBPF` consists of two major components. The first component is the agent in userspace. The userspace agent takes as input the vulnerability report generated by Syzkaller and static analysis tools as examplified in [177]. By analyzing the vulnerability report and kernel source code, the userspace agent identifies the type of vulnerable object (e.g., `struct napi_struct`). Further, the agent can pinpoint the allocation sites of vulnerable objects and calculate the binary address of these sites in the kernel image (e.g., `0xffffffff83189624`). These addresses will

be passed to eBPF program - the second component in the the kernel space through BPF maps. The eBPF program intercepts the allocation sites of loaded kernel image. When allocation happens, the eBPF program will be executed to divert the allocation to vmalloc region. As such, the vulnerable object (e.g., `struct napi_struct`) is allocated in vmalloc region and thus separated from other kernel objects. When attackers make attempt to perform heap spray, the spray object resides in the memory managed by SLUB/SLAB allocator and thus fails to overlap with the vulnerable object. In some situations, the vulnerable object and the spray object are of the same type. As a result, both vulnerable object and spray object will be allocated in vmalloc region which creates opportunity for overlapping. To prevent this, in the vmalloc region, we further implement One-time allocation (OTA) using BPF maps in eBPF program. More specifically, once the vulnerable object is freed, the memory region holding the vulnerable object will not reclaimed for future allocation. As such, the spray object will not take over the freed memory and thus overlap with the vulnerable object. More technical details can be found in Section 6.4.2.

6.4 Technical Details

In this section, we elaborate on the technical details of `HotBPF`. First, we describe how to analyze a kernel bug report and identify vulnerable structures (i.e., type of objects where corruption happens). Second, we discuss how to extend eBPF mechanism to intercept the allocation sites of vulnerable object, divert the allocation to `vmalloc` region, and realize an one-time-allocation.

6.4.1 Vulnerable Object Identification

In this work, we utilize backward taint analysis to identify vulnerable structures (i.e., type of vulnerable object where corruption happens). Here, we detail how we identify the source and the sink and thus perform backward taint analysis accordingly.

6.4.1.1 Report Analysis & Taint Source Identification

The Linux kernel has a variety of debugging features implemented in different ways (e.g., `BUG`, `WARN`, and `KASAN`). However, most of them follow the same pattern. That is enforcing checks during the kernel execution and examining whether pre-defined conditions are satisfied. If the conditions do not hold, then the kernel runs into an error state and logs

critical information for debugging purposes. Following the critical information logging, the kernel may take further action to panic itself or kill the current process.

```
1  // in drivers/vhost/vhost.c
2  void vhost_dev_cleanup(struct vhost_dev *dev)
3  {
4      WARN_ON(!list_empty(&dev->work_list));
5      if (dev->worker) {
6          kthread_stop(dev->worker);
7          dev->worker = NULL;
8          dev->kcov_handle = 0;
9      }
10 }
11 // in include/asm-genric/bug.h
12 #define WARN_ON(condition) ({      \
13   int __ret_warn_on = !!(condition);    \
14   if (unlikely(__ret_warn_on))     \
15     __WARN();      \
16   unlikely(__ret_warn_on);     \
17 })
```

Listing 6.2. The code snippet that performs explicit checking.

Take, for example, the case shown in Listing 6.2. The function `vhost_dev_cleanup()` cleans the worker attached to the `vhost_dev` device. In line 4, the kernel examines the `work_list`. If the `WARN_ON` macro deems the list is empty, the kernel continues its execution at line 5, which performs the cleanup task. Otherwise, the kernel will execute the code in `WARN_ON` macro and log the error. In this example, the error is reported if and only if the pre-defined condition "`!list_empty(&dev->work_list)`" is true at runtime. Therefore, the variable `dev->work_list` in the condition indicates a cause of the bug and should become the starting point of our analysis (i.e., the taint source of our backward analysis). In this example, the kernel developers explicitly formulate the pre-defined condition as an expression and pass it to the macro `WARN_ON` for error handling. However, for some other debugging features, the checking is instrumented by a compiler or completed by hardware instead of a piece of source code written by kernel developers. For these features, the condition is implicitly formulated and cannot be identified from the kernel source code. In the following, we describe how we deal with various debugging features and their error logging mechanisms and thus identify the taint source.

Explicit Checking. Similar to the aforementioned example, the kernel developers explicitly formulate the checking as an expression and pass it to the standard debugging

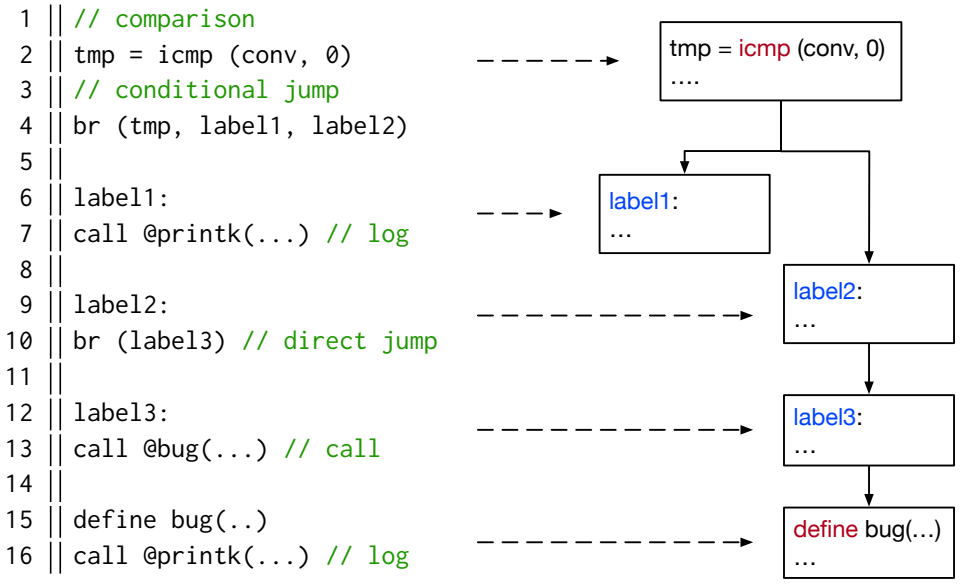


Figure 6.2. An illustrating example and its dominator tree, which demonstrate two different methods of logging kernel errors. The line 7 is a logging statement responsible for kernel error recording. The line 15 is the wrapper of the logging statement at line 16. The variable `conv` in line 1 is the taint source that our proposed approach identifies. Note that for simplicity we place the two error logging functions at two different branches sharing the same conditional jump block. In the real world, the error logging cannot occur in this way.

features such as `WARN_ON` and `BUG_ON`. Inside these macros, it is a patterned code block that includes a condition statement and a logging statement that will be executed if the condition is satisfied. Apart from this standardized way to log kernel errors, the developers can also build their own macro that wraps a logging statement in a helper function (e.g., the code in the line 15 & 16 shown in Figure 6.2).

To identify the condition that triggers the execution of the logging statement and thus pinpoint the taint source, we first trace back along the dominator tree until we find a dominator basic block, the last statement of which is a conditional jump (e.g., given the wrapped logging statement in line 16 in Figure 6.2, the line 4 is the statement linking to the dominator basic block). Second, we treat the corresponding comparison as the condition that triggers the execution of the error logging (e.g., the line 2 in Figure 6.2). Finally, we extract the corresponding variable in the condition as our taint source (e.g., `conv` in Figure 6.2).

Implicit Checking. Implicit checking refers to the situation where the checking is not part of the kernel source code but instrumented by a compiler or completed by hardware. For implicit checking done by compiler instrumentation, Kernel Address Sanitizer (KASAN) is such an example in which KASAN-enabled compiler instruments

every memory access so that the kernel could examine whether the access to a memory address is legal. KASAN relies on shadow memory to record the memory status. If the instrumented kernel, for example, touches a freed memory region, it will generate a bug report indicating the instruction that triggers a use-after-free error. Regarding the implicit checking done by interrupts (e.g., general protection fault detected by MMU), the interrupt handling routine is responsible for logging the corresponding instruction.

```
1 // source code
2 walk->offset = sg->offset;
3
4 // pseudo binary code after instrumentation
5 kasan_check_read(&sg->offset, sizeof(var));
6 tmp = LOAD(&sg->offset, sizeof(var)); // first access
7 kasan_check_write(&walk->offset, sizeof(var));
8 STORE(tmp, &walk->offset); // second access
```

Listing 6.3. The code snippet that performs implicit checking.

From bug reports generated by these debugging mechanisms, we can easily identify the instruction that performs the invalid memory access. With this information in hand, our next step is to identify the variable associated with that invalid memory access. However, the binary instruction enclosed in the report contains no type information. To deal with this problem, from the debugging information, we map binary instructions with their corresponding statements in the source code. Suppose the mapped source code is a simple statement with only one load or store. In that case, we directly conclude that this statement is the one causing the illegal memory access and treat the operand variable as a taint source. However, if the identified instruction links to a compound statement involving multiple memory loads and stores (e.g., `walk->offset = sg->offset` depicted in Listing 6.3), we perform further analysis. To be specific, we first examine the bug report and pinpoint the specific instruction that captures the kernel error. Then, we treat the memory access associated with the error-catching instruction as our taint source. To illustrate this, we again take, for example, the case shown in Listing 6.3. The bug report indicates the error is captured by the statement `kasan_check_read(&sg->offset ↪ , sizeof(var))` which associates with `sg->offset`. We deem `sg->offset` in line 2 as the taint source.

6.4.1.2 Taint Propagation & Sink Identification

Recall that backward taint analysis aims to find vulnerable structures, i.e., those structures involved in the error specified in the given bug report. To do it, we again extract the call

trace from the bug report. Based on the trace, we then construct its control flow graph and propagate the taint source backward on the graph.

Along with the backward propagation, we use the following strategy to perform variable tainting. If the tainted variable is a field of a nested structure or a union variable, we further taint its parent structure variable and treat the parent structure as a vulnerable structure. The reason is that the nested structure or the union variable is part of the parent structure variable in the memory. If a field of the nested structure or the union variable carries an invalid value, it likely results from inappropriate use of its parent structure variable.

When backward taint propagation encounters a loop, we also propagate the taint to the loop counter if the taint source was updated inside the loop. An example of this practice is some of the out-of-bound access errors in which the loop counter is corrupted, unexpectedly enlarged, and eventually used as an offset to reach out to an invalid memory region. By extending the taint to the loop variable, we can include the corrupted variable, which could further help us identify other structure variables relevant to the corruption.

In this work, we terminate our backward taint process until one of the following conditions holds. First, we terminate our taint analysis if the backward propagation reaches out to the definition of a tainted variable. Second, we terminate our taint propagation if it reaches out to a system call's entry, an interrupt handler, or the entry of the function that starts the scheduler of work queue. It is simply because they indicate the sites where the kernel debugging features start to trace kernel execution for later stage debugging. It should be noted that, while performing backward taint propagation, we also extend propagation to the aliases of the tainted variable. In this work, we treat structural types of all the taint variables as the vulnerable structure candidates for isolation.

6.4.1.3 Pinpoint Allocation Sites

With the candidate vulnerable structures in hand, our next step is to first track down all the sites of heap memory allocation. Then, we examine whether allocated objects at these sites are in the types of candidate structures. In this work, we preserve all the objects that survive the checks above, treat them as our final vulnerable object candidates and record their allocation sites.

To pinpoint the sites of heap memory allocation, we search for critical kernel functions in the kernel source code. In Linux kernel, there are two types of kernel functions used by SLAB/SLUB allocator to allocate memory on the heap. One is `kmalloc` series which

are used for object allocation on the general cache/zone. The other is `kmem_cache` series which are designed for allocation on the special cache/zone. In our design, we deem the invocation of these functions in the kernel code as the sites of memory allocation.

To determine the type of objects allocated at these sites, we analyze the return values of these functions. The reasons are ❶ their return values are always pointers referencing the objects allocated, and ❷ by analyzing the type of the return value, we can easily point out whether the type of an allocated object is within the set of candidate structures. To be more specific, when analyzing the types of return values, we follow a use-def chain and resolve memory alias. We implement our use-def chain analysis by using LLVM. As a result, when performing use-def analysis, we keep track of those instructions relevant to type casting, pointer dereferencing, and argument passing. The operands of these instructions explicitly reveal the object type. By using this information, we can easily infer and conclude the type of each allocated object accordingly. With all the allocation sites for vulnerable structure candidates, we obtain their binary addresses in the kernel image using extracted debug information.

6.4.2 eBPF-based Isolation

In this work, we extend eBPF mechanism to enforce the isolation of vulnerable object on-demand and on-the-fly. Here, we detail how to enrich the functionality of eBPF mechanism with more helper functions, how to utilize these helper functions to divert allocation operations and skip free operations.

As illustrated in Figure 6.3, the eBPF program includes a kmalloc handler, a kfree handler, and a BPF map structure. When the eBPF program is installed, the kernel will overwrite the call instruction of identified allocation sites and all free sites with `int 3`. This instruction traps the kernel into interrupt handlers where kmalloc hanlder and kfree handler are registered respectively.

If the object other than the vulnerable structure is going to be allocated, the kmalloc handler will not be executed and the returned memory is in the region managed by SLAB/SLUB allocator. Otherwise, the kmalloc handler is triggered and does the following things. First, it obtains the execution context - the value of registers and extract the request size (e.g., `rdi` if `kmalloc` is the called over x86 convention). Then, the kmalloc handler invokes helper function `bpf_vmalloc()` to obtain a memory piece with the requested size. After this, the handler adds the start address of the returned memory in the BPF map. This BPF map records the addresses of all vulnerable object after the eBPF program is installed. Finally, the kmalloc handler invokes helper function `bpf_over_return`

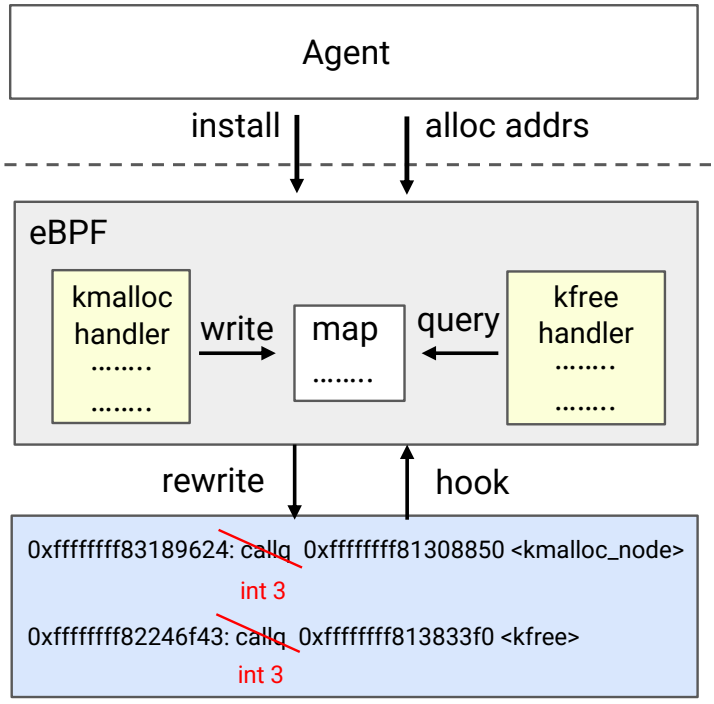


Figure 6.3. Illustration of eBPF program in the kernel space. The eBPF program rewrite allocation sites and free sites with `int 3` to trap kernel into kmalloc handler and kfree handler respectively. The kmalloc handler diverts the allocation to vmalloc region and records the allocated address in the eBPF map. The kfree handler query the eBPF map to examine whether the free request is for vulnerable object. If it is, the kfree handler will skip the free operation and not reclaim the memory to achieve one-time allocation.

to return the allocated address. Note that in the procedure, the original kernel allocation function is not executed but instead allocation happens in vmalloc region.

As mentioned in Section 6.2.3, eBPF relies on helper functions to interact with the other parts of the kernel and access specific kernel data (e.g., map data, time since boot up). To support the kmalloc handler in the eBPF program, we extend the set of helper functions by adding `bpf_vmalloc()`. In the `vmalloc()` function, the kernel make attempts to do allocation for several times and sleep if all these attempts fail. As kmalloc handler is executed in the interruption context, sleep is not allowed. Therefore, in the `bpf_vmalloc()` helper function, the kernel will avoid sleeping by directly return back 0 to indicate failure. Correspondingly, the kamlloc handler will deal with this failure by stepping back to SLAB/SLUB allocator.

When the calling to `kfree` is invoked in kernel, the kfree handler is executed. Similar to the kmalloc handler, the kfree handler first obtains the execution context and extract the address arguments of `kfree`. Then, the kfree handler queries the BPF map to examine

whether the freed address belongs to the vulnerable objectin the vmalloc region. If not, the kfree handler will transfer to execute the `kfree()` function. Otherwise, the kfree handler will directly return back without doing nothing. In this case, the memory holding the vulnerable object will continue to be tracked in the eBPF program but never be recycled, which becomes an one-time allocation specific to the reported vulnerability.

6.5 Implementation

Based on LLVM infrastructure and eBPF mechanism, we implement our idea as a tool and name it after `HotBPF`.

Vulnerable structure identification. The input of our tool is LLVM IR and a single bug report. In our implementation, we employ the approach in previous works [5, 64] to generate the bitcode files. Briefly, we patch the LLVM compiler to dump bitcodes before invoking any compiler optimization passes. In this way, we can prevent compiler optimization from influencing the accuracy of our analysis. Recall that we extract the call trace from the bug report. The call trace indicates the functions that have been called but not yet returned when the kernel is experiencing errors. In this extracted call trace, the function that has been called last could be the one instrumented by the compiler. It does not indicate the buggy function contributing to the error. As such, we neglect these functions in the call trace and start our analysis from the statement that activates the debugging feature.

When using the backward taint analysis to identify vulnerable structures, `HotBPF` uses three instructions to extract the structures' type information. The first instruction is `BitCast` in which the types before and after casting are specified. `HotBPF` records the types extracted from this instruction as vulnerable structures. The second instruction is `Getelementor` that contains a pointer referencing a kernel object and the object's corresponding type information. Through the analysis of this instruction, we can quickly obtain vulnerable structures. The third instruction is `CallInst`. We infer the type information from the callee's prototype and record the structural type as vulnerable structures. As is mentioned earlier, we treat system calls' entries, interrupt handlers, and workqueue processings as our taint sink. In our work, we manually annotate all these sinks based on their naming patterns.

eBPF-based isolation. As is described in Section 6.4.2, we add one helper function `bpf_vmalloc` to the kernel to support kmalloc handler. In our implementation, this new helper function is assigned with a new call ID so that the eBPF interpret can recognize

it. The BPF map maintained in the eBPF program is in the type of `BPF_MAP_TYPE_HASH` which supports quick query. The eBPF program is synthesized for each vulnerabiilty using a Phython script and we leverage Clang tool-chain to compile the eBPF program.

6.6 Evaluation

In this section, we first quantify `HotBPF`'s effectiveness in identifying vulnerable objects. Then, we evaluate the performance overhead of `HotBPF` using benchmarks at different levels of granularity.

6.6.1 Experiment Setup & Design

Syzbot is a bug reporting platform that well archives the kernel bugs identified by Syzkaller. To evaluate our tool - `HotBPF`, we select kernel bugs and their reports from the platform as our test cases. While selecting these bugs, we follow two different strategies.

Our first strategy is a purely random selection process that follows two criteria. First, the bug report has to attach a PoC program so that we can build a customized benchmark to measure the performance overhead of `HotBPF` in a fine granularity. Second, the reported kernel error needs to have multiple reported bug behaviors. This is because the bug reports for different behaviors are of different quality. Typically, reports generated by KASAN include more information than those produced by implicit check. Using this diverse set of bug reports, we can evaluate how the effectiveness of static analysis can influenced by report quality. Our second bug selection strategy is a process dedicated to different kernel versions. To be specific, we select bugs from 5 different Linux kernel versions. From each kernel version, we choose two recently-reported reproducible kernel bugs as our test cases. Following these two criteria, we construct a test corpus containing 22 bugs with 30 bug reports.

We asked our kernel professional team to thoroughly and manually inspect every bug in our corpus and identify the vulnerable object to build our ground-truth dataset. In this way, we can evaluate `HotBPF`'s false negatives or, in other words, understand how complete GREBE could identify vulnerable objects. It should be noted that the Linux kernel's codebase is huge and sophisticated. Given a kernel bug, it usually requires extensive manual efforts and significant expertise, spending hundreds of hours to perform through manual analysis for exploring all the possible bug reports. In addition to the manual effort above, we also build a customized benchmark using the PoC program

attached with the bug report. The benchmark is constructed by removing system calls that can trigger bugs but keep system calls that operate vulnerable objects. By repeatedly running the tailored PoC programs, we can obtain how much time it takes the eBPF program to provide protection for every vulnerable object operation.

Given a kernel bug and its reports of our selection, we build a kernel image using the version specified in the report. We extend the kernel image by adding the new eBPF helper functoin and install it to a a bare-metal desktop. This desktop has a Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz and 16GB RAM. One benchmark we used for performance overhead evaluation is LMbench which invokes system calls and measure the latency as metrics. As such, LMbench works at a coarse granularity. Another benchmark, as is mentioned above is the benchmark customized from PoC programs, which works at a fine granularity. For each bug and corresponding reports, we ran both benchmark for three rounds and calculate the average latency as the representative overhead.

6.6.2 Experiment Results

False positives/False Negatives. As is mentioned above, we put manual efforts to investigate test cases and identify their vulnerable object as the ground-truth. Compared the statically identified vulnerable structures with this ground-truth set, we examine how soudn and how complete `HotBPF` can identify the vulnerable structures. In our experiment, the test cases used for our false negative study are listed in Table 6.1. Our manual inspection shows that `HotBPF` has no false negative over the dataset. In other words, `HotBPF` does not miss any vulnerable object in all test cases. This is because the identification technique of `HotBPF` is conservative enough as we will not only include the structure directly associated with the report record but also consider its parent structure. From Table 6.1 we can further observe that for 6 out of 30 bug reports, `HotBPF` doesn't have false positive which means that we don't identify kernel structures that are not vulnerable. For the remaining 24 bug reports, the highest false positive is 6 while most cases are one or two. The number of false positive will increase the performance burden. However, as we will see very soon that the performance overhead is very low (on average 2% - 3%) which demonstrates the effectiveness of `HotBPF`.

Table 6.1 also presents how the quality of reports influences the identification results. As we mentioned above that the same bug can be manifested in different program sites which generate reports containing different amount of useful information. For all the cases with more than one reports, Table 6.1 shows that the difference of false positive

SYZ ID	Groud-truth Vulnerable Structures	Statically Identified Vulnerable Structures	FP/FN	# of Alloc Sites
f0ec9a3 [178]	hci_conn	hci_conn, hci_dev	1/0	2
ba1aecb [179]	nft_trans	nft_object, nft_trans	1/0	2
37a6804 [180]	nft_trans	nft_flowtable, nft_rule, nft_set, nft_table, nft_trans	4/0	5
4cf5ee7 [181]	vb2_buffer	file, vb2_buffer, video_device	2/0	5
be93025 [182]	vb2_buffer	vb2_buffer, video_device	1/0	3
4c0ccb2 [183]	vb2_fileio_data	vb2_fileio_data, video_device	1/0	2
1be0e67 [184]	vb2_fileio_data	vb2_fileio_data, video_device	1/0	2
3a6c997 [185]	void*	ip_set	1/0	2
6491ea8 [186]	void*	ip_set	1/0	2
1b726e0 [187]	p9_conn	p9_client, p9_conn, p9_trans_fd	2/0	3
001410d [188]	p9_conn	p9_conn	0/0	1
17535f4 [189]	vb2_buffer, vb2_vmalloc_buf	vb2_buffer, vb2_vmalloc_buf, video_device	2/0	6
2389bfc [190]	tcindex_filter_result	tc_action, tcindex_filter_result	1/0	3
a152f63 [191]	delayed_uprobe	delayed_uprobe	0/0	1
3b7409f [192]	rdma_id_private	rdma_id_private, sockaddr, ucma_context, ucma_file	3/0	4
da2591e [193]	rdma_id_private	rdma_id_private, ucma_context, ucma_file	2/0	3
c897760 [194]	nft_table	nft_set, nft_trans, nft_table	2/0	3
437bf61 [195]	nft_table	nft_flowtable, nft_rule, nft_set, nft_trans, nft_table	4/0	5
d463fb9 [196]	void*	ip_set, void*	1/0	2
b8febdb [197]	rdma_id_private	file, rdma_id_private, sockaddr, ucma_context, ucma_file	4/0	6
cbb2898 [198]	sctp_endpoint	crypto_shash, sctp_endpoint	1/0	2
7be8b46 [199]	sk_buff	sk_buff	0/0	6
57e9851 [200]	hci_conn	hci_conn, sco_conn	1/0	2
39d35c9 [201]	l2cap_chan	l2cap_chan	0/0	1
6a03985 [202]	route4_filter	Qdisc, route4_filter, tcf_block, tcf_chain, tcf_proto	4/0	10
5bb09c0 [203]	route4_filter	Qdisc, route4_bucket, route4_filter, route4_head, tcf_block, tcf_chain, tcf_proto	6/0	12
e4be308 [204]	void*	crypto_shash, kdf_sdesc, shash_desc, void*	3/0	17
5a8b026 [205]	net_device	net_device	0/0	6
a6b8566 [206]	super_block	super_block	0/0	2
b5b251b [207]	usb_hcd	urb, usb_ctrlrequest, usb_device, usb_hcd	3/0	24

Table 6.1. Effectiveness of static analysis in `HotBPF`. The "SYZ ID" column is the case ID. The "Ground-truth Vulnerable Structures" means the vulnerable structure manually identified. The "Statically Identified Vulnerable Structures" column indicates the structures that are identified by the static analysis tools that are utilized by `HotBPF`. The "FP/FN" column stands for False Positive (# of identified structures that are not vulnerable) and False Negative (# of vulnerable structures that are not identified). The "# of Alloc Site" column means how many allocation sites of identified vulnerable structures are pinpointed. Note that the several continuous lines in the same color represent reports associated with the same vulnerability.

number is within 3. Therefore, `HotBPF` is capable of calculating accurate results given a diverse of bug reports.

Performance Overhead Table 6.2 samples the performance overhead of `HotBPF` over four cases using LMbench. First, the results show that the average overhead is from 2% to 3% which is very low compared with existing defenses in the Linux kernel [208]. Second, we can observe that the overhead of system calls related to networking functionality is relatively high compared with other system calls. That is because the network stack

	w/o defense	f0ec9a3 [178]		cbb2898 [198]		57e9851 [200]		5a8b026 [205]	
syscall()	0.372	0.382	2.73%	0.374	0.53%	0.373	0.17%	0.371	-0.30%
read()	0.461	0.458	-0.57%	0.457	-0.82%	0.456	-0.90%	0.456	-1.03%
write()	0.421	0.419	-0.48%	0.419	-0.40%	0.419	-0.45%	0.419	-0.53%
stat()	0.741	0.743	0.26%	0.739	-0.31%	0.741	-0.02%	0.742	0.16%
fstat()	0.458	0.461	0.58%	0.461	0.71%	0.461	0.58%	0.460	0.52%
open()/close()	1.523	1.528	0.36%	1.522	-0.08%	1.525	0.17%	1.527	0.27%
select() (10 fds)	0.513	0.510	-0.44%	0.511	-0.31%	0.512	-0.05%	0.509	-0.64%
select() (100 fds)	1.092	1.094	0.14%	1.095	0.23%	1.096	0.33%	1.092	0.00%
Pipe I/O	4.830	4.944	2.44%	4.940	2.36%	4.913	1.81%	4.939	2.33%
UNIX socket I/O	7.344	8.011	9.08%	7.982	8.68%	8.038	9.44%	7.972	8.55%
fork() + exit()	75.244	75.731	0.65%	76.950	2.27%	77.117	2.49%	76.588	1.79%
fork() + execve()	259.587	262.239	1.02%	263.628	1.56%	263.429	1.48%	265.064	2.11%
fork() + /bin/sh	803.286	812.190	1.11%	815.905	1.57%	828.238	3.11%	842.476	4.88%
UDP socket I/O	8.368	9.709	16.03%	9.787	16.97%	9.829	17.46%	9.774	16.81%
TCP socket I/O	10.298	11.600	12.64%	11.555	12.21%	11.664	13.27%	11.588	12.53%
Average			2.56%		2.61%		2.80%		2.70%

Table 6.2. Performance overhead of `HotBPF` using LMbench. The "w/o defense" column indicates the latency of system calls when `HotBPF` is not enabled. The two columns of each cases stands for the latency and performance drop. All latency is in ms.

contains more allocation and free operations. Using the benchmark customized from PoC programs, we measure the time spent in eBPF program. The results show that the average time in kmalloc handler is 0.023 ms while the time in kfree handler is 0.006 ms which is negligible compared with core Linux kernel operations.

6.7 Discussion

In this section, we briefly discuss the limitation of `HotBPF`. `HotBPF` identifies vulnerable objects and diverts the allocation to vmalloc region. When the vulnerable object is DMA-based object (e.g., data region of networking packet), this separation cannot be achieved. The reason is because DMA-based object requires physically continuous memory which cannot be met by vmalloc region. Howver, we argue that this limitation doesn't dilute the contribution of `HotBPF` as it's very rare to see that the vulnerable object is DMA-based.

6.8 Conclusion

Oftentimes there can be a large window between a kernel vulnerability disclosure and its remediation, leaving the system open for exploitation. In this work, we present the design of a mechanism named `HotBPF` that can protect the Linux kernel from

memory exploitation during this time window. This protection mechanism has the following advantages. First, it can automatically deploy protection immediately after the vulnerability disclosure. Second, this protection can be enabled on-the-fly which means there is no need to disrupt critical services to recompile and reboot the system. Third, compared with existing memory exploitation mitigations in the Linux kernel, this protection can offer stronger security protection by preventing cross-cache exploitation. Fourth, this protection is independent of hardware features and hypervisor virtualization. So it can be widely deployed in a variety of scenarios (e.g., embedded systems, desktops, and cloud servers). Last but not least, this protection is lightweight as it introduces an overall 2% - 3% performance overhead.

Chapter 7
Conclusion

7.1 Summary

In this dissertation, we investigate the hypothesis that by researching exploitable design patterns, we can design more advanced protection against memory-based exploitation techniques.

Towards this goal, we presented `FUZE`: an effective framework to explore corruption capability of kernel UAF vulnerabilities. We showed that `FUZE` could identify various system calls essential for exploiting an UAF vulnerability. We demonstrated the utility of `FUZE`, using 15 real-world kernel UAF vulnerabilities. We presented that `FUZE` could provide security analysts with critical ability to expedite exploit generation for kernel UAF vulnerabilities, and thus support more accurate evaluation over the exploitability of design patterns in the system.

With more comprehensive corruption capabilities, we presented `ELOISE` in which we identify elastic objects as a design pattern commonly adopted in commodity operating systems including `Linux`, `FreeBSD`, and `XNU`. After this, `ELOISE` demonstrates that this design pattern is exploitable by using the explored corruption capability. Furthermore, taking a close look at existing kernel defense mechanisms, we discover that none can be useful or practical for hindering the threat of elastic kernel objects. Inspired by this finding, we introduce a new lightweight defense mechanism. This defense pioneers the design of `HotBPF`.

In addition, we built `SLAKE` to facilitate the process of exploitability evaluation. Technically, `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 obtain exploitable primitives. Moreover, we model commonly-adopted

exploitation methods and develop a technical approach to facilitate the slab layout adjustment, which plays an essential role in exploitability evaluation.

However, However, obtaining primitives is insufficient to accurately evaluate exploitability because of widely deployed exploit mitigations in an off-the-shelf modern Linux kernel. We therefore propose `KEPLER` to facilitate exploit generation by automatically generating a "single-shot" exploitation chain. `KEPLER` accepts as input a control-flow hijacking primitive and bootstraps any kernel ROP payload by symbolically stitching an exploit chain taking advantage of prevalent kernel coding style and corresponding gadgets. Comparisons with previous automatic exploit generation techniques and previous kernel exploit techniques show `KEPLER` effectively facilitates evaluation of control-flow hijacking primitives in the Linux kernel.

Base on above works on investigating design patterns and evaluating their exploitability, we presented `HotBPF`: a protection that can prevent exploitable design patterns from being misused by attackers to perform memory exploitation. `HotBPF` can automatically deploy protection immediately after the vulnerability disclosure. Besides, `HotBPF` supports on-the-fly enabling so that there is no need to disrupt critical services to recompile and reboot the system. Compared with existing memory exploitation mitigations in the Linux kernel, `HotBPF` can offer stronger security protection by preventing cross-cache exploitation. `HotBPF` is lightweight as it introduces an overall 2% - 3% performance overhead. Additionally, `HotBPF` is independent of hardware features and hypervisor virtualization. So it can be widely deployed in a variety of scenarios (e.g., embedded systems, desktops, and cloud servers).

7.2 Future Directions

As computer system is evolving very quickly, we are seeing more and more new applications, new infrastructures, and new architectures. These new contexts contain new exploitable design patterns that need to be researched and mitigated. Here, I briefly describe two contexts and corresponding exploitable design patterns as examples.

GPU & cuDNN downgrading. Nowadays, lots of applications, especially machine learning services and blockchain networking run over GPU. Due to its novel architecture, GPU contains new exploitable design patterns that are seldom studied in previous research. For example, Sang-Ok et al [209] introduces a novel attack vector against deep learning systems, namely GPU memory manipulation. This exploitation technique can

stealthily perturb the predictions from the deep learning applications so that they are no longer different from random guessing. As the design pattern misused in this attack is totally different from those used in CPU-based attacks, new techniques are needed to identify them and evaluate their exploitability accordingly. Moreover, another challenge lies in how to leverage GPU hardware features to design protections that are not only effective but also lightweight.

IoT devices & ret2bootloader. Almost one decade ago, Aurélien et al [210] introduces ret2bootloader as a unique exploitation technique against IoT devices. Unfortunately, we still cannot mitigate it until today. The challenges are many-fold. To name a few, first, the IoT devices are resource constrained. Typically, these devices don't have enough computing resource and energy power to support widely used defenses in desktops and cloud servers. Some low-end devices even don't have basic MPU and MMU for memory isolation. Therefore, only software-based and lightweight protection can be deployed to mitigate the unique exploitation design patterns. Second, most IoT devices use real-time system - running tasks have time deadlines. When designing mitigations, we need to ensure that these time deadlines can still be met after the mitigation is enabled. This is also very challenging and require significant research efforts.

Bibliography

[1] Kemerlis, V. P., M. Polychronakis, and A. D. Keromytis (2014) "ret2dir: Rethinking Kernel Isolation," in Proceedings of the 23th Conference on USENIX Security Symposium (USENIX Security).

[2] Wang, Y. (2016) "A New CVE-2015-0057 Exploit Technology," in Blackhat Asia.

[3] Zhao, H. (2017), "PWN2OWN 2017 Linux kernel privilege escalation analysis," https://zhuanlan.zhihu.com/p/26674557.

[4] Wu, W., Y. Chen, J. Xu, X. Xing, W. Zou, and X. Gong (2018) "FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities," in Proceedings of the 27th USENIX Security Symposium (USENIX Security).

[5] (2019), "Code and Exploits for ELOISE," https://github.com/chenyueqi/w2l.

[6] Chen, Y. and X. Xing (2019) "SLAKE: Facilitating Slab Manipulation for Exploiting Vulnerabilities in the Linux Kernel," in Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS).

[7] Wu, W., Y. Chen, X. Xing, and W. Zou (2019) "KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities," in Proceedings of the 28th USENIX Security Symposium (USENIX Security).

[8] Guo, P. J., T. Zimmermann, N. Nagappa, and B. Murphy (2010) "Characterizing and Predicting Which Bugs Get Fixed: An Empirical Study of Microsoft Windows," in Proceedings of the 32th International Conference on Software Engineering (ICSE).

[9] Penalver, C. M. (2016), "How to triage bugs," https://wiki.ubuntu.com/Bugs/Importance.

[10] Mitre (2018), "CWE-416: Use After Free," https://cwe.mitre.org/data/definitions/416.html.

[11] Azad, B. (2016), "Mac OS X Privilege Escalation via Use-After-Free: CVE-2016-1828," https://bazad.github.io/2016/05/mac-os-x-use-after-free/#use-after-free.

[12] JNDOK (2016), "Analysis and exploitation of Pegasus Kernel vulnerabilities," https://jndok.github.io/2016/10/04/pegasus-writeup/.

[13] AVGERINOS, T., S. K. CHA, B. L. T. HAO, and D. BRUMLEY (2011) "AEG: Automatic Exploit Generation," in Proceedings of the 2011 Network and Distributed System Security Symposium (NDSS).

[14] BAO, T., R. WANG, Y. SHOSHITAISHVILI, and D. BRUMLEY (2017) "Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits," in Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P).

[15] BRUMLEY, D., P. POOSANKAM, D. X. SONG, and J. ZHENG (2008) "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," in Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P).

[16] REPEL, D., J. KINDER, and L. CAVALLARO (2017) "Modular Synthesis of Heap Exploits," in ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS).

[17] XU, W., J. LI, J. SHU, W. YANG, T. XIE, Y. ZHANG, and D. GU (2015) "From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel," in Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security (CCS).

[18] EDGE, J. (2014), "The kernel address sanitizer," https://lwn.net/Articles/612153/.

[19] NIKOLENKO, V. (2016), "Linux Kernel ROP - Ropping your way to # (Part 1)," https://www.trustwave.com/Resources/SpiderLabs-Blog/Linux-Kernel-ROP---Ropping-your-way-to---(Part-1)/.

[20] ARGYROUDIS, P. (2012), "The Linux kernel memory allocators from an exploitation perspective," https://argp.github.io/2012/01/03/linux-kernel-heap-exploitation/.

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

[22] GRUSS, D., C. MAURICE, A. FOGH, M. LIPP, and S. MANGARD (2016) "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS).

[23] JANG, Y., S. LEE, and T. KIM (2016) "Breaking Kernel Address Space Layout Randomization with Intel TSX," in Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS).

[24] Google (2019), "syzkaller - kernel fuzzer," https://github.com/google/syzkaller.

[25] (2017), "angr - a binary analysis framework," http://angr.io/index.html.

[26] (2018), "Kernel exploit release, 2018," https://github.com/ww9210/Linux_kernel_exploits.

[27] Database, N. V. (2017), "CVE-2017-7374 Detail," https://nvd.nist.gov/vuln/detail/CVE-2017-7374.

[28] PaX/Grsecurity (2017), "PaX/Grsecurity –> KSPP –> AOSP kernel: Linux kernel mitigation checklist," https://github.com/hardenedlinux/grsecurity-101-tutorials/blob/master/kernel_mitigation.md.

[29] Hund, R., C. Willems, and T. Holz (2013) "Practical Timing Side Channel Attacks against Kernel Space ASLR," in Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P).

[30] Evtyushkin, D., D. V. Ponomarev, and N. B. Abu-Ghazaleh (2016) "Jump over ASLR: Attacking branch predictors to bypass ASLR," in Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).

[31] Lipp, M., M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg (2018) "Meltdown," in arXiv preprint arXiv:1801.01207.

[32] Jurczyk, M. and G. Coldwind (2011), "SMEP: What is it, and how to beat it on Windows," http://j00ru.vexillium.org/?p=783.

[33] Lu, K., M. Walter, D. Pfaff, and S. 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).

[34] Hu, H., S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang (2016) "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P).

[35] Cha, S. K., T. Avgerinos, A. Rebert, and D. Brumley (2012) "Unleashing Mayhem on Binary Code," in Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P).

[36] Mothe, R. and R. R. Branco (2016) "DPTrace: Dual Purpose Trace for Exploitability Analysis of Program Crashes," in Blackhat USA.

[37] Shoshitaishvili, Y., R. Wang, C. Hauser, C. Kruegel, and G. Vigna (2015) "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware," in Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS).

[38] Stephens, N., J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna (2016) "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS).

[39] Corbet, J. (2016), "Exclusive page-frame ownership," https://lwn.net/Articles/700647/.

[40] Proskurin, S., M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis (2020) "xMP: selective memory protection for kernel and user space," in Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P).

[41] Konovalov, A. (2018), "A proof-of-concept exploit for CVE-2017-18344," https://github.com/xairy/kernel-exploits/blob/master/CVE-2017-18344/poc.c.

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

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

[44] 0x3f97 (2018), "cve-2017-8890 root case analysis," https://0x3f97.github.io/exploit/2018/08/13/cve-2017-8890-root-case-analysis/.

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

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

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

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

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

[50] Hund, R., C. Willems, and T. Holz (2013) "Practical Timing Side Channel Attacks Against Kernel Space ASLR," in Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P).

[51] Evtyushkin, D., D. Ponomarev, and N. Abu-Ghazaleh (2016) "Jump over ASLR: Attacking branch predictors to bypass ASLR," in Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).

[52] Gruss, D., C. Maurice, and A. Fogh (2016) "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS).

[53] Lipp, M., M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg (2018) "Meltdown: Reading Kernel Memory from User Space," in Proceedings of the 27th USENIX Security Symposium (USENIX Security).

[54] dp304 (2018), "Alternative to flexible array members for avoiding multiple allocations," https://www.gamedev.net/forums/topic/696730-alternative-to-flexible-array-members-for-avoiding-multiple-allocations/.

[55] Spudd86 (2010), "Flexible array member in C-structure," https://stackoverflow.com/questions/3047530/flexible-array-member-in-c-structure.

[56] Corbet, J. (2012), "Supervisor mode access prevention," https://lwn.net/Articles/517475/.

[57] ——— (2017), "The current state of kernel page-table isolation," https://lwn.net/Articles/741878/.

[58] Labs, A. (2020), "Grooming the iOS Kernel Heap," https://azeria-labs.com/grooming-the-ios-kernel-heap/.

[59] Hautebas, M. (2018), "empty_list - exploit for p0 issue 1564 (CVE-2018-4243) iOS 11.0 - 11.3.1 kernel r/w," https://github.com/Jailbreaks/empty_list.

[60] Beer, I. (2017), "Exception-oriented exploitation on iOS," https://googleprojectzero.blogspot.com/2017/04/exception-oriented-exploitation-on-ios.html.

[61] Chen, Y., X. Xing, and J. Su (2019), "Hands off and putting SLAB/SLUB fengshui in a blackbox," https://i.blackhat.com/eu-19/Wednesday/eu-19-Chen-Hands-Off-And-Putting-SLAB-SLUB-Feng-Shui-In-A-Blackbox.pdf.

[62] OpenWall (2020), "John the Ripper password cracker," https://www.openwall.com/john/.

[63] Stallman, R. M. (2019), "GNU Debugger," https://www.gnu.org/software/gdb/.

[64] Lu, K. and H. Hu (2019) "Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis," in Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS).

[65] Lu, K., A. Pakki, and Q. Wu (2019) "Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences," in Proceedings of the 28th USENIX Security Symposium (USENIX Security).

[66] Kengiter and adityapakki (2019), "Crix: Detecting Missing-Check Bugs in OS Kernels," https://github.com/umnsec/crix.

[67] Research, M. (2020), "Z3," https://github.com/Z3Prover/z3.

[68] Kemerlis, V. P., M. Polychronakis, and A. D. Keromytis (2014) "ret2dir: Rethinking Kernel Isolation," in Proceedings of the 23rd USENIX Security Symposium (USENIX Security).

[69] Google (2020), "syzbot Dashboard," https://syzkaller.appspot.com/upstream.

[70] Chiang, E. (2019), "User Namespaces," https://ericchiang.github.io/post/user-namespaces/.

[71] syzbot (2018), "KASAN: use-after-free Write in dst_release," https://syzkaller.appspot.com/bug?id=bf967d2c5ba62946c61152534c8b84823d848f05.

[72] ——— (2018), "KASAN: slab-out-of-bounds Write in mpol_parse_str," https://syzkaller.appspot.com/bug?id=3d67d693e0529df8ac89ba55b00b54e5d967e021.

[73] ——— (2018), "KASAN: slab-out-of-bounds Write in pipe_write," https://syzkaller.appspot.com/bug?id=422a020e119fbac4c15d8fed114cc1696fe5c51a.

[74] Cook, K. (2017), "security things in Linux v4.14," https://outflux.net/blog/archives/2017/11/14/security-things-in-linux-v4-14/.

[75] Esser, S. (2016), "iOS 10 - Kernel Heap Revisited," .

[76] Cook, K. (2017), "security things in Linux v4.13," https://outflux.net/blog/archives/2017/09/05/security-things-in-linux-v4-13/.

[77] Hussein, N. (2017), "Randomizing structure layout," https://lwn.net/Articles/722293/.

[78] Horn, J. (2020), "Linux Email list: CONFIG_DEBUG_INFO_BTF and CONFIG_GCC_PLUGIN_RANDSTRUCT," https://www.spinics.net/lists/bpf/msg16648.html.

[79] Edge, J. (2016), "Hardened usercopy," https://lwn.net/Articles/695991/.

[80] McVoy, L. and C. Staelin (2015), "LMbench - Toos for Performance Analysis," http://lmbench.sourceforge.net/.

[81] (2015), "Phoronix Test Suite," http://www.phoronix-test-suite.com/.

[82] Chen, S., J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer (2005) "Non-Control-Data Attacks Are Realistic Threats," in Proceedings of the 14th USENIX Security Symposium (USENIX Security).

[83] Shoshitaishvili, Y., R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna (2016) "SoK:(State of) The Art of War: Offensive Techniques in Binary Analysis," in Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P).

[84] Hu, H., Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang (2015) "Automatic Generation of Data-oriented Exploits," in Proceedings of the 24th USENIX Security Symposium (USENIX Security).

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

[86] ——— (2019) "Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters," in Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS).

[87] Wang, Y., C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou (2018) "Revery: From Proof-of-Concept to Exploitable," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.

[88] Ispoglou, K. K., B. AlBassam, T. Jaeger, and M. Payer (2018) "Block Oriented Programming: Automating Data-Only Attacks," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS).

[89] Yun, I., D. Kapil, and T. Kim (2020) "Automatic Techniques to Systematically Discover New Heap Exploitation Primitives," in Proceedings of the 29th USENIX Security Symposium (USENIX Security).

[90] Cho, H., J. Park, J. Kang, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn (2020) "Exploiting Uses of Uninitialized Stack Variables in Linux Kernels to Leak Kernel Pointers," in 14th USENIX Workshop on Offensive Technologies (WOOT).

[91] Chen, W., X. Zou, G. Li, , and Z. Qian (2020) "KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities," in Proceedings of the 29th USENIX Security Symposium (USENIX Security).

[92] Mochel, P. and M. Murphy (2020), "sysfs - The filesystem for exporting kernel objects," https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt.

[93] Jones, M. (2010), "User space memory access from the Linux kernel," https://developer.ibm.com/technologies/linux/articles/l-kernel-memory-access/.

[94] Mauerer, W. (2008), "Professional Linux Kernel Architectures," Chapter 12.11.

[95] syzbot (2018), "KASAN: use-after-free Read in ___lock_acquire (2)," https://syzkaller.appspot.com/bug?id=1379b6b21a2ffecd1ea4e2b564cc7e35d9f388b2.

[96] ——— (2020), "KASAN: use-after-free Read in route4_get," https://syzkaller.appspot.com/bug?id=5bb09c0c5b65ab2ce628ba26fe7cbd06144bd952.

[97] ——— (2018), "KASAN: use-after-free Write in bpf_tcp_close," https://syzkaller.appspot.com/bug?id=6a6fd266a962be281b17c864a073675150e36ca5.

[98] ——— (2018), "KASAN: slab-out-of-bounds Write in crypto_dh_encode_key," https://syzkaller.appspot.com/bug?id=a84d6ad70b281bfc5632f272f745104fb43d219d.

[99] ——— (2018), "KASAN: slab-out-of-bounds Write in sha512_final," https://syzkaller.appspot.com/bug?id=e4be30826c1b7777d69a9e3e20bc7b708ee8f82c.

[100] ——— (2018), "KASAN: use-after-free Read in snd_timer_open," https://syzkaller.appspot.com/bug?id=e9287fe57ad2f862eedb05012481132486f3b887.

[101] ——— (2019), "KASAN: use-after-free Write in ___xfrm_policy_unlink," https://syzkaller.appspot.com/bug?id=ebeba334a8a886e3d5dc25641e201e894d4d9657.

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

[103] ——— (2018), "Dissecting a 17-year-old kernel bug," https://duasynt.com/slides/bevx-talk.pdf.

[104] ——— (2018), "Linux Kernel universal heap spray," https://duasynt.com/blog/linux-kernel-heap-spray.

[105] Vyukov, D. (2018), "syzbot and the tale of thousand kernel bugs," https://events.linuxfoundation.org/wp-content/uploads/2017/11/Syzbot-and-the-Tale-of-Thousand-Kernel-Bugs-Dmitry-Vyukov-Google.pdf.

[106] Penalver, C. M. (2016), "How to triage bugs," https://wiki.ubuntu.com/Bugs/Importance.

[107] Avgerinos, T., S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley (2014) "Automatic Exploit Generation," Commun. ACM, **57**.

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

[109] Cook, K. (2010), "CVE-2010-2963 v4l compat exploit," https://outflux.net/blog/archives/2010/10/19/cve-2010-2963-v4l-compat-exploit/.

[110] Grob, S. (2014), "Linux local root exploit for CVE-2014-0038," https://github.com/saelo/cve-2014-0038.

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

[112] Hund, R., T. Holz, and F. C. Freiling (2009) "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms," in Proceedings of the 18th USENIX Security Symposium (USENIX Security).

[113] Perla, E. and M. Oldani (2010) A Guide to Kernel Exploitation, Elsevier.

[114] Rostedt, S. (2009), "Debugging the kernel using Ftrace," https://lwn.net/Articles/365835/.

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

[116] Wang, X., H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek (2012) "Improving Integer Security for Systems with KINT," in Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI).

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

[118] Yamada, M. and J. Nikula (2019), "kcov:code coverage for fuzzing," https://github.com/torvalds/linux/blob/master/Documentation/dev-tools/kcov.rst.

[119] Pailoor, S., A. Aday, and S. Jana (2018) "MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation," in Proceedings of the 27th USENIX Security Symposium (USENIX Security).

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

[121] Corporation, T. M. (2019), "common Vulnerability and Exposures," https://cve.mitre.org/cve/.

[122] You, W., P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. 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).

[123] Foundation, T. F. (2019), "The FreeBSD Project," https://www.freebsd.org/.

[124] Project, A. O. S. (2019), "Common Android Kernel Tree," https://android.googlesource.com/kernel/common/.

[125] Mackall, M. (2005), "slob: introduce the SLOB allocator," https://lwn.net/Articles/157944/.

[126] Bovet, D. P. and M. Cesati (2010) Understanding the Linux Kernel, Elsevier.

[127] Gloger, W. (2006), "Wolfram Gloger's malloc homepage," http://www.malloc.de/en/.

[128] Schumilo, S., C. Aschermann, and R. Gawlik (2017) "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in Proceedings of the 25th USENIX Security Symposium (USENIX Security).

[129] Jeong, D. R., K. Kim, B. Shivakumar, B. Lee, and I. Shin (2019) "Razzer: Finding Kernel Race Bugs through Fuzzing," in Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P).

[130] Serebryany, K., D. Bruening, A. Potapenko, and D. Vyukov (2012) "AddressSanitizer: A Fast Address Sanity Checker." in Proceedings of 2012 USENIX Annual Technical Conference (USENIX ATC).

[131] Cha, S. K., T. Avgerinos, A. Rebert, and D. Brumley (2012) "Unleashing mayhem on binary code," in Proceedings of IEEE Symposium on Security and Privacy (SP), 2012.

[132] Edge, J. (2014), ""Strong" stack protection for GCC," https://lwn.net/Articles/584225/.

[133] Knox, S. (2016), "Real-time Kernel Protection(RKP)," https://www.samsungknox.com/en/blog/real-time-kernel-protection-rkp.

[134] Corbet, J. (2015), "Post-init read-only memory," https://lwn.net/Articles/666550/.

[135] Shutemov, K. A. (2015), "pagemap: do not leak physical addresses to non-privileged userspace," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce.

[136] Edge, J. (2011), "Extending the use of RO and NX," https://lwn.net/Articles/422487/.

[137] Nakajima, J. and S. Grandhi (2017) "Kernel Protection Using Hardware Based Virtualization," in The Linux Foundation events.

[138] Oh, M. (2017), "Detecting and mitigating elevation-of-privilege exploit for CVE-2017-0005," https://cloudblogs.microsoft.com/microsoftsecure/2017/03/27/detecting-and-mitigating-elevation-of-privilege-exploit-for-cve-2017-0005/.

[139] itsZN (2015), "Bypassing SMEP Using vDSO Overwrites," https://itszn.com/blog/?p=21.

[140] Seaborn, M. and T. Dullien (2015), "Exploiting the DRAM rowhammer bug to gain kernel privileges," https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

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

[142] Abadi, M., M. Budiu, U. Erlingsson, and J. Ligatti (2005) "Control-flow integrity," in Proceedings of the 12nd ACM conference on Computer and communications security (CCS).

[143] Criswell, J., N. Dautenhahn, and V. Adve (2014) "KCoFI: Complete control-flow integrity for commodity operating system kernels," in Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P).

[144] Zhang, M. and R. Sekar (2013) "Control Flow Integrity for COTS Binaries," in Proceedings of the 22nd USENIX Security Symposium (USENIX Security).

[145] Zhang, C., T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou (2013) "Practical control flow integrity and randomization for binary executables," in Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P).

[146] Prakash, A. and H. Yin (2015) "Defeating ROP through denial of stack pivot," in Proceedings of the 31st Annual Computer Security Applications Conference, pp. 111–120.

[147] Roemer, R., E. Buchanan, H. Shacham, and S. Savage (2012) "Return-oriented programming: Systems, languages, and applications," ACM Transactions on Information and System Security (TISSEC), **15**(1), p. 2.

[148] Schwartz, E. J., T. Avgerinos, and D. Brumley (2011) "Q: Exploit Hardening Made Easy." in Proceedings of the 20th USENIX Security Symposium (USENIX Security).

[149] Shoshitaishvili, Y., A. Bianchi, K. Borgolte, A. Cama, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, et al. (2018) "Mechanical Phish: Resilient Autonomous Hacking," IEEE Security & Privacy, **16**(2), pp. 12–22.

[150] Snow, K. Z., F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi (2013) "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in Proceedings of the 34 IEEE Symposium on Security and Privacy (S&P).

[151] Follner, A., A. Bartel, H. Peng, Y.-C. Chang, K. Ispoglou, M. Payer, and E. Bodden (2016) "PSHAPE: Automatically Combining Gadgets for Arbitrary Method Execution," in International Workshop on Security and Trust Management.

[152] Salwan, J. (2012), "ROPgadget," https://github.com/JonathanSalwan/ROPgadget.

[153] Göktas, E., E. Athanasopoulos, H. Bos, and G. Portokalidis (2014) "Out of Control: Overcoming Control-Flow Integrity," in Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P).

[154] Carlini, N., A. Barresi, M. Payer, D. Wagner, and T. R. Gross (2015) "Control-flow bending: On the effectiveness of control-flow integrity," in Proceedings of the 24th USENIX Security Symposium (USENIX Security).

[155] Schuster, F., T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz (2015) "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in In Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P).

[156] Evans, I., F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.

[157] Hu, H., Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang (2015) "Automatic Generation of Data-Oriented Exploits." in Proceedings of the 24nd USENIX Security Symposium (USENIX Security).

[158] Hu, H., S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang (2016) "Data-oriented programming: On the expressiveness of non-control data attacks," in Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P).

[159] Lexfo (2018), "CVE-2017-11176: A step-by-step Linux Kernel exploitation," https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part4.html#stack-pivoting.

[160] Team, P. (2015), "RAP: RIP ROP," https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf.

[161] Ge, X., N. Talele, M. Payer, and T. Jaeger (2016) "Fine-grained control-flow integrity for kernel software," in Proceedings of 2016 IEEE European Symposium on Security and Privacy (Euro S&P).

[162] Davi, L., D. Gens, C. Liebchen, and A.-R. Sadeghi (2017) "PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables." in Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS).

[163] Song, C., B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee (2016) "Enforcing Kernel Security Invariants with Data Flow Integrity." in Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS).

[164] dong-hoon you (2016), "New Reliable Android Kernel Root Exploitation Techniques," http://powerofcommunity.net/poc2016/x82.pdf.

[165] Van Bulck, J., M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx (2018) "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in Proceedings of the 27th USENIX Security Symposium (USENIX Security).

[166] spender (2017), "wait for kaslr to be effective," https://grsecurity.net/~spender/exploits/wait_for_kaslr_to_be_effective.c.

[167] National Vulnerability Database (2017), "CVE-2017-8890 Detail," https://nvd.nist.gov/vuln/detail/CVE-2017-8890.

[168] Cook, K. (2016), "x86/mm: Always enable CONFIG_DEBUG_RODATA and remove the Kconfig option," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9ccaf77cf05915f51231d158abfd5448aedde758.

[169] Baldoni, R., E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi (2018) "A Survey of Symbolic Execution Techniques," ACM Comput. Surv., **51**(3).

[170] david942j (2018), "The one-gadget in glibc," https://david942j.blogspot.com/2017/02/project-one-gadget-in-glibc.html.

[171] Eagle, C. (2011) The IDA Pro Book (Second edition), no starch press.

[172] (2018), "Linux kernel Vulnerability Statistics," https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[173] Ge, X., W. Cui, and T. Jaeger (2017) "Griffin: Guarding control flows using intel processor trace," ACM SIGOPS Operating Systems Review, **51**(2).

[174] Lin, Z. (2021), "How AUTOSLAB Changes the Memory Unsafety Gamel," https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game.

[175] Wickman, B., H. Hu, I. Yun, D. Jang, J. Lim, S. Kashyap, and T. Kim (2021) "Preventing Use-After-Free Attacks with Fast Forward Allocation," in Proceedings of the 30th USENIX Security Symposium (USENIX 2021), Vancouver, B.C., Canada.

[176] McCanne, S. and V. Jacobson (1993) "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in Proceedings of the USENIX Winter 1993 Technical Conference.

[177] syzbot (2020), "KASAN: use-after-free Read in free_netdev," https://syzkaller.appspot.com/bug?extid=aeb990c485fd48663c5a.

[178] ——— (2020), "BUG: corrupted list in kobject_add_internal," https://syzkaller.appspot.com/bug?id=f0ec9a394925aafbdf13d0a7e6af4cff860f0ed6.

[179] ——— (2020), "BUG: corrupted list in nft_obj_del," https://syzkaller.appspot.com/bug?id=ba1aecb13c60bd02d7370dd3c7c2ff4777f61407.

[180] ——— (2020), "BUG: corrupted list in nf_tables_commit," https://syzkaller.appspot.com/bug?extid=37a6804945a3a13b1572.

[181] ——— (2020), "general protection fault in vb2_mmap," https://syzkaller.appspot.com/bug?id=4cf5ee79b52a4797c5bd40a58bd6ab243d40de48.

[182] ——— (2020), "KASAN: use-after-free Read in vb2_mmap," https://syzkaller.appspot.com/bug?extid=be93025dd45dccd8923c.

[183] ——— (2020), "INFO: task hung in vivid_stop_generating_vid_cap," https://syzkaller.appspot.com/bug?id=4c0ccb254972cc51bdf6838cb1eff4fcc00de597.

[184] ——— (2020), "KASAN: use-after-free Read in ___vb2_perform_fileio," https://syzkaller.appspot.com/bug?id=1be0e67cda1ee2828c4cdbfd089e8ed9fc349e53.

[185] ——— (2020), "KASAN: slab-out-of-bounds Read in bitmap_ip_add," https://syzkaller.appspot.com/bug?id=3a6c9972ff471c4dbc3f45e83dd5fa2f18cb82a4.

[186] ——— (2020), "KASAN: slab-out-of-bounds Read in bitmap_ip_ext_cleanup," https://syzkaller.appspot.com/bug?extid=6491ea8f6dddbf04930e.

[187] ——— (2020), "KASAN: use-after-free Read in p9_fd_poll," https://syzkaller.appspot.com/bug?id=1b726e0a253ee75e902d090f68705da3d42d6ae0.

[188] ———, "KASAN: slab-out-of-bounds Read in bitmap_ip_add," https://syzkaller.appspot.com/bug?id=001410d947fa7742a85647e596c45661118c7d24, year=2020.

[189] ——— (2020), "KASAN: null-ptr-deref Read in refcount_sub_and_test_checked (2)," https://syzkaller.appspot.com/bug?id=17535f4bf5b322437f7c639b59161ce343fc55a9.

[190] ——— (2020), "KASAN: slab-out-of-bounds Read in tcf_exts_destroy," https://syzkaller.appspot.com/bug?id=2389bfc4b1c4ea3969629ed19bef0b3b2ec741f2.

[191] ——— (2020), "KASAN: slab-out-of-bounds Read in vcs_scr_readw," https://syzkaller.appspot.com/bug?id=a152f63add75a07ddbd234667e25286d8da14a34.

[192] ——— (2020), "KASAN: use-after-free Read in cma_bind_port," https://syzkaller.appspot.com/bug?id=3b7409f639067d927b8ad1b11a5e08bae27061af.

[193] ——— (2020), "KASAN: use-after-free Read in cma_bind_port," https://syzkaller.appspot.com/bug?extid=da2591e115d57a9cbb8b.

[194] ——— (2020), "KASAN: use-after-free Read in ___nf_tables_abort," https://syzkaller.appspot.com/bug?id=c897760daaa146a37be25eb5005660a110b3aa6e.

[195] ——— (2020), "BUG: corrupted list in ___nf_tables_abort," https://syzkaller.appspot.com/bug?extid=437bf61d165c87bd40fb.

[196] ——— (2020), "KASAN: use-after-free Read in bitmap_port_ext_cleanup," https://syzkaller.appspot.com/bug?id=d463fb9bf88946c8b83d95b2f216f295b50c70de.

[197] ——— (2020), "KASAN: use-after-free Read in rdma_listen," https://syzkaller.appspot.com/bug?id=b8febdb3c7c8c1f1b606fb903cee66b21b2fd02f.

[198] ——— (2020), "KASAN: use-after-free Read in sctp_auth_free," https://syzkaller.appspot.com/bug?id=cbb289816e728f56a4e2c1b854a3163402fe2f88.

[199] ——— (2020), "KASAN: use-after-free Read in tcp_fastretrans_alert," https://syzkaller.appspot.com/bug?id=7be8b464a3a27e6dc5c73d3ffe3b56dc0cf51e52.

[200] ——— (2020), "KASAN: use-after-free Write in ___sco_sock_close," https://syzkaller.appspot.com/bug?id=57e98513afbe427bbd65ac295130bcf5bc860dd8.

[201] ——— (2020), "WARNING: refcount bug in l2cap_chan_put," https://syzkaller.appspot.com/bug?id=39d35c93d0856ca3134bf97f8bb3f249808c2751.

[202] ——— (2020), "WARNING: ODEBUG bug in tcf_queue_work," https://syzkaller.appspot.com/bug?id=6a039858238a38cbc7f372607fc5d49f4469cf2c.

[203] ——— (2020), "KASAN: use-after-free Read in route4_get," https://syzkaller.appspot.com/bug?id=5bb09c0c5b65ab2ce628ba26fe7cbd06144bd952.

[204] ——— (2020), "KASAN: slab-out-of-bounds Write in sha512_final," https://syzkaller.appspot.com/bug?id=e4be30826c1b7777d69a9e3e20bc7b708ee8f82c.

[205] ——— (2020), "KMSAN: uninit-value in geneve_xmit," https://syzkaller.appspot.com/bug?id=5a8b026b9858500883281937188077 2d34a5bd44.

[206] ——— (2020), "UBSAN: array-index-out-of-bounds in dquot_resume," https://syzkaller.appspot.com/bug?id=a6b85665ee98a38adff4459261702f3c3a6bf6e9.

[207] ——— (2020), "UBSAN: shift-out-of-bounds in dummy_hub_control," https://syzkaller.appspot.com/bug?id=b5b251b9bcc4653c39164dfee969dafb903ae25e.

[208] Ren, X. J., K. Rodrigues, L. Chen, C. Vega, M. Stumm, and D. Yuan (2019) "An Analysis of Performance Evolution of Linux's Core Operations," in Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)).

[209] Park, S.-O., O. Kwon, Y. Kim, S. K. Cha, and H. Yoon (2021) "Mind control attack: Undermining deep learning with GPU memory exploitation," Computers Security, **102**, p. 102115.

[210] Francillon, A. and C. Castelluccia (2008) "Code injection attacks on harvard-architecture devices," in Proceedings of the 2008 ACM SIGSAC Conference on Computer and Communications Security (CCS).

Vita

Yueqi Chen

Education

The Pennsylvania State University — State College, PA
Ph.D. in Information Sciences and Technology — Aug. 2017 - Aug. 2022 (Expected)
Advisor: Dr. Xinyu Xing & Dr. Peng Liu

Nanjing University — Nanjing, China
B.Sc. in Computer Science and Technology — Aug. 2013 - Jul. 2017

Honors and Awards

DEFCON CTF, 7th — 2021

IBM Ph.D. Fellowship — 2020

DEFCON CTF, 16th — 2018

NSA codebreaker, 5th — 2017

Travel Grant from BlackHat, USENIX Security, IEEE S&P, IST — 2017 - 2022

Industrial Experience

Research Intern, IBM Watson Research Center — Yorktown, NY
Mentors: Dr. Michael Le, Dr. Dan Williams — May 2021 - Jul. 2021

Research Intern, Baidu X-Lab — Sunnyvale, CA
Mentor: Dr. Peng Li — May 2019 - Aug. 2019

Research Intern, JD R&D Center — Mountain View, CA
Mentor: Dr. Yueh-Hsun Lin — May 2018 - Aug. 2018