FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities

Wei Wu[⋆1,2,3], Yueqi Chen[1], Jun Xu[1], Xinyu Xing[1], Xiaorui Gong[†2], and Wei Zou[‡2]

[1]College of Information Sciences and Technology, Pennsylvania State University

[2]{CAS-KLONAT[†], BKLONSPT[‡]}, Institute of Information Engineering, Chinese Academy of Sciences
{wuw92, yxc431, jxx13, xxing}@ist.psu.edu, {gongxiaorui, zouwei}@iie.ac.cn

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 the kernel Use-After-Free (UAF) vulnerabilities. This is mainly because of the complexity of UAF 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 UAF 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 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 real-world 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.

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 Use-After-Free (UAF) vulnerabilities residing in OS kernels.

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 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.

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, 29]). 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 work was done while studying at Pennsylvania State University.

[†]Key Laboratory of Network Assessment Technology, CAS

[‡]Beijing Key Laboratory of Network Security and Protection Technology

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

Different from the existing techniques (*e.g.,* [5, 7, 8, 29]), 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 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.

```
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).

In summary, this paper 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.

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. Finally, we conclude this work in Section 8.

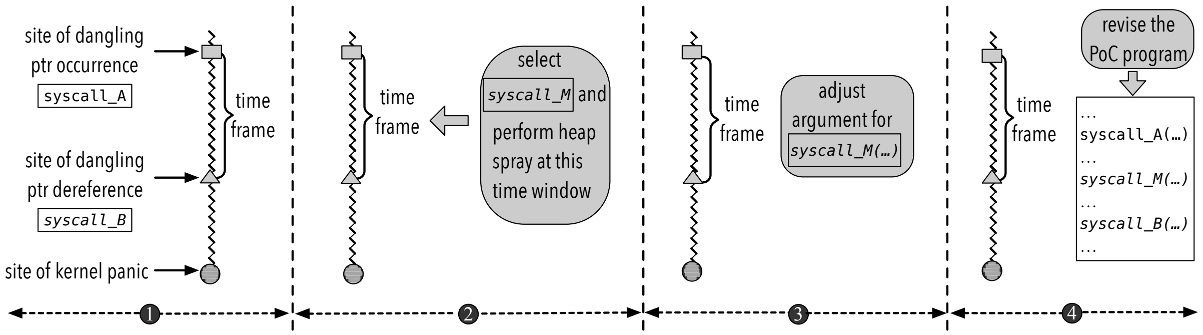

site of dangling ptr occurrence
syscall_A
time frame
site of dangling ptr dereference
syscall_B
site of kernel panic
time frame
select syscall_M and perform heap spray at this time window
time frame
adjust argument for syscall_M(…)
time frame
revise the PoC program
…
syscall_A(…)
…
syscall_M(…)
…
syscall_B(…)
…
❶
❷
❸
❹

Figure 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; ❸ adjusting the argument of the system call `syscall_M`; ❹ 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 occurence of the danlging pointer and its dereference respectively.

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 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 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 1.

In the past, research (e.g., [33]) 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 the step ❷ shown in Figure 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.

2.1 PoC Program for Kernel UAF Vulnerability

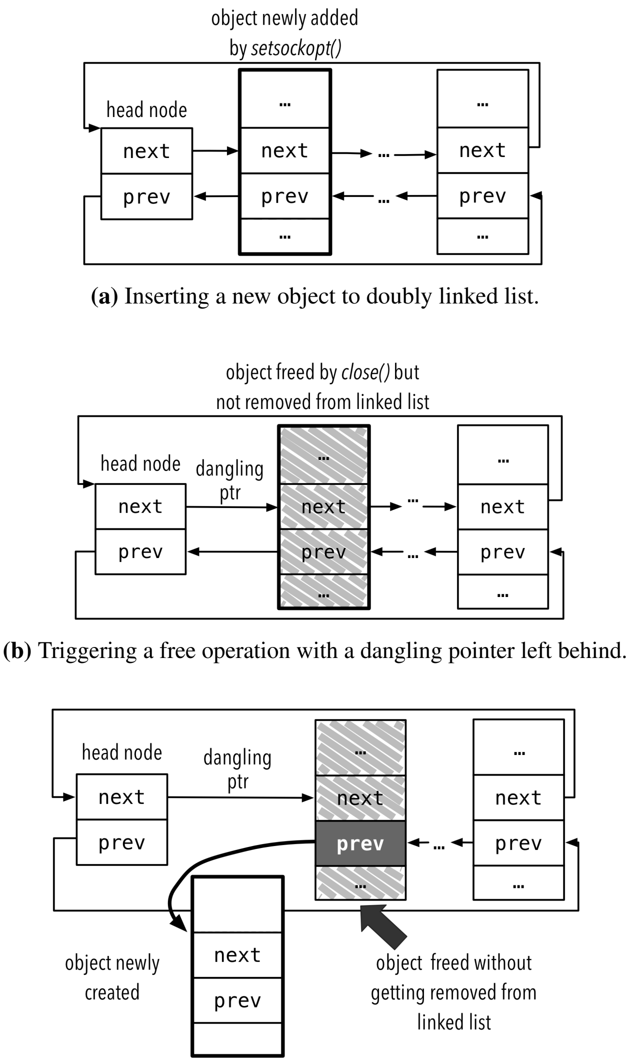
Table 1 shows a PoC program in `C` code, capable of triggering the kernel UAF vulnerability indicated by `CVE-2017-15649`. As is shown in `line 3`, `setsockopt()` is a system call in Linux. Upon its invocation over a certain type of socket (created in `line 13`), it creates a new object in the Linux kernel, and then prepends it at the beginning of a doubly linked list (see Figure 2a).

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 2b).

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 2c, 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

In a PoC program, the occurrence of a dangling pointer as well as its dereference might be triggered in the same system call.



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

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

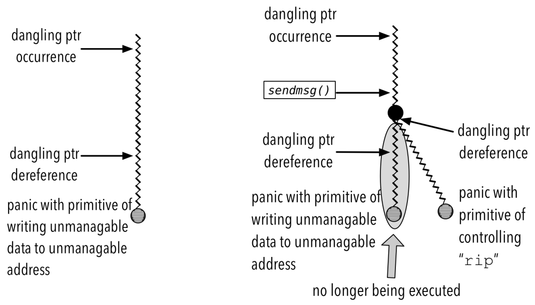
(c) Writing unmanageable data to a memory chunk freed previously.

Figure 2: Demonstrating a kernel panic triggered through a real-world kernel Use-After-Free vulnerability indicated by `CVE-2017-15649`.

freed object, resulting in an unexpected write operation which further triggers a kernel panic in consecutive kernel execution.

2.2 Challenge of Crafting Working Exploits

Following the typical workflow specified in Figure 1 to craft an exploit for the vulnerability above, in the step ❷, a security analyst needs to identify a proper system call, 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,



(a) Original running context. **(b)** New running context.

Figure 3: Context variation before and after. The original context is indicated by the PoC program in Table 1 and the new context is obtained through the insertation of the new system call `sendmsg()`.

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 2c.

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.

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.

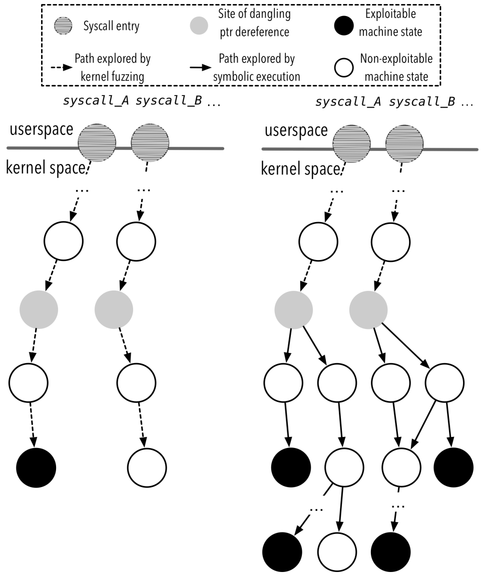
As is illustrated in Figure 3, the new kernel panic (or in other words the new PoC program) represents a new running context, where the system call `sendmsg()` retrieves the data in the freed object, dereferences it as an invalid function pointer and thus drives the kernel to a new panic state. Different from the original running context indicated by the PoC program in Table 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.

3.1 Requirement for Design

As is mentioned earlier in Section 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 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 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.


Syscall entry
Site of dangling ptr dereference
Exploitable machine state
Path explored by kernel fuzzing
Path explored by symbolic execution
Non-exploitable machine state
syscall_A syscall_B ...
userspace
kernel space
...
syscall_A syscall_B ...
userspace
kernel space
...

(a) Exploitable machine states identified by kernel fuzzing.

(b) Exploitable machine states identified by symbolic execution.

Figure 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.

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 3b). 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.

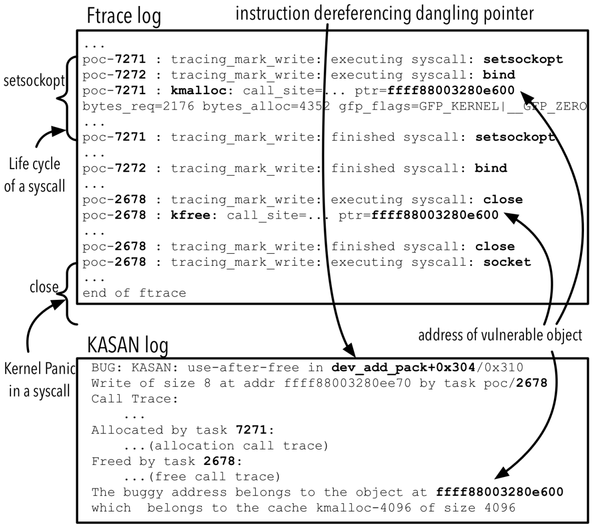
Similar to the context represented by that original PoC, a new context (*i. e.,* new kernel panic) does not necessarily assist an analyst to craft a working exploit. Moreover, as is mentioned above, a security analyst generally has difficulty in determining, following which contexts he could craft a 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 4a, 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 4b, 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 4b, symbolic execution under the context does not mean that symbolically executing kernel code at the site of kernel panic. 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.

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.



```
...
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|__GFP_ZERO
...
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
```

```
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 5: A KASAN log obtained from kernel address sanitizer as well as a kernel trace obtained through dynamic tracing.

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 [19] 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 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.

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.

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.

```
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.

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

```
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.

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. 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.

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

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.

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 3.2, 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.

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 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.

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 [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 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 [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 pre-defined 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 pre-defined 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. 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.

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 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 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 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 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 vul-

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.

nerabilities 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 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 [33]. To fulfill exploitation using the exploitable states identified, we eventually redirect the execution to an ROP chain [26] 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 [3].

6.2 Effectiveness

Table 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 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 machine states 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 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 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 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 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 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 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 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 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.

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.* [33] 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 [29], Repel *et al.* make use of symbolic execution to generate exploits for heap overflow vulnerabilities residing in user-mode applications. In [30–32], 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 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.

Acknowledgement

We thank our anonymous reviewers for their helpful feedback and valuable comments. This work was partially supported by NSF grants CNS-1718459, Chinese National Natural Science Foundation (No. 61572481) and the National Key Research and Development Program of China (No. 201604w0905). 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 funding agencies.

References

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

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

[3] Kernel 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] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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]
}
```