From Thousands of Hours to a Couple of Minutes: Towards Automating Exploit Generation for Arbitrary Types of Kernel Vulnerabilities

Wei Wu, Xinyu Xing, Jimmy Su

Abstract

Software vendors usually prioritize their bug remediation based on ease of their exploitation. However, accurately determining exploitability typically takes tremendous hours and requires significant manual efforts. To address this issue, automated exploit generation techniques can be adopted. In practice, they however exhibit an insufficient ability to evaluate exploitability particularly for kernel vulnerabilies (*e.g.,* use after free, heap overflow, double free). This is mainly because of the complexity of kernel exploitation as well as the scalability of an OS kernel.

In this paper, we therefore propose `FUZE`, a new framework to facilitate the process of kernel exploitation. 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 expedite exploitability evaluation. Technically, `FUZE` utilizes kernel fuzzing along with symbolic execution to identify, analyze and evaluate the system calls valuable and useful for kernel exploitation.

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 real-world kernel UAF vulnerabilities on Linux systems, we then demonstrate `FUZE` could not only escalate kernel exploitability and but also diversify working exploits. In addition, we show that `FUZE` could facilitate security mitigation bypassing, making exploitability evaluation less labor-intensive and more efficient.

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 [13] and Ubuntu [28] 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 memory corruption vulnerabilities residing in OS kernels. Based on the root cause of the vulnerability, there are different exploitation technique handling various kinds of vulnerabilities.

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.,* [5, 7, 8]). However, the techniques proposed are insufficient for the problem above. On the one hand, this is because their technical approaches explore exploitability only in the context of a crashing process whereas generating an exploit for a kernel vulnerability typically needs to vary the context of a kernel panic. On the other 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.

In this work, we propose `FUZE`, an exploitation framework to evaluate the exploitability of kernel vulnerabilities. Technically speaking, our framework utilizes a fuzzing technique to diversify the contexts of a kernel panic and then leverages symbolic execution to explore exploitability under different contexts. To showcase the effectiveness, we demonstrate our technique through kernel UAF vulnerabilities. But, it should be noted that the proposed technique can be also applied for other type of kernel vulnerabilities.

Use-After-Free vulnerabilities [24] 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 [6] and critical data leakage [17]. 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 select system calls and corresponding parameters based on the goal of his exploitation and the mitigation deployed in the OS kernel (*e.g.,* SMEP [18, 29] and SMAP [31]). We showcase this process through a concrete example in Section 2.2.

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 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 3, we will describe more details about this exploitation framework.

Different from the existing techniques (*e.g.,* [5, 7, 8]), 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 guides a security analyst to craft 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 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 vulnerabilities.

In summary, this paper makes the following contributions.

```c
void *task1(void *unused) {
  ...
  int err = setsockopt(fd, 0x107, 18,
        ↪ ..., ...);
}

void *task2(void *unused) {
  int err = bind(fd, &addr, ...);
}

void loop_race() {
  ...
  while(1) {
    fd = socket(AF_PACKET, SOCK_RAW,
        ↪ htons(ETH_P_ALL));
    ...
    //create two racing threads
    pthread_create (&thread1, NULL,
        ↪ task1, NULL);
    pthread_create (&thread2, NULL,
        ↪ task2, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    close(fd);
  }
}
```

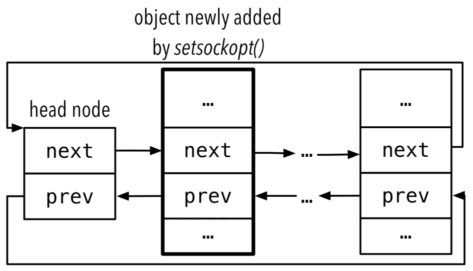
Table 1: A PoC code fragment pertaining to the kernel UAF vulnerability (CVE-2017-15649).

- We designed FUZE, an exploitation framework that utilizes kernel fuzzing along with symbolic execution to facilitate kernel 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.

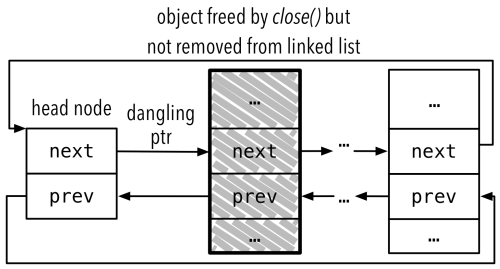
The rest of this paper is organized as follows. Section 2 describes the background and challenge of our research. Section 3 presents the overview of FUZE. Section 4 describes the design of FUZE in detail. Section 5 describes the implementation of FUZE, followed by Section 6 demonstrating the utility of FUZE. Section 7 summarizes the work most relevant to ours. Section 8 discusses how to apply the proposed techniques to other kinds of kernel vulnerabilities. Finally, we conclude this work in Section 9.

2 Background and Challenge

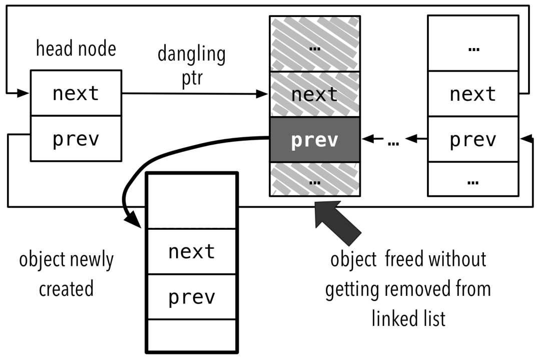
To craft an exploit for a UAF vulnerability residing in an OS kernel, a security analyst needs to first identify a system call to perform a heap spray to a freed object. Then, he needs to find a system call to dereference the



(a) Inserting a new object to doubly linked list.



(b) Triggering a free operation with a dangling pointer left behind.



(c) Writing unmanageable data to a memory chunk freed previously.

Figure 1: Demonstrating a kernel panic triggered through a real-world kernel Use-After-Free vulnerability indicated by `CVE-2017-15649`.

data sprayed and thus influence the execution of the OS kernel, *i. e.,* redirecting the control flow of the system to unauthorized operations, such as privilege escalation or critical data leakage.

In the past, research has focused on how to perform an effective heap spray so that one could take full control over the content in the freed object (e.g., [35]). This significantly facilitates the process of crafting exploits. However, in practice, it is still challenging and labor-intensive for a security analyst to craft a working exploit for a kernel UAF vulnerability. To specify the challenges and the manual efforts involved for exploitation, we use a real-world example to go through the whole process of exploit generation. In the following, we first briefly introduce a kernel UAF vulnerability. Then, we elaborate the challenges and manual efforts involved in exploiting this vulnerability.

2.1 Kernel Use-After-Free Vulnerability

Table 1 shows a `C` program, 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 1a).

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 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 1b).

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 1c, 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 Challenges of Exploit Generation

To craft an exploit for the vulnerability above, an intuitive approach is to turn the aforementioned kernel panic into a working exploit. To be specific, a security analyst could explore whether he could leverage the unexpected write operation to carry out a control flow hijacking or manipulate critical data in the Linux kernel so that it could fulfill a privilege escalation.

By taking a close look at the unexpected write primitive, we could observe that this write operation could provide us with an ability to write the address of a new object to a kernel address (indicated by the dark-gray box in Figure 1c).

Given that one could leverage a heap spray to take over the region of the freed object and thus obtain the full control over the content of the freed object, we could conclude that the write primitive provides us with the ability to write an *unmanageable* data (*i. e.,* the address of the new object) to a *fixed* address in Linux kernel.

It is clear that this primitive barely provides us with the ability to hijack a control flow or fulfills a privilege escalation. As a result, it is almost infeasible for the example above to generate an exploit by simply following the context of the kernel panic.

To address this issue, an alternative approach is to force Linux kernel to panic in different running contexts, and then use various kernel panics to explore primitives useful for UAF exploitation. To be more specific, a security analyst could first explore all the system calls and identify those that can also dereference the data in the freed object. For each system call identified, he could then insert it into the site right behind `line 22`. Since the site indicates a moment when control flow transfers from one iteration to another, the new system call could perform dereference prior to the system call already present in the PoC program, and let Linux kernel panic in different contexts. For example, using system call `sendmsg()` with appropriate arguments right behind `line 22`, the kernel would experience a panic before the PoC runs into a next iteration.

Since a new kernel panic indicates a different context, it could provide a security analyst with the opportunity to find a different primitive useful for exploitation. Continue this example. After the invocation of `sendmsg()` ↪ , the Linux kernel would panic and he could identify a primitive that allows him to perform an arbitrary write to register `rip`. Since `rip` is an instruction register, the security analyst could potentially conduct an ROP attack, disable mitigation SMEP/SMAP, and eventually fulfill a control flow hijacking.

In comparison with the intuitive approach mentioned above, this approach with context variation could significantly improve the possibility of performing a successful exploitation. In practice, it is however less effective for the following reasons.

First, to explore system calls, a security analyst has to conduct a manual search on the kernel code. Given its scalability, this search could take significant amount of time and require extensive manual efforts. Second, after pinpointing a system call, the security analyst needs to further figure out the parameters for the system call to reach to the site where it dereferences the freed object. Given the complexity of kernel code, this generally requires profound knowledge in Linux kernel. Third, since Linux kernel typically enables mitigation mechanisms, the security analyst also needs to examine whether the system call identified could provide him with the capability of bypassing mitigation, and if not he also needs to revisit kernel code to track down alternative system calls. As a result, this again involves manual efforts and domain expertise. In the following sections, we will design and develop a new technical approach to address all these issues and thus facilitate the ability of crafting working exploits.

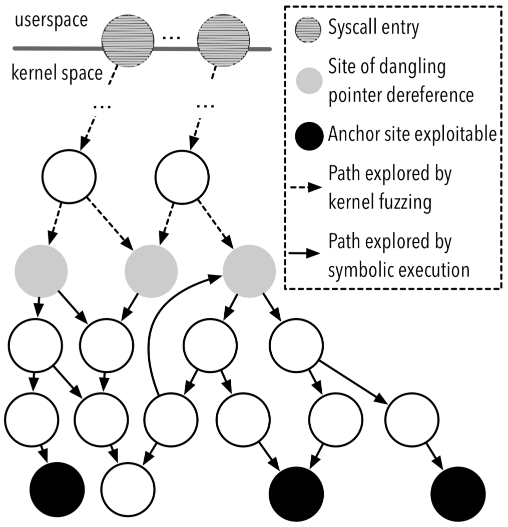

userspace
kernel space
Syscall entry
Site of dangling pointer dereference
Anchor site exploitable
Path explored by kernel fuzzing
Path explored by symbolic execution

Figure 2: An overview of FUZE in which kernel fuzzing explores system calls that dereferences dangling pointers, followed by symbolic execution that tracks down anchor sites truly useful for UAF exploitation.

3 Overview

In this section, we propose FUZE, a new framework to expedite the process of exploit generation. In the following, we discuss the design principle of FUZE as well as the high level idea of this exploitation framework.

3.1 Design Principle

As is showcased in the motivating example above, crafting exploits for a kernel UAF vulnerability involves many steps, most of which need significant manual efforts and domain expertise. This significantly increases the difficulty in evaluating exploitability for kernel UAF vulnerabilities. Therefore, we decide to design FUZE to facilitate exploit crafting from the following four aspects.

First, we design FUZE with the ability to monitor heap allocation so that it could automatically pinpoint the time frame when a vulnerable object occurs. Second, we augment FUZE with an automated mechanism which could help identify the system calls that one could potentially leverage for UAF exploitation. Third, we extend FUZE to have the ability to automatically construct the data that needs to be sprayed to the region pertaining to the vulnerable object. Considering exploitability also implies the ability to bypass built-in security mitigation, last but not least, we also augment FUZE with the ability to explore the different ways to exploit UAF vulnerabilities.

3.2 Technical Approach

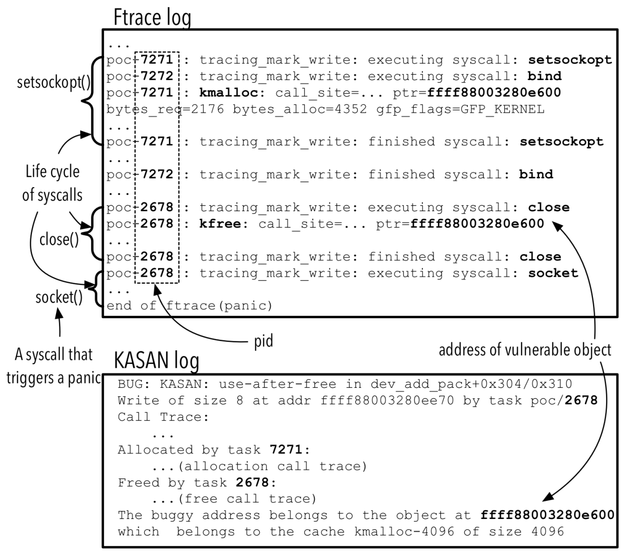
At the high level, FUZE works as follows. Similar to the manual analysis of a kernel UAF vulnerability, FUZE takes as input a PoC program, which does not cause the success of exploitation but a kernel panic. Then, it automatically performs a vulnerability analysis, through which FUZE triages the type of the vulnerabilities, discards those not in the types of Use-After-Free, and extracts the information critical for vulnerability exploitation. As we will discuss in Section 4.1, this information typically contains the size of the vulnerable object, the site where it occurs and the site where it starts to be dereferenced, etc.

As is showcased in Section 2.2, by introducing a new system call pertaining to the vulnerable object right after the occurrence of a dangling pointer, we could let an OS kernel panic at a different context and thus obtain other primitives potentially useful for vulnerability exploitation. Therefore, FUZE also tracks down system calls pertaining to the vulnerable objects. To do this, FUZE utilizes a dynamic analysis scheme to identify system calls attached to a vulnerable object. Then, it further explores how to use the system calls identified to reach to the sites potentially useful for UAF exploitation.

Ideally, we could design FUZE to perform our dynamic analysis – starting from the entry point of a system call identified – by using symbolic execution because this would allow us to identify all the paths that could lead to the anchor sites. However, given the scalability and complexity of a kernel code base, intuition suggests this approach might inevitably incur path explosion without reaching to any sites useful for exploitation, even though we limit the scope of the symbolic execution within a single system call.

To address this issue, FUZE leverages kernel fuzzing along with symbolic execution. In particular, FUZE first uses kernel fuzzing to explore system calls that could trigger dangling pointer dereference. Starting from the site of dangling pointer dereference, it then leverages symbolic execution to track down the sites potentially useful for UAF exploitation. We illustrate this practice in Figure 2.

The intuition behind the design above is as follows. Kernel fuzzing is a technique that allows us to explore paths within a system call by mutating its arguments and shuffling the combination of other system calls. Used as a dynamic analysis scheme, it could provide FUZE with the ability to explore a code base without the concern of path explosion. Different from symbolic execution, kernel fuzzing however cannot explore the primitives hidden in the deep layer of an OS kernel, and thus naturally limits the capability of tracking down sites potentially useful for exploitation. With the combination of both kernel fuzzing and symbolic execution, FUZE could take advantage of the scalability benefit provided by kernel fuzzing and at the same time avoid its explorability weakness.



```
...
poc-7271 : tracing_mark_write: executing syscall: setsockopt
poc-7272 : tracing_mark_write: executing syscall: bind
poc-7271 : kmalloc: call_site=... ptr=ffff88003280e600
bytes_req=2176 bytes_alloc=4352 gfp_flags=GFP_KERNEL
...
poc-7271 : tracing_mark_write: finished syscall: setsockopt
...
poc-7272 : tracing_mark_write: finished syscall: bind
...
poc-2678 : tracing_mark_write: executing syscall: close
poc-2678 : kfree: call_site=... ptr=ffff88003280e600
...
poc-2678 : tracing_mark_write: finished syscall: close
poc-2678 : tracing_mark_write: executing syscall: socket
...
end of ftrace(panic)
```

```
BUG: KASAN: use-after-free in dev_add_pack+0x304/0x310
Write of size 8 at addr ffff88003280ee70 by task poc/2678
Call Trace:
    ...
Allocated by task 7271:
    ...(allocation call trace)
Freed by task 2678:
    ...(free call trace)
The buggy address belongs to the object at ffff88003280e600
which  belongs to the cache kmalloc-4096 of size 4096
```

Figure 3: A kernel log trace obtained through instrumentation and dynamic analysis.

Recall that the ultimate goal of this work is to facilitate exploit generation and thus exploitability evaluation. As we will discuss in Section 4.3, the sites identified through the aforementioned approach however may not reflect the ones that could truly facilitate exploitation. To address this issue, FUZE further summarizes a set of exploitation approaches commonly adopted. Under the guidance of these approaches, it then evaluates each of the sites identified and pinpoints anchor sites truly beneficial for exploitation facilitation. In the following two sections, we describe and discuss the design details of FUZE and illustrate its utility using real world examples.

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 and then perform kernel fuzzing. Third, we specify how FUZE performs symbolic execution and pinpoints anchor sites truly useful for exploitation. Finally, we discuss some limitations and other technical details.

4.1 Critical Information Extraction

Prior to the kernel fuzzing and symbolic execution, FUZE first takes as input a PoC program. Then it extracts information needed for vulnerability exploitation. To do this, we utilize an off-the-shelf kernel address sanitizer KASAN [19] along with a dynamic tracing mechanism.

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.

Dynamic Tracing. As is discussed in Section 2.2, kernel UAF exploitation could be achieved by following or varying a context pertaining to a kernel panic. To do that, we need to not only perform analysis to the system calls invoked by the PoC program but also trace the kernel execution pertaining to these calls. In this work, we design a dynamic tracing mechanism to facilitate the ability of performing such analysis and kernel tracing.

First, we 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.

Critical Information Extraction. To obtain the information needed for exploitation, we utilize both KASAN log and the trace obtained from dynamic tracing. To illustrate this, we take for example the kernel trace shown in Figure 3. 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 determine system call close() 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.

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.

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

(a) 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;
  }
}
```

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

Table 2: The wrapping functions preventing dangling pointer dereference.

4.2.1 Fuzzing Context Initialization

As is mentioned in Section 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 2a, 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 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 2b).

`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[1]. 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.

[1]At the fuzzing stage, our objective is to identify system calls for diversifying running contexts but not directly for generating exploitation. Therefore, we disable kernel address randomization for reducing the complexity of tracking down dangling pointers.

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

Table 3: The pseudo-code indicating the way of performing concurrent kernel fuzz testing.

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 framework, 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 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 6, this approach however does not jeopardize our capability in finding system calls useful for exploitation.

4.3 Symbolic Execution

As is mentioned in Section 3, we perform symbolic execution with the goal of identifying sites potentially useful for vulnerability exploitation. In this section, we first describe how to set up a running context for symbolic execution. Then, we discuss how to identify the anchor sites truly useful for exploitation by using symbolic execution.

4.3.1 Symbolic Execution Context Initialization

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 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 context for consecutive symbolic execution.

4.3.2 Anchor Site Identification

Starting from the initial context, we create symbolic values for each byte of the freed object. Then, we symbolically resume kernel execution and explore anchor sites potentially useful for vulnerability exploration. To identify anchor sites, we define a set of primitives indicating the operations needed for exploitation. Then, we look up these primitives and take them as candidate anchor sites 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 anchor sites. 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.,* anchor sites), we evaluate primitives as follows.

As is specified in [26], 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 execution 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 [4] 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 magic 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 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 magic 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.,* [12, 16]) and the facilitation of information leak provided by `FUZE` is neither a necessary nor a sufficient condition for successful exploitation.

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 anchor site 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 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 diversifies fuzzing contexts based on the practice of how a PoC program performs object free and dereference (see the two different approaches in Table 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 anchor site 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 6, FUZE still provides us with the facilitation for UAF exploitation even if there is only one context for exploration.

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 1), we insert function `sleep()` at the end of each iteration.

Kernel fuzzing. As is described in Section 4.2, we need to identify candidate system calls potentially useful for exploitation using kernel fuzzing. To do this, we can utilize `syzkaller` [2], 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 (see Appendix).

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 performing 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` [1], 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.

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.

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

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 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 5: The Efficiency of fuzzing and symbolic execution.

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 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 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.

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 anchor sites we identified. To be specific, we computed the data that needs to be sprayed based on the constraints tied to the anchor sites. Then, we performed the heap spray with three different system calls – add_key(), msgsnd() ↪ , sendmsg() – by following the techniques introduced in [35]. To fulfill exploitation using the anchor sites identified, we eventually redirect the execution to an ROP chain [26] commonly used for exploitation. We release the generated exploit at [3].

6.2 Effectiveness

Table 4 specifies the amount of distinct exploits publicly available for each kernel UAF vulnerability as well as their capability in 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 4.

With regard to the ability to perform exploitation and bypass SMEP illustrated in Table 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 anchor sites useful for exploitation. It should be noticed that we count the amount of distinct exploits shown in Table 4 based on the number of contexts capable of facilitating exploitation but not the anchor sites 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 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 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 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 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 anchor sites 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 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 [10] 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.

6.3 Efficiency

Table 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 anchor site. 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 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 and extends their amount to 1,203. As is mentioned in Section 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 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 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 anchor sites within a maximum depth of 200 basic blocks, we could not only ensure the identification of anchor sites but also reduce the risk of experiencing path explosion.

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 [27]) 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.* [15] demonstrate a timing side channel attack that infers kernel memory layout by exploiting the memory management system; Evtyushkin *et al.* [11] 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.* [12] infer kernel address information by exploiting prefetch instructions; Lipp *et al.* [22] 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 [18, 26] 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 [20]. 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.* [35] 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.* [23] 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.* [7] 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.* [14] 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 [5] and [9], Brumley *et al.* explore automatic exploit generation for stack overflow and format string vulnerabilities using preconditioned symbolic execution and concolic execution, respectively. In [25], Mothe *et al.* utilize forward and backward taint analysis to craft working exploits for simple vulnerabilities in user-mode applications. In [30], Repel *et al.* make use of symbolic execution to generate exploits for heap overflow vulnerabilities residing in user-mode applications. In [32–34], 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.

8 Discussion

In this section, we discuss how we apply the aforementioned technique for the exploitation of other vulnerability types.

8.1 Exploitation knowledge accumulation

Knowledge about dynamic kernel objects boosts exploitation of various different kinds of vulnerabilities. To make the best use of some exploit primitives (such as write-what-where and increase the probability of hijacking sensitive control flow and data flow objects) we need the exploitation knowledge of kernel objects. There are two categories of exploitation knowledge. The first category of exploitation knowledge includes the sizes and sensitive fields of an kernel object, which can be extracted through compiler-assisted static analysis. The second category of exploitation knowledge is concrete programs which allocate and dereference corresponding kernel objects.

8.2 Heap overflow exploitation

Heap overflow exploitation is similar to use-after-free exploitation in the following aspects. First, a victim object can be controlled by an attack by heap layout manipulation. Second, the attacker needs to dereference the corrupted data to gain exploitable primitives.

Starting from a PoC that demonstrates a kernel heap out-of-bounds write. A heap overflow exploitation is divided into three steps. First, an attacker needs to understand the power of the overflow, (e.g., how many bytes he can write out-of-bounds and whether he could obtain the control over its content). Second, the attacker needs to select a suitable victim object which contains sensitive fields (*e.g.,* function pointer), and then place it right behind the vulnerable object. Third, the attacker needs to figure out a way to dereference the corrupted victim object and the required overflown bytes to perform exploitation.

We can apply symbolic analysis to understand a heap overflow vulnerability. More specifically, we can take kernel snapshot at the entry of the system call that triggers the overflow. Using QEMU, we then record the control flow trace starting from the entry of the system call until the overflow site. By replaying that trace with symbolic execution engine, we can obtain the path constraints over parameters. In this process, we set parameters which is under the attackers' control as symbolic values and then force the symbolic execution to go through exactly the same path. When the symbolic executor reaches the overflow site, we can inspect the current state and try to exploit this write primitive.

We can apply kernel fuzzing to find allocations and dereferences of a target object. By setting a trap at the allocation and dereference site of a target object and fuzzing to reach these sites, we can build a dataset of system calls corresponding to allocation and dereference of target objects. After we identify the dereference of target objects, we can set content of the victim object as symbolic value and use symbolic execution to find potentially exploitable primitives.

9 Conclusion

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

10 Acknowledgement

We thank Yueqi Chen, Jun Xu, Xiaorui Gong, Wei Zou for their help and support to this work. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any affiliations.

References

[1] angr - a binary analysis framework, 2017. http://angr.io/index.html.

[2] Syzkaller - kernel fuzzer, 2017. https://github.com/google/syzkaller.

[3] Exploit release, 2018. https://github.com/ww9210/Linux_kernel_exploits.

[4] P. Argyroudis. The linux kernel memory allocators from an exploitation perspective, 2012. https://argp.github.io/2012/01/03/linux-kernel-heap-exploitation/.

[5] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Commun. ACM*, 57, 2014.

[6] B. Azad. Mac OS X privilege escalation via use-after-free: CVE-2016-1828, 2016. https://bazad.github.io/2016/05/mac-os-x-use-after-free/#use-after-free.

[7] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P)*, 2017.

[8] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P)*, 2008.

[9] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*, 2012.

[10] N. V. Database. CVE-2017-7374 detail, 2017. https://nvd.nist.gov/vuln/detail/CVE-2017-7374.

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

[12] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[13] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In *Proceedings of the 32th International Conference on Software Engineering (ICSE)*, 2010.

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

[15] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P)*, 2013.

[16] Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2010.

[17] jndok. Analysis and exploitation of pegasus kernel vulnerabilities, 2016. https://jndok.github.io/2016/10/04/pegasus-writeup/.

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

[19] KASAN. The kernel address sanitizer(kasan), 2017. https://github.com/google/kasan/wiki.

[20] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23th Conference on USENIX Security Symposium (USENIX Security)*, 2014.

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

[22] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. In *arXiv preprint arXiv:1801.01207*, 2018.

[23] K. Lu, M. Walter, D. Pfaff, and S. Nürnberger and Wenke Lee and Michael Backes. 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)*, 2017.

[24] Mitre. CWE-416: Use after free, 2018. https://cwe.mitre.org/data/definitions/416.html.

[25] R. Mothe and R. R. Branco. Dptrace: Dual purpose trace for exploitability analysis of program crashes. In *Blackhat USA*, 2016.

[26] V. Nikolenko. Linux kernel ROP - ropping your way to # (part 1), 2016. https://www.trustwave.com/Resources/SpiderLabs-Blog/Linux-Kernel-ROP---Ropping-your-way-to---(Part-1)/.

[27] PaX/Grsecurity. Pax/grsecurity –> kspp –> aosp kernel: Linux kernel mitigation checklist, 2017. https://github.com/hardenedlinux/grsecurity-101-tutorials/blob/master/kernel_mitigation.md.

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

[29] A. Popov. CVE-2017-2636: exploit the race condition in the n_hdlc linux kernel driver bypassing SMEP, 2017.

[30] D. Repel, J. Kinder, and L. Cavallaro. Modular synthesis of heap exploits. In *ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS)*, 2017.

[31] C. Salls. Exploiting CVE-2017-5123 with full protections. SMEP, SMAP, and the Chrome Sandbox, 2017.

[32] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*, 2015.

[33] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 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)*, 2016.

[34] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2016.

[35] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu. 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)*, 2015.

Appendix

A Extended System Calls in Syzkaller

```
dccp_level_option = SOL_SOCKET,
    ↪ SOL_DCCP
getsockopt$inet_dccp_int(fd sock_dccp,
    ↪ level flags[dccp_level_option],
    ↪ optname flags[
    ↪ dccp_option_types_int], optval
    ↪ ptr[out, int32], optlen ptr[inout
    ↪ , len[optval, int32]])
setsockopt$Inet_dccp_int(fd sock_dccp,
    ↪ level flags[dccp_level_option],
    ↪ optname flags[
    ↪ dccp_option_types_int], optval
    ↪ ptr[in, int32], optlen len[optval
    ↪ ])
getsockopt$inet6_dccp_int(fd sock_dccp6,
    ↪  level flags[dccp_level_option],
    ↪ optname flags[
    ↪ dccp_option_types_int], optval
    ↪ ptr[out, int32], optlen ptr[inout
    ↪ , len[optval, int32]])
setsockopt$Inet6_dccp_int(fd sock_dccp6,
    ↪  level flags[dccp_level_option],
    ↪ optname flags[
    ↪ dccp_option_types_int], optval
    ↪ ptr[in, int32], optlen len[optval
    ↪ ])
getsockopt$inet_dccp_buf(fd sock_dccp,
    ↪ level flags[dccp_level_option],
    ↪ optname flags[
    ↪ dccp_option_types_buf], optval
    ↪ ptr[out, int32], optlen ptr[inout
    ↪ , len[optval, int32]])
setsockopt$Inet_dccp_buf(fd sock_dccp,
    ↪ level flags[dccp_level_option],
    ↪ optname flags[
    ↪ dccp_option_types_buf], optval
    ↪ ptr[in, int32], optlen len[optval
    ↪ ])
getsockopt$inet6_dccp_buf(fd sock_dccp6,
    ↪  level flags[dccp_level_option],
    ↪ optname flags[
    ↪ dccp_option_types_buf], optval
    ↪ ptr[out, int32], optlen ptr[inout
    ↪ , len[optval, int32]])
setsockopt$Inet6_dccp_buf(fd sock_dccp6,
    ↪  level flags[dccp_level_option],
    ↪ optname flags[
    ↪ dccp_option_types_buf], optval
    ↪ ptr[in, int32], optlen len[optval
    ↪ ])
settimeofday(tv ptr[in, timeval], tz
    ↪ ptr[in, timezone])
gettimeofday(tv ptr[in, timeval], tz
    ↪ ptr[in, timezone])
timezone {
   tz_minuteswest int32
   tz_dsttime int32
}
resource sock_vsock_stream[sock_vsock]
socket$stream(domain const[AF_VSOCK],
    ↪ type const[SOCK_STREAM], proto
    ↪ const[0]) sock_vsock_stream
adjtimex(buf ptr[in, timex])
timex {
   modes int32
   offset int64
   freq int64
   maxerror int64
   esterror int64
   status int64
   constant int64
   precision int64
   tolerance int64
   time timeval
   tick int64
   ppsfreq int64
   jitter int64
   shift int32
   stabil int64
   jitcnt int64
   calcnt int64
   errcnt int64
   stbcnt int64
   tai int32
}
sethostname(name ptr[inout, string["foo
    ↪ "]], len const[3])
socket$key(domain const[AF_KEY], type
    ↪ const[SOCK_RAW], proto const[
    ↪ PF_KEY_V2]) sock
sendmsg$key(fd sock, msg ptr[in,
    ↪ send_msghdr_key], f flags[
    ↪ send_flags])
sendmmsg$key(fd sock, mmsg ptr[in,
    ↪ array[send_msghdr_key], vlen len[
    ↪ mmsg], f flags[send_flags]])
send_msghdr_key {
   msg_name ptr[in, sockaddr_storage,
       ↪ opt]
   msg_namelen len[msg_name, int32]
   msg_iov ptr[in, iovec_sadb_msg]
   msg_iovlen len[msg_iov, intptr]
   msg_control ptr[in, array[cmsghdr]]
   msg_controllen len[msg_control,
       ↪ intptr]
   msg_flags flags[send_flags, int32]
}
```