



Article

ERACE: Toward Facilitating Exploit Generation for Kernel Race Vulnerabilities

Danjun Liu , Pengfei Wang *, Xu Zhou  and Baosheng Wang

National College of Computer, National University of Defense Technology, Changsha 410003, China

* Correspondence: pfwang@nudt.edu.cn

Abstract: Since a large number of Linux kernel vulnerabilities are discovered every year, many vulnerabilities cannot be patched in time. Security vendors often prioritize patching high-risk vulnerabilities, and the ratings of vulnerabilities need to be evaluated based on factors such as exploitability and the scope of influence. However, evaluating exploitability is challenging and time-consuming, especially for race vulnerabilities, whose exploitation process is complicated and the success rate of exploitation is low, making them more likely to be overlooked. In this paper, we propose a new framework, called ERACE, to facilitate the process of exploiting kernel race vulnerabilities. Given a program called a proof of concept (PoC) that can trigger a race vulnerability, ERACE first applies a combination of dynamic and static analysis techniques to locate the instruction that causes the race. Then, it applies code instrumentation and static analysis to determine the timing relationship between the race instructions and the triggering type of the vulnerability and records the vulnerability context information. Next, it uses backward taint analysis to identify checkpoints that can be used to determine whether the race condition and heap spraying are satisfied and records the system calls to which the checkpoints belong. Finally, we can generate exploits based on the information collected above. To demonstrate the utility of ERACE, we tested it on 23 real-world vulnerabilities. As a result, we successfully detected the race points of 19 vulnerabilities, the timing relationship among the race instructions, and the triggering types of 17 vulnerabilities and succeeded in generating exploits for 13 vulnerabilities. ERACE can effectively help security researchers simplify the analysis process of kernel race vulnerabilities, select appropriate exploitation methods, and use checkpoints to increase the success rates of exploitations.

Keywords: race vulnerability; kernel exploitation; system security

Citation: Liu, D.; Wang, P.; Zhou, X.; Wang B. ERACE: Toward Facilitating Exploit Generation for Kernel Race Vulnerabilities. *Appl. Sci.* **2022**, *12*, 11925. <https://doi.org/10.3390/app122311925>

Academic Editors: Leandros Maglaras, Helge Janicke and Mohamed Amine Ferrag

Received: 27 October 2022

Accepted: 18 November 2022

Published: 22 November 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

With the exploding magnitude of kernel code and the advent of the multi-core era, the operating systems we now use are full of invisible threads. In this way, it is inevitable that multiple threads access the same shared code, variables, and files concurrently or out of order. Due to the infinite threads and limited resources, some resources cannot be accessed at the same time. Multiple threads can cause race vulnerabilities due to chaotic access order. According to the Common Vulnerabilities and Exposures database [1], Linux kernel race vulnerabilities discovered in 2017 nearly doubled from those discovered in 2016, and since then, nearly 100 system race vulnerabilities have been discovered every year. There are still many race vulnerabilities that are fixed without being assigned a CVE number or not fixed with a PoC available to the public [2].

A race vulnerability is very harmful. Some race conditions can lead to deadlocks and breakdowns, which can be used to perform denial-of-service attacks. Severe race conditions can trigger different types of program errors or warnings from the sanitizer, and we call them triggering types, such as out-of-bounds write (OOB) [3], use-after-free (UAF) [4], double-free [5], and direct control-flow hijacking (CFH) errors. After investigating

CVE over the past 20 years, we found that these four triggering types accounted for 85 percent of all race vulnerabilities. Some vulnerabilities can be used by attackers to escalate privileges, such as CVE-2016-8655 [6] and CVE-2019-11815 [7]. Due to the large number of vulnerabilities being discovered every year, it is difficult to fix all vulnerabilities in time. Large software vendors such as Microsoft [8] and Ubuntu [9] adopt certain strategies to prioritize the patching of security bugs. The most common strategy used to determine priority is based on the exploitability of the vulnerability.

It is more challenging to manually analyze the exploitability and generate exploits for race vulnerabilities than for other kinds of vulnerabilities. The first problem is related to debugging. We should care not only about the control flow and data flow within the thread but also about the concurrent execution between multiple threads. Due to the complex scheduling mechanism of the kernel and race conditions of a vulnerability, it is difficult for us to deterministically trigger race vulnerabilities, which makes debugging time-consuming. The second problem is related to exploitation. Race vulnerabilities have different kinds of triggering results. To determine the triggering type and choose the appropriate exploitation method, researchers have to thoroughly study the logic of the entire vulnerable code and the vulnerability context. This analysis process requires a high level of human security knowledge and experience. The third problem is related to how to detect whether a race condition is satisfied. Kernel race vulnerabilities are caused by competition between kernel threads or between kernel threads and user threads. Once the competition starts, it is difficult to perceive the execution states of the kernel space code from the user space. Due to the complexity of race vulnerability exploitations, no research has systematically analyzed the process of exploiting race vulnerabilities. Thus, race exploitation is still perplexing to the public.

In this paper, we propose an exploitation framework named ERACE to evaluate the exploitability of kernel race vulnerabilities. ERACE can simplify the process of exploiting race vulnerabilities in three aspects. First, the race point detector helps researchers automatically locate the position of race instructions triggered by a PoC. Second, the vulnerability analyzer determines the execution order of the race instructions that cause the vulnerability and analyzes the triggering type of the race vulnerability. Therefore, researchers can choose an appropriate exploitation method more quickly. Third, the checkpoint detector identifies the checkpoint near the race instruction, that is, the instruction that can reflect whether the race condition and heap spraying are satisfied, and records the system call to which the checkpoint belongs. Finally, we construct an exploit based on the information obtained above. ERACE greatly standardizes and simplifies the process of exploiting race vulnerabilities. We introduce the detailed workflow of ERACE in Section 4.

To evaluate the accuracy and effectiveness of ERACE, we tested 23 real-world vulnerabilities. As a result, we successfully detected the race points of 19 vulnerabilities, the timing relationships among the race instructions, and the triggering types of 17 vulnerabilities, and assisted in generating exploits for 13 vulnerabilities.

In general, this paper makes the following contributions.

- We design and implement ERACE, a new framework to facilitate the process of exploiting kernel race vulnerabilities, which applies a combination of dynamic and static analysis techniques. ERACE can effectively help security researchers simplify the analysis of kernel race vulnerabilities, select appropriate exploitation methods, and use checkpoints to increase the success rate of privilege escalation.
- We analyze the kernel race vulnerabilities in depth and modeled their exploitation process. The main process includes locating the race point, identifying the triggering type of the vulnerability, and identifying the checkpoint of the race.
- We demonstrate the utility of ERACE. By testing 23 real-world race vulnerabilities in the Linux kernel, ERACE facilitates the exploit generation for 13 race vulnerabilities.

The organizational structure of this article is as follows. Section 2 describes the background and challenges of our research. Section 3 provides an example to show the process of exploiting race vulnerabilities and presents an overview of ERACE. Section 4 de-

describes the design details of ERACE. Section 5 describes the implementation of ERACE. Section 6 demonstrates the utility evaluation of ERACE. Section 7 summarizes the work related to this research. Finally, we conclude this work in Section 8.

2. Background and Challenges

It is challenging to exploit kernel race vulnerabilities. First, it requires researchers to be equipped with comprehensive security knowledge and experience such as knowledge of the underlying operating system, binary reverse engineering, heap, stack, etc. Second, the indeterminacy of a race vulnerability decreases our analysis efficiency. For common vulnerabilities such as stack overflows, we can set accurate breakpoints to figure out whether the vulnerability has been triggered. However, for race vulnerabilities, it is unknown whether every execution can satisfy the race condition, further increasing the difficulty of debugging and analysis. Moreover, even senior security researchers have to spend a lot of time on reverse analysis to find useful exploit primitives or heap-spraying structures. This process is quite complicated and boring.

Many researchers have defined race vulnerabilities. Jeong D.R. et al. [10] defined a data race, which is a behavior in which the output is dependent on the sequence or the timing of other non-deterministic events. If two memory-access instructions meet the following three conditions, a data race may occur: (i) they access the same memory location; (ii) at least one instruction is a write instruction; and (iii) they can be executed concurrently. A data race is one type of concurrent vulnerability. The concept of concurrency vulnerabilities is broader. However, most of the exploitable and user-controllable vulnerabilities belong to the data race type. Therefore, the definition of a data race by Jeong D.R. is more relevant to our research. As shown in Figure 1, when a race occurs, C1, C2, and C3 are the instructions that access the same memory location. C3 is a write instruction. C1 and C3, and C2 and C3 can be executed concurrently and satisfy all three conditions.

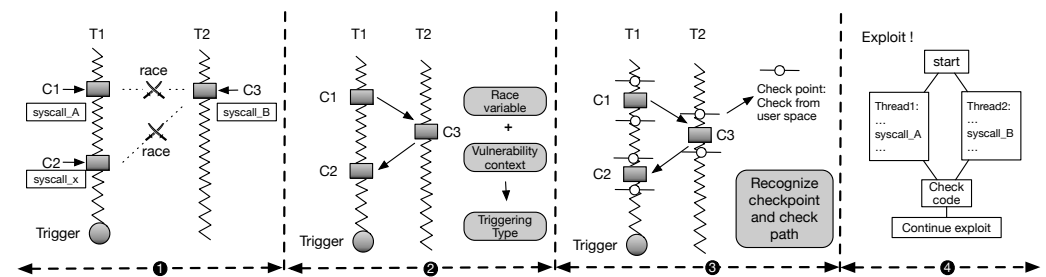


Figure 1. The typical workflow of crafting a working exploit. ① Identify the race instruction; ② Determine the timing relationship of the race instruction and the triggering type and select the appropriate exploitation method; ③ Determine how to check whether the race condition and heap spraying are satisfied; ④ Construct the exploit. Note that the zigzag line indicates the kernel execution, and `syscall_A` and `syscall_B` denote the system calls that are attached to the occurrence of the race points in the two threads, T1 and T2. `syscall_x` can be `syscall_A` or another system call.

2.1. The Process of Exploiting Race Vulnerability

The process of exploiting a data race is shown in Figure 1. Suppose that a researcher obtains a PoC that can trigger a data race but cannot successfully exploit the vulnerability yet. A successful exploitation means that we can escalate privileges from ordinary users to superusers. For a vulnerability caused by a data race, the race point is equal to the vulnerability point, whereas the vulnerability triggering point is equal to the crash point. The vulnerability point is different from the crash point for most data races. However, for most common kernel vulnerabilities, such as an 00B write and UAF not caused by data races, the vulnerability point is no different from the triggering point and the vulnerability point. Researchers need to perform the following steps to successfully escalate a privilege.

First, the researcher should find the root cause of a race vulnerability based on the PoC, that is, the race instructions that accessed the same variable or memory, and determine

the system call to which the race instruction belongs (see ❶ in Figure 1). This is conducive to the subsequent analysis and the construction of competing threads. Determining the root cause of the vulnerability requires analyzing the source code, understanding the logic of the source code, or completing a reverse analysis based on the output of the stack traceback information by the kernel address sanitizer (KASAN) [11] at the crash point. For most data races, crash points are different from race points and they may even be far away from each other. Hence, it is not easy to locate race instructions.

Second, the timing relationship of race instructions and the triggering type of the vulnerability need to be determined (see ❷ in Figure 1). Competition is the cause, whereas triggering is the result. A pure race is not necessarily exploitable. However, if a race can lead to UAF or OOB write vulnerabilities, then the race may be exploitable. Its exploitability also depends on the specific vulnerability context.

Third, the method of checking whether the race and heap spraying were successful needs to be determined (see ❸ in Figure 1). Assuming that the race triggers a UAF issue and only after the race succeeds, heap spraying can be used to complete control-flow hijacking and privilege escalation. We can directly perform heap spraying without confirming whether the race succeeded or not and just confirm whether the privilege was escalated at the end of the exploit. However, what happens when the race fails? We have to restore the heap-space layout and perform competing and heap spraying again, which are both time-consuming and greatly prolong the entire exploitation process. What is worse is that the heap-space layout might have been destroyed once the heap spraying occurred, making it difficult to recover the heap-space layout. Sometimes the system may directly crash and the exploitation cannot continue. Similarly, proceeding to the next exploitation stage without checking whether the heap spraying was successful may also reduce the success rate of the exploitation.

Fourth, the exploitation strategy is determined and the final exploitation is finished (see ❹ in Figure 1). According to the recognized race instruction and the relative system call, we can construct competing threads. We can craft race and spraying checking codes with the check path identified and choose an appropriate exploitation method based on the triggering type.

2.2. Challenge of Crafting Working Exploits

In step ❶, it is hard to locate the race instructions. First, there is no existing tool that can directly locate the race points. The only tool that can be used is the KASAN [11] instrumentation mechanism. The latest instrumentation tool called TCSAN [12] can only be used in kernel 5.8 and above so it cannot cover all kernel versions. Generally, if we compile the kernel with the KASAN option set, the kernel will output warning information when the vulnerability is triggered. For most common types of kernel vulnerabilities, such as an OOB write or UAF not caused by data races, the triggering point is equal to the vulnerability point and it is more convenient to locate the vulnerability point by gathering useful information from the KASAN report. However, for a data race, the triggering point is not the vulnerability point. The triggering point usually occurs later than the race point in the instruction execution sequence. Second, the magnitude of kernel source code is too big and it is difficult to perform manual code auditing. Therefore, a more effective method should be adopted to locate race points.

In step ❷, there is no existing tool to determine the execution order of race instructions when a vulnerability is triggered and it is difficult to accurately identify the triggering type. KASAN's warning information alone is not enough to determine the triggering type of the vulnerability. For example, KASAN often reports null-dereference warnings for race vulnerabilities, which is of little help to the exploitation because in the current system with the general configuration, a null dereference cannot be used to escalate a privilege. In fact, the triggering type of the race vulnerability may be an OOB write or another type.

In step ❸, our exploit runs in the user space. With limited privilege, how can we confirm whether competing and heap spraying succeeded? For vulnerabilities that can

trigger direct CFH, we can judge whether the race condition was satisfied by checking whether the malicious code was executed successfully. However, it is hard to check the race status for vulnerabilities that can trigger a UAF, double-free, or heap OOB write. We can only check whether the malicious code was executed successfully after heap spraying succeeded, which costs too much time and may lead to exploitation failure. If we can check it before heap spraying, we can greatly reduce the running time of the exploit and increase the success rate of the exploitation. Similarly, it can also improve the success rate of the exploitation by checking whether the heap spraying succeeded.

To solve the problems mentioned above, we need to adopt automatic program analyzing techniques to avoid complicated and repetitive manual analysis, which may produce some omissions. The original objective of ERACE is to assist in the exploitation of race vulnerabilities to quickly evaluate its risk rating. Therefore, ERACE does not fully automate exploit generation but facilitates the process of exploitation.

To solve the problems mentioned above, we need to adopt automatic program analyzing techniques to avoid complicating and repetitive manual analysis, which may produce some omissions. The original objective of ERACE is to assist in the exploitation of race vulnerabilities to quickly evaluate its risk rating. Therefore, ERACE does not fully automate exploit generation but facilitates the process of exploitation. ERACE should meet the following three design requirements.

First, ERACE should help researchers locate race instructions and relative system calls, which corresponds to step ❶ in Figure 1. The race point is equal to the vulnerability point, from which researchers can quickly determine the root cause of the vulnerability. Second, ERACE should help researchers determine the execution order of the race instructions and the triggering type and extract the vulnerability context, which corresponds to step ❷ in Figure 1. In this way, a researcher can select a suitable exploitation method according to the triggering type and the vulnerability context. Determining the execution order of race instruction makes it easier to debug the vulnerabilities without wasting time waiting for the success of the race. Third, ERACE should help researchers locate checkpoints to check whether the race condition and heap spraying are satisfied, which corresponds to step ❸ in Figure 1. Some code in the kernel can modify variables or data structures. The user layer can indirectly determine where the kernel code is executed or whether kernel memory is sprayed with user-forged data by detecting changes in these variables or data structures. Checkpoints are instructions that modify variables or kernel data structures. ERACE should also find the program path for the user layer to access these variables or data structures.

3. Motivating Example and Overview

3.1. Motivating Example

Figure 2 shows a PoC program in the C coding language that can trigger the kernel race vulnerability CVE-2016-8655. `setsockopt()` in line 3 and line 9 is a system call that can operate on the socket created in line 15. If the operation option in line 3 is `PACKET_RX_RING`, the kernel will call `packet_set_ring()` to set the ring buffer. If the operation option in line 9 is `PACKET_VERSION`, the kernel will call `packet_setsockopt()` to modify the packet version. Both functions access the value `tp_version`, which may cause a competing conflict.

In line 21 and line 22, the PoC creates two threads. These two threads will make kernel call `packet_set_ring()` and `packet_setsockopt()`, respectively. Calling these two threads in an infinite loop may lead to race conditions and trigger the vulnerability.

The triggering principle is shown in Figure 3. Normally, when the `packet_set_ring()` creates a ring buffer, if the packet version is `TPACKET_V3`, it will call `init_prb_bdqc()` to initialize the timer. When the packet version is greater than `TPACKET_V2`, the kernel will call `prb_shutdown_retire_blk_timer()` to release and delete the previously initialized timer. If another thread modifies the packet version in the middle of the call, although the socket will call `kfree()` to release the previously initialized timer when the `close()` is executed

at the end, the timer is not deregistered in the kernel queue. When the timer expires, the callback in the timer is called, triggering a UAF issue.

```

1 void *race1(void *arg) {
2     ...
3     setsockopt(fd, 0x107, PACKET_RX_RING, &tp, ...);
4 }
5
6 void *race2(void *arg) {
7     ...
8     val = TPACKET_V1;
9     setsockopt(fd, 0x107, PACKET_VERSION, &val, ...);
10 }
11
12 int race_loop()
13 {
14     while(1) {
15         fd = socket(AF_PACKET, SOCK_DGRAM, htons(ETH_P_ARP));
16         ...
17         // initialize the packet version to be V3
18         val = TPACKET_V3;
19         setsockopt(fd, 0x107, PACKET_VERSION, &val, ...);
20         // creat two racing threads
21         pthread_create(&thread1, NULL, race1, NULL);
22         pthread_create(&thread2, NULL, race2, NULL);
23
24         pthread_join(thread1, NULL);
25         pthread_join(thread2, NULL);
26
27         close(fd);
28     }
29 }

```

Figure 2. A PoC code fragment of kernel race vulnerability, CVE-2016-8655.

In this vulnerability, the race point is the instruction, where the variable `po->tp_version` is marked in red in Figure 3. Thread A reads `tp_version` twice (V1 and V2) and thread B writes `tp_version` once (V3). When we tested the kernel (Linux kernel version 4.12.1) compiled with KASAN, the vulnerability point reported by KASAN is in the kernel's `run_timer_softirq()`. Actually, this is the triggering point, not the race point. There is no output information about the stack traceback at the race point. Researchers still need to spend a lot of time auditing the source code so it is difficult to locate the vulnerability point simply from the KASAN report. In addition, there is no other tool that is more helpful for locating race points.

The triggering type of this vulnerability is a UAF. According to our observations, KASAN sometimes reports the vulnerability type as a UAF and sometimes reports it as a null dereference. The KASAN report is not accurate enough. Many vulnerability types, such as the 00B write and double-free types, can also trigger null dereferences so inaccurate reports will increase the difficulty of the exploitation. In our experiments, we compare the accuracy of ERACE and KASAN in recognizing triggering types. Moreover, the information provided by KASAN is very limited and we can neither evaluate the exploitability nor choose an appropriate exploitation method. For the triggering types, in addition to the UAF, there are also the 00B write, double-free, and direct CFH types. Different triggering types have different exploitation methods, which we introduce in detail in Section 4.

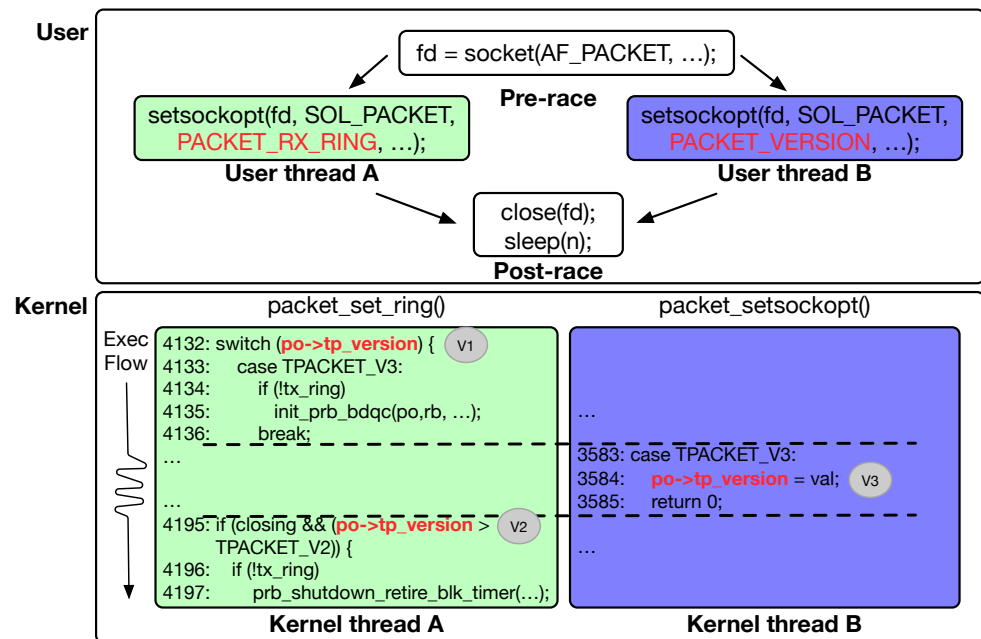


Figure 3. Principle of the race vulnerability CVE-2016-8655. As a user program calls two system calls concurrently, a data race on `po->tp_version` may occur depending on the execution order, which triggers a UAF issue that allows an attacker to launch a privilege escalation attack.

The checkpoint of this vulnerability can be seen in Figure 4. Confirming whether the race condition is satisfied is equivalent to confirming whether V3 is executed between V1 and V2. Since the vulnerable code is located in the kernel space, we can only craft user-space programs to confirm this. If we want the kernel to successfully execute V3, we must bypass the condition at point P2, otherwise, it will return directly. The statement at P3 will set `pg_vec` to 1, making the condition of P2 unable to be bypassed so P2 must be executed before P3. As long as V3 can modify `tp_version`, it means that P2 was executed before P3. Then, because the time window between P2 and V3 is smaller than the time window between P3 and V2, it is more likely that V3 is executed before V2. The first checkpoint is `po->tp_version`, which is modified by V3, and we need to perceive the change of the `tp_version` value from the user space. However, at this time, we can only know that there is a high possibility that V3 is executed before V2 and do not know whether V3 is executed after V1. If V3 is executed before V1, `init_prb_bdq()` will not be called to construct the dangling timer. Therefore, confirming whether V1 is executed before V3 is equivalent to judging whether `init_prb_bdq()` is executed. The `init_prb_bdq()` function will execute a statement P1 and P1 will modify the `packetblockdescriptor->header.bh1.offset_to_first_pkt` area. We need to perceive the change in this value from the user space. The second checkpoint is the memory value modified by P1. In general, P1 and V3 are what we call checkpoints because these two statements modify the memory value and the user-space program can obtain this value. Therefore, we can indirectly confirm the kernel's execution state by perceiving the changes in these values from the user space. Similarly, we can also use checkpoints to confirm whether the kernel's memory is sprayed with user-forged data. Identifying checkpoints also requires researchers to spend much time auditing the logic of the source code.

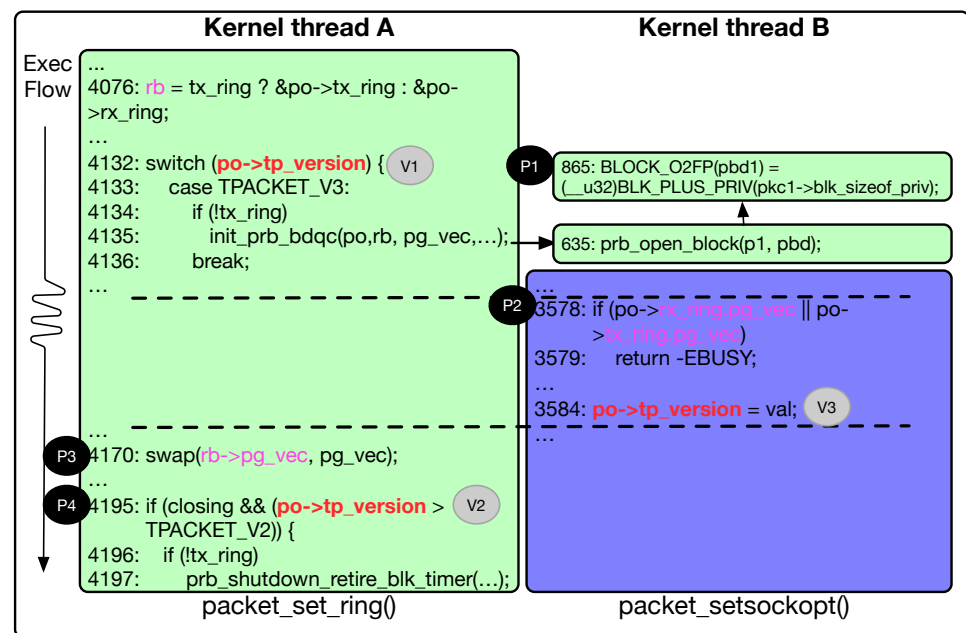


Figure 4. Checkpoint of CVE-2016-8655. P1 and V3 are the two checkpoints.

3.2. Overview

To meet the requirements mentioned above, we designed an exploitation framework, named ERACE, to assist in the exploitation of a race vulnerability. The top-level design of ERACE can be seen in Figure 5. ERACE contains three components: a race-point detector, vulnerability analyzer, and checkpoint detector. They can satisfy the above three requirements, respectively.

First, the input of the race-point detector is the kernel source code containing the race vulnerability and the PoC that can trigger the vulnerability. The source code is compiled into an LLVM intermediate representation and an executable binary file bzImage. Then, the multi-thread program in the PoC is converted into two single-thread programs. Next, the two programs in the executable kernel file bzImage run and the corresponding execution path is recorded. At the same time, the LLVM intermediate representation is statically analyzed to identify all the race instruction pairs. Finally, whether the actual execution path executes the race instruction pairs is determined. Therefore, we can accurately locate the race instructions.

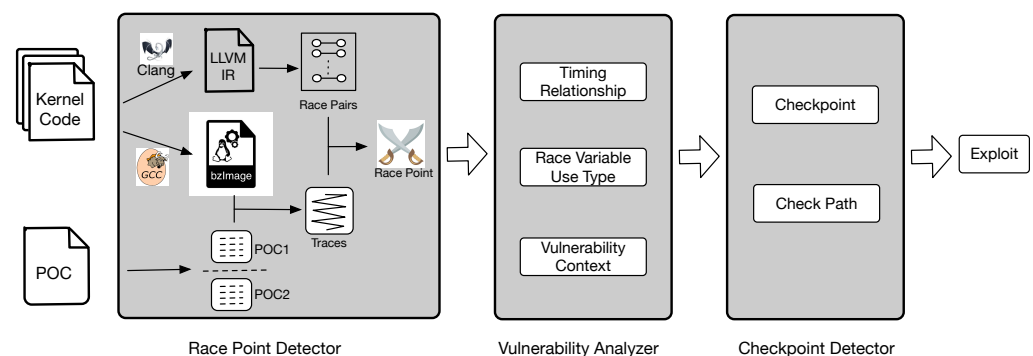


Figure 5. Overview of ERACE.

The vulnerability analyzer is responsible for collecting the vulnerability context, identifying the triggering type, and determining the exploitation method. To obtain an accurate and detailed vulnerability context, race points are instrumented to determine their timing relationship. Then, the static analysis is applied to recognize the race variable and its use

type and the vulnerability context is recorded at the same time. Based on the use type of the race variable and the vulnerability context, ERACE can determine the triggering type.

The checkpoint detector is responsible for locating checkpoints and determining the corresponding system call interface. Checkpoints can reflect the success of competing and heap spraying. The principle is to perceive the executing states of the kernel from the user space or confirm that the kernel memory is arranged with user-forged data. First, backward taint analysis is applied to find the kernel data that can be leaked by a leaking anchor. Then, the control-flow analysis is used to check whether there is a system call interface that can reach the leaking anchor without root privileges.

The last step is to generate the final exploit. First, competing threads are crafted based on the identified race point and related system call. Then, a checking code is crafted based on the identified checkpoint and related system calls. Next, the subsequent exploitation method and structure based on the identified race-triggering type and vulnerability context are determined. The follow-up exploitation phase can be found in existing research [13–16].

4. Design

This section is divided into three subsections to introduce the three components of ERACE.

4.1. Race-Point Detector

This section mainly introduces how to identify race instructions in a PoC, which is mainly divided into three steps. First, we need to obtain the execution path of the PoC, then perform alias analysis on the kernel code to find all possible race instruction pairs, and finally, compare the race candidates that appear in the PoC trace to find the real race points.

4.1.1. Tracking Down PoC Traces

We need to obtain the execution path of the PoC in the actual environment so that we can analyze the position of the race point. It seems simple to put the PoC into a real system to run, and record the execution trace of the system until the vulnerability is triggered. However, there are three problems with doing this. First, the real system environment is very complicated and the race window for many race vulnerabilities is very small (as shown in Figure 3, the race window is the code length between V1 and V2) so it is difficult to satisfy the race conditions. Even if a race condition is accidentally triggered, it is difficult for us to detect whether a race has occurred because the race does not exhibit abnormal behaviors. Second, as the PoC itself is a multi-thread program and there is still a large number of background threads running inside the system while the PoC is running, it is difficult for us to distinguish between threads that can cause a race. Third, the magnitude of kernel source code is very large and a large number of execution traces will be generated during this period, which is too complicated to analyze.

When debugging the vulnerability, we found that the competing threads in the PoC executed the race instructions but did not trigger the vulnerability if the race condition was not satisfied. Now that the PoC has executed the race instruction, we can locate the race point by analyzing its execution trace. Based on this discovery, we can narrow down the search space of instructions as much as possible, avoiding the above three problems. The method we use is to convert the multi-thread PoC into two single-thread programs, namely PoC1 and PoC2, and collect the execution traces of the two threads running separately once so that the execution traces of the two competing threads can be distinguished. In the process of transforming the program, we cannot change the parameter settings of the original thread to avoid affecting the original control flow. In addition, the competing thread is converted into an ordinary function and we can extract its one-time execution trace.

We use kcov [17] to record the execution traces and convert the binary traces into kernel source traces. kcov is a lightweight tool and we can enable it by setting `CONFIG_KCOV` when compiling the kernel. We can use `ioctl` to send `KCOV_ENABLE` and `KCOV_DISABLE` to the kernel to start

and pause the trace recording, and the final path is recorded in the shared memory between the kernel and the user. We mark the paths of PoC1 and PoC2 as T, which contains two separate traces.

4.1.2. Identifying Race Candidates

We should statically analyze the kernel's LLVM intermediate representation to find possible race instruction pairs, which we call race candidates. First, we need to perform alias analysis on the LLVM intermediate representation and record the aliases from all variables and structure pointers. Then, we should identify the instructions that reference the same variable or its aliases and pair the race instructions to obtain the race candidates. We use the term $Race_{cand}$ to denote a set of possible race instructions, where each member is represented by $\langle r, w \rangle$ or $\langle w, r \rangle$, where r is a read instruction and w is a write instruction.

The race instruction pair must contain a read instruction and a write instruction. The execution order of two read instructions does not affect the execution result of the program; hence, there is no need to study this case. The execution order of two write instructions may affect the results of the program because the value stored in the memory depends on the last write instruction, which affects the read values of subsequent instructions. However, it is ultimately the competition between a write instruction and a read instruction. If there are only two write instructions writing to the same memory and the written value is not used by any code, it will not affect the execution result of the program.

4.1.3. Confirming the Race Point

Finally, we search for whether there is a race candidate appearing in $Race_{cand}$ and the PoC trace. In Section 2, we introduced the features of exploitable race vulnerabilities. If there are two race candidates $\langle C_i, C_k \rangle$ and $\langle C_j, C_k \rangle$ appearing in the PoC trace, then C_i , C_j , and C_k are very likely to be real race points. According to our definition of race vulnerabilities in Section 2 and the above analysis, the race instruction pair must contain a read instruction and a write instruction. There are two situations. One is when C_i and C_j are read instructions and C_k is a write instruction. The other is when C_i and C_j are write instructions and C_k is a read instruction (see, for example, CVE-2016-8655 in Figure 3). V1 and V2 are equivalent to C_i and C_j , V3 is equivalent to C_k , and V3 is a write instruction.

4.2. Vulnerability Analyzer

This section mainly discusses how to analyze race vulnerabilities to obtain important vulnerability information. The first step is to identify the execution order of the race instructions that trigger race conditions, which is beneficial for vulnerability debugging. Then, the triggering type is identified and finally, the appropriate exploitation method is selected according to the triggering type and the vulnerability context.

4.2.1. Identifying the Race Timing Relationship

We design a user-controllable delay injection method to identify the execution order that triggers the race condition. The previous race point detector locates the race points C_i , C_j , and C_k . There are six execution orders for the three race points, as shown in Figure 6. We test these six execution orders to determine which execution order can trigger a race. To control the timing of the instruction execution accurately and easily, we propose a user-controllable delay injection method. Specifically, we insert a delay instruction before each race instruction and the delay time is determined by the user. We can set the instruction execution timing in the PoC. When the kernel executes to the race point, each race point is delayed for a certain time. For example, if C_i , C_j , and C_k are delayed by t , $2t$, and $3t$, respectively, then the timing relationship of the race instructions is $C_i - > C_j - > C_k$. t represents the minimum delay time. The value of t cannot be too small. If the value of t is too small, the execution order will be disordered. This user-controllable delay injection method not only simplifies timing control but also facilitates vulnerability debugging.

Researchers can quickly trigger vulnerabilities when debugging vulnerabilities without wasting time waiting for the success of the race.

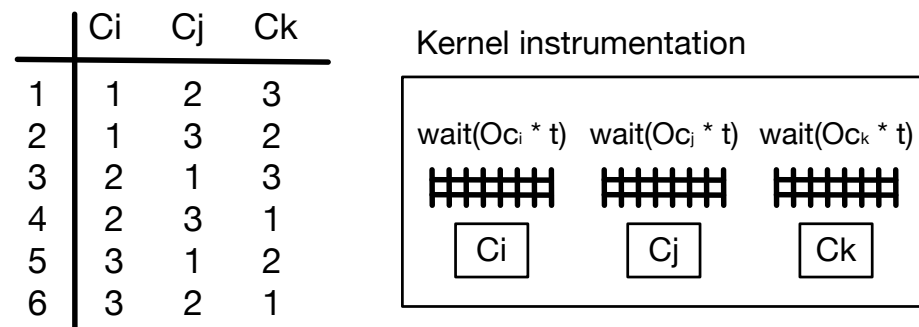


Figure 6. The permutations of the race point's execution order and kernel instrumentation. The left line denotes the sequence number. The contents of the table are the execution orders of the race points. O_{C_i} , O_{C_j} , and O_{C_k} indicate the execution order of C_i , C_j , and C_k , respectively. t denotes the minimal delay time.

4.2.2. Identifying the Triggering Type and Tracking Down the Triggering Context

Because the KASAN report on the vulnerability triggering type is not accurate, we cannot just rely on the KASAN warning information. We should comprehensively determine the vulnerability triggering type based on the use type of the race variable and the vulnerability context. We divide the triggering types into four categories, namely the UAF, double-free, 00B write, and direct CFH types. Generally speaking, double-free is a special kind of UAF in which a dangling pointer is used at a releasing site. However, the general exploitation methods of the UAF and double-free types are different. Therefore, we have to distinguish between these two triggering types to help determine the specific exploitation method. There exist other triggering types that cannot be effectively exploited or that are rarely seen. For example, some null-dereference vulnerabilities cannot be exploited at present and use-before-initialization vulnerabilities are relatively rare. The STACKLEAK [18] mitigation mechanism can easily prevent use-before-initialization vulnerabilities from being exploited.

Now, we perform data-flow analysis on the race variable to determine the triggering type and record the corresponding vulnerability context. The race variable is the object operated by the race instruction. For example, `po->tp_version` is the race variable in CVE-2016-8655. We trace the race variable to find where it flows and observe its behaviors. First, if it can flow to a jump table index or jump address to explicitly affect the control flow, it may trigger a direct CFH. Then, we perform backward data-flow analysis to find where it comes from. If it can be controlled by the user, we record the relative syscall and user parameters. Second, if the race variable represents a length or a string pointer and it finally directly or indirectly affects the length parameter of the memory copy function or the index of the memory access, it may trigger an 00B. We should record the overflow point and overflow length. Third, if the race variable is used in a conditional statement, which implicitly influences the control flow or points to a certain kernel structure, it is possible to trigger a UAF or double-free. We should further identify the program point that references the dangling pointer. If the reference points are all releasing points, it is most likely to be a double-free. If the reference point contains general memory access, it is very likely to be a UAF. We should record the allocation point, allocator type, release point, and reference point of the vulnerable object.

After the triggering type is identified, the subsequent exploitation process used can be chosen from existing studies. If the triggering type is an 00B, we can use KOUBE [16] to continue to explore the ability of the vulnerability, that is, the write length and the write value. Then, appropriate target objects are selected for exploitation. If the triggering type is a UAF, we can use FUZE [13] to obtain the statement that generates and references the dangling

pointer; then, the system call to which the statement belongs is identified, and whether it can hijack the control flow or perform arbitrary read or write operations is verified. If the triggering type is a direct CFH, we can use KEPLER [14] to search for blooming gadgets and bridging gadgets in the kernel and construct a return-oriented programming (ROP) chain to complete the exploitation. If the triggering type is the double-free type, we can use SLAKE [15] to identify the kernel objects that are useful for exploiting and the corresponding system calls and control the kernel space layout to achieve the exploitation.

4.3. Checkpoint Detector

The main task of this section is to statically analyze the kernel's LLVM intermediate representation, identify instructions that modify the kernel data structure near the race point, and then check whether there is a channel through which the user can read the modified value. ELOISE [19] performed similar work. Its goal is to identify the elastic object in the kernel and check whether the user can leak the content of the elastic object to bypass the kernel's KASLR mitigation mechanism. However, there are two fundamental differences between the work of this paper and that of ELOISE. First, the purposes are different. The objective of ELOISE is to find all elastic objects, whereas the purpose of ERACE is to find channels that can leak the value modified by the checkpoints. Therefore, the technique adopted by ELOISE is more limited. Second, ELOISE only considers the channel through system calls, whereas this paper also considers the interface provided by the module itself and the special communication method of the shared memory, which makes the analysis results more comprehensive.

4.3.1. Tracking Down Checkpoint Candidates

First, we need to identify the checkpoint candidates and mark the trace recorded when the race-point detector executes the PoC in the first step as T . T contains the execution paths of PoC1 and PoC2. Since the kernel LLVM intermediate representation we compiled contains symbol information, we can find the execution path of T that corresponds to the LLVM intermediate representation through the symbol information. We identify the memory write instruction on T as a checkpoint candidate and mark it as C_{cand} .

Then, we should accurately identify the structure type in the kernel, denoted as S . The kernel structure is used to store state information and important data when the kernel is running and it is also the object that our checkpoint modifies. The kernel structure is divided into two categories: one is the global structure in the kernel and the other is dynamically generated when the kernel is running and may be located on the stack or the heap. There are two common types of heap allocation functions. One is the functions allocated to the general cache or zone space such as `kmalloc`, `kzalloc`, and `malloc`. The other is the functions allocated to a special cache or zone space such as `kmem_cache_create`, `kmem_cache_alloc`, and `kmem_cache_alloc_node`. In the LLVM intermediate representation, structure allocation instructions or value-transfer instructions will expose the structure type information.

4.3.2. Filtering out Checkpoint Candidates

As mentioned above, the checkpoint needs to meet two requirements. One is that the value modified by the checkpoint must be in a kernel data structure and it is meaningless if it is just an ordinary value-transfer instruction. The other is that there must be a channel to leak the kernel data modified by the checkpoints to the user space and it does not require root privileges.

For the first requirement, we need to determine which kernel structure C_{cand} modifies. We use C_{cand} as a starting point to perform backward taint analysis on the modified value to check whether its source comes from S . If it is not in S , then the checkpoint is filtered out. The remaining checkpoint is denoted as C_{real} , which indicates that the source of the leaked memory comes from the kernel structure S . The modified kernel structure and corresponding offset are denoted as S_{real} .

Before we expound on the second requirement, we should discuss the communicating mechanism of the kernel. There are three traditional methods of communication between the user space and kernel space, `/proc`, `netlink`, and `ioctl`. The `/proc` directory is a virtual file system through which users can read data about system hardware and the currently running processes. However, root privileges can access all kernel information. The kernel data that users can view are very limited. `Netlink` is a kind of duplex communication between the kernel and user but it needs the support of specific kernel functions. For a vulnerability exploitation, the scenarios used are very limited. `ioctl` is a more flexible communication method and is implemented in many Linux modules. The channel that can obtain the checkpoint information studied in this paper is similar to `ioctl`. Users can obtain the information in the kernel through the channel provided by the Linux kernel module without affecting the original execution flow of the kernel. There are two main methods for obtaining checkpoint data. First, the kernel module can leak the kernel data to the user space through some important functions. For example, when capturing packets, users can obtain the timestamp and version information of the kernel layer data packets (called by `getsockopt`). Second, the shared memory between the kernel space and user space can be used to directly obtain the kernel data.

There are some functions in the kernel that can leak kernel data to the user space. We use them as the starting points for backward taint analysis. We summarized these functions and called them leaking anchors and they include `copy_to_user()`, `put_user()`, `copyout()`, `nla_put()`, and `skb_put()`. Leaking anchors generally contain two important parameters; one is the address of the kernel data, denoted as k_addr , and the other is the length of the leaked data, denoted as k_len . We use k_addr and k_len as the starting points for backward taint analysis and find the sources of k_addr and k_len , denoted as S_addr and S_len , respectively. If $[S_addr, S_addr + S_len]$ belongs to S_{real} , it means that the value modified by the checkpoint can be leaked. Moreover, control-flow analysis should be performed to ensure that the leaking anchors can be reached from the user interface with normal privileges. If a function such as `capable(CAP_SYS_ADMIN)` appears on the path, it means that root privileges are required to obtain the kernel data. The general taint analysis and control-flow analysis principles are shown in Figure 7. Note that `copy_to_user()` is only a representative function in leaking anchors and there are other leaking functions mentioned above.

Users can also obtain kernel data through the shared memory. ELOISE [19] did not consider this special communication method. Some Linux modules require a large amount of data transmission between the kernel space and user space. To improve the efficiency of data transmission, it is necessary to create a shared memory between the kernel and the user. For example, paths recorded by the `kcov` module and data packets captured in `af_packet` are all stored in the shared memory so that users can save time by reducing one copy and one system call when reading data. When creating the shared memory, `vm_insert_page(struct vm_area_struct *, unsigned long addr, struct page *)` is called to complete the mapping between the user space and kernel space. The parameter `addr` is the user-space address and the parameter `page` is the kernel address. ERACE performs backward taint analysis on the page parameters to find the relative source structure.

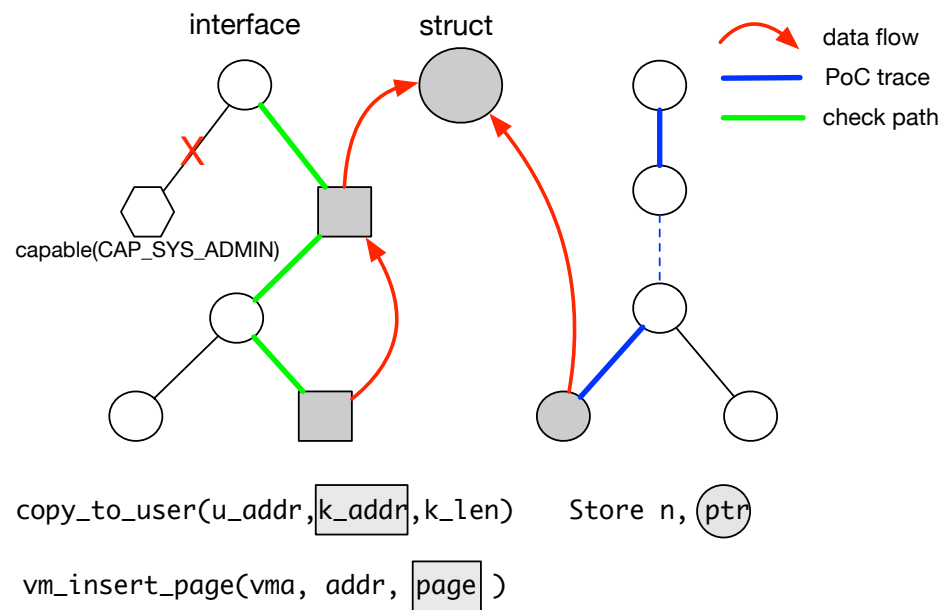


Figure 7. Illustration of backward taint analysis starting from `copy_to_user()`, `vm_insert_page()`, and `ptr`. Arguments `k_addr` and `ptr` originate from the same structure. Interface refers to a system call or an export function from a module.

5. Implementation

We implemented a prototype of ERACE, which contains three main components: the race-point detector, vulnerability analyzer, and checkpoint detector. The static analysis part of the race-point detector is based on SVF [20] and the dynamic path collection is based on kcov [17]. The race-point detector contains 2500 lines of C code and is mainly responsible for identifying the race points. The static analysis and instrumentation of the vulnerability analyzer contain 1500 lines of C code and are mainly responsible for identifying the timing relationships and triggering types. The static analysis of the checkpoint detector contains 2000 lines of C code and is mainly responsible for identifying the checkpoints. The following paragraphs contain some implementation details about the above design.

Kernel division. During static analysis, the kernel is divided and the correlative modules are merged. As the Linux kernel source code is too large, i.e., reaching millions in magnitude, the lines of the compiled LLVM intermediate representation with symbolic information are three to four times those of the source code, which severely reduces the speed of static analysis and increases memory overhead. To solve this problem, Razzer [10] and CRIX [21] divided the different modules according to the directory structures, and k-miner [22] analyzed them from different system calls. Although this effectively improved the speed of static analysis, some vulnerabilities were related to several modules. For example, the traces of the PoC we collected span multiple modules. If the kernel is divided into modules, it leads to false negatives. Therefore, when dividing the kernel, we integrate the modules involved in the PoC trace.

Relaxed alias analysis. When the race-point detector identifies suspicious race instruction pairs, it should first perform alias analysis on the variables and structure pointers. There is a characteristic in the Linux kernel. Two unrelated functions may reference the same structure member. Unrelated means that there is no direct or indirect calling relationship. For example, in Figure 3, the `packet_set_ring()` and `packet_setsockopt()` functions can be called through different user call interfaces. However, as long as the same socket handle parameter is passed in, the same structure member `po->tp_version` will be referenced. When using the existing SVF tool [20] or the `AliasAnalysis` pass in LLVM itself to perform alias analysis, it is found that this situation will be under-reported. This is because traditional alias analysis can only identify alias relationships in the same function or the same path. Therefore, we improved the alias analysis method of the SVF and adopted relaxed alias

analysis. As long as the structure type pointed to by two pointers is the same, we treat the two pointers as aliases. The relaxed alias analysis has the problem of false positives, but our goal is to locate more race-point candidates, avoiding omissions. In our research, the problem of false negatives is more serious than that of false positives, and even if there are false positives, we can filter them based on the PoC traces.

Single-thread generator algorithm. As mentioned above, the indeterminacy of the data race makes it hard to trigger the vulnerability. However, even if the multi-thread PoC does not trigger the race condition, it may have passed the race point. To gather the execution traces of two threads, we have to transform the multi-thread PoC into two single-thread programs. We designed a single-thread algorithm to finish the task. As we know, Linux uses two functions, `pthread_create()` and `fork()`, to make programs run concurrently. First, we have to recognize these two functions. For the `pthread_create()` function, we find the two subfunctions on behalf of two threads. The third parameter of `pthread_create()` reveals the subfunctions. For the `fork()` function, we find the two branches on behalf of two processes, which we call two threads for convenience. Then, we delete `pthread_create()` or `fork()` and execute `kcov-control` statements before and after the two threads. Finally, we delete the other thread to obtain PoC1 and PoC2, respectively. We can collect execution traces in a real environment and compare them with suspicious race instruction pairs from the alias analysis to locate the final race points.

6. Evaluation

In this section, we use real-world kernel race vulnerabilities to demonstrate the utility of ERACE. Specifically, we demonstrate the accuracy of ERACE's identification of race points, triggering types, and timing relationships of race instructions, and its effectiveness in facilitating the exploitation. In addition, we also discuss some kernel race vulnerabilities that ERACE failed to facilitate.

6.1. Setup

To demonstrate the utility of ERACE, we conducted an investigation into Linux kernel race vulnerabilities in the past seven years from the Common Vulnerabilities and Exposures [1] database and syzbot [2]. Google's syzbot [2] platform has published many PoCs for Linux kernel vulnerabilities including a lot of race vulnerabilities that have not been confirmed. As discovered in our investigation of these two platforms, nearly 62 percent of race vulnerabilities had available PoCs, which can satisfy the input requirement of ERACE. Then, we excluded the race vulnerabilities that were not related to data races or that involved specific hardware devices. Finally, we selected 17 CVE that had rich public vulnerability information to evaluate our results. In addition, we randomly chose six race vulnerabilities that had available PoCs from syzbot [2]. We show these 23 vulnerabilities in Table 1.

Since the race vulnerabilities selected involved different kernel versions, in order to better evaluate the utility of ERACE we transplanted all the vulnerabilities to the 4.12.1 version of the Linux kernel. Some vulnerabilities were located in the older version of the kernel, which does not support `kcov` or KASAN instrumentation. The research in this paper did not involve bypassing the mitigation mechanisms so we turned off the default protection mechanisms of the system such as the kernel address space layout randomization SMEP and SMAP mitigation mechanisms. Since ERACE needs to perform a static analysis of the Linux kernel to compare the warning information of KASAN and collect the dynamic execution path and instruction timing relationship, it was necessary to compile three versions of the kernel. One version used Clang to compile the kernel source code into the LLVM intermediate representation. The second used `gcc` to compile the kernel, with KASAN and `kcov` configured. The third performed delay instrumentation for the race points to recognize their timing relationship. The system selected for the experiments was an Ubuntu 18.04 system, running on a desktop with 128 G RAM and an Intel(R) Core(TM) i9-10900KF CPU @3.70 GHz.

Table 1. Accuracy of ERACE. Race variable denotes the variable that race points read from or write to; AA-SVF indicates the use of traditional alias analysis of SVF; Improved AA-ERACE indicates the use of relaxed alias analysis in ERACE; Timing relationship stands for whether ERACE could recognize the true timing relationship of the race instructions; UBI denotes a use-before-initialization vulnerability. ✗ means it fails to identify, while ✓ means it succeeds in identifying.

CVE	Race Variable	Race Point		Timing Relationship	Triggering Type	Recognized Triggering Type	
		AA-SVF	Improved AA-ERACE			KASAN	ERACE
CVE-2015-8550	✗	✗	✗	✗	CFH	✗	✗
CVE-2016-6516	arg->dest_count	✓	✓	✗	OOB	✓	✓
CVE-2016-8655	po->tp_version	✗	✓	✓	UAF	✓	✓
CVE-2017-10661	ctx->might_cancel	✓	✓	✓	UAF	✗	✓
CVE-2017-11176	sock	✗	✓	✓	Double-Free	✗	✓
CVE-2017-15265	port	✗	✓	✗	UAF	✓	✓
CVE-2017-15649	po->running	✗	✓	✓	UAF	✓	✓
CVE-2017-15951	✗	✗	✗	✗	✗	✗	✗
CVE-2017-17712	inet->hdrincl	✗	✓	✓	UBI	✗	✗
CVE-2017-2636	n_hdlc.tbuf	✗	✓	✓	Double-Free	✗	✓
CVE-2017-2671	sk	✗	✓	✓	Null-Dereference	✓	✗
CVE-2017-7533	file_name	✗	✓	✓	OOB	✗	✓
CVE-2019-11599	VMA	✗	✗	✗	UAF	✗	✗
CVE-2019-11815	tc->t_sock	✗	✓	✓	UAF	✓	✓
CVE-2019-18683	vb2_queue->bufs	✗	✓	✓	UAF	✓	✓
CVE-2019-6974	dev	✗	✓	✓	UAF	✗	✓
CVE-2021-26708	vsk->transport	✗	✓	✓	UAF	✓	✓
7341... [23]	udp_sk->sk_user_data	✗	✓	✓	UAF	✗	✓
4d33... [24]	sel_buffer	✓	✓	✓	UAF	✓	✓
62d9... [25]	usbhid->urbin	✗	✓	✓	UAF	✓	✓

Table 1. Cont.

CVE	Race Variable	Race Point		Timing Relationship	Triggering Type	Recognized Triggering Type	
		AA-SVF	Improved AA-ERACE			KASAN	ERACE
def3... [26]	✗	✗	✗	✗	PAGEFAULT	✗	✗
e010... [27]	prev->refs	✓	✓	✓	Double-Free	✗	✓
9824... [28]	sk->sk_dst_cache	✗	✓	✓	Double-Free	✗	✓
Overall	*	4	19	17	*	10	17

To evaluate the accuracy in identifying the race points, timing relationships of the race instructions, and triggering types, we compared the output results of ERACE with the publicly available vulnerability information. For vulnerabilities with less publicly available information, we manually analyzed the patch information and the related vulnerable code to confirm the validity. To show the effectiveness of ERACE in facilitating the exploitation of race vulnerabilities, we chose a corresponding exploitation method according to the triggering type and vulnerability context recorded by ERACE. The subsequent exploitation process mainly refers to the method introduced in previous articles [13–16]. There are two main methods for the final privilege escalation. One uses an ROP chain [29] to call `commit_creds(prepare_kernel_cred(0))` and the other uses arbitrary write operations to modify the credential structure of the process. If ERACE identifies the available checkpoints for the vulnerability, we can compare the impact of using checkpoints and not using checkpoints on the success rate of the vulnerability exploitation.

6.2. Accuracy

In this section, we evaluate the accuracy in identifying the race points, the timing relationships among the race instructions, and the triggering types (see Table 1 for the experimental results). In general, ERACE can effectively identify most race points, the timing relationships among the race instructions, and the triggering types.

ERACE identified a total of 19 vulnerabilities' race points. The experiments show that ERACE can effectively help researchers locate vulnerable points, that is, race points. Moreover, to verify whether relaxed alias analysis brings improvements, we compared the number of race points that can be identified with and without relaxed alias analysis. To ensure the accuracy of the results, the other conditions were unchanged, except for the alias analysis component in ERACE. AA-SVF indicates the use of alias analysis in SVF, and Improved AA-ERACE indicates the use of relaxed alias analysis. The experimental results show that without improving alias analysis, only 4 vulnerabilities' race points were identified, whereas 15 more vulnerabilities' race points were identified with relaxed alias analysis.

We analyzed the specific program context of each vulnerability's race point. The race points of CVE-2016-6516, CVE-2017-10661, and e010 [27] were located in the same function or on the same path, and the race variable of 4d33 [24] was on a stack so traditional alias analysis could easily identify these race points. The race points of the 15 vulnerabilities were located on different paths. That is to say, users must use different system call interfaces to trigger the execution of different race instructions. As long as the user passes in certain identical parameters, the race variables accessed by the race instruction satisfy the alias relationships. Traditional alias analysis cannot identify such aliases. Relaxed alias analysis will treat the same objects in different functions as the same memory. Although this will cause false positives, filtering with the collected dynamic execution paths can effectively reduce false positives.

We analyzed four cases where ERACE failed to identify the race points. CVE-2015-8550 is quite special. This vulnerability does not exist in the source code. It was introduced after a gcc compilation. The user-space data were accessed twice but the consistency of the data was not checked, resulting in control-flow hijacking. When compiled with Clang, there was

no such vulnerability so the race point could not be identified. The PoC of CVE-2017-15951 could not trigger kernel vulnerabilities so it was impossible to determine whether the vulnerability path was collected. Public information on CVE-2017-15951 is very vague and it cannot provide researchers with enough information to confirm its triggering type and root cause. CVE-2019-11599 is due to access competition between VMA virtual mapping spaces and not between the general memory variables so ERACE could not detect the race points. `def3` [26] could only trigger a page fault so we could not find its race points.

ERACE identified a total of 17 vulnerabilities' timing relationships among the race points, which can assist researchers in debugging. There were four vulnerabilities whose race points were not identified so the execution order of the race instructions could not be identified. CVE-2016-6516 belongs to the double-fetch vulnerability, whose user-space data were read twice, but the consistency of the data was not checked. The writing thread was located in the user layer but ERACE could only execute the kernel. Therefore, it could not identify the user-space competition and the instruction sequence when the vulnerability was triggered. The competition involved in CVE-2017-15265 was more complicated. After executing the suspicious three race points, we identified the relative instruction sequence but still could not stably trigger the vulnerability. After analyzing the source code, we found that only when the four program points in each thread were executed in a specific order could the vulnerability be stably triggered. Therefore, the execution method proposed in this paper could not identify the complete instruction order.

ERACE identified a total of 17 vulnerabilities' triggering types, which can assist researchers in choosing an appropriate exploitation method. We compared the recognition accuracy of KASAN and ERACE. KASAN falsely identified eight triggering types, whereas ERACE was able to correct the KASAN identification errors. Of the six vulnerabilities that ERACE failed to identify, four of them were not identified because the race-point detector did not identify the race point. The CVE-2017-2671 vulnerability could trigger only a null deference, which did not belong to the currently exploitable triggering type so ERACE failed to identify it. CVE-2017-17712 is a vulnerability caused by using uninitialized stack variables. Neither KASAN nor ERACE was able to identify it. After using STACKLEAK developed by Grsecurity/PaX [18], it was easy to eliminate this vulnerability so it cannot be exploited.

6.3. Effectiveness

This section mainly tests the effectiveness of ERACE in facilitating the process of exploiting the race vulnerability (see Table 2 for the experimental results). In general, ERACE can effectively help researchers generate exploits, and the identified checkpoints can increase the success rate of the privilege escalation.

Table 2. Effectiveness of ERACE. # of checkpoints indicates the number of checkpoints ERACE could find. We compare whether the exploit succeeded in escaping privilege by using and not using checkpoints. ✕ means it has no checkpoints or public exploit, or it fails to exploit the vulnerability. ✓ means it has public exploit or it succeeds in exploiting the vulnerability.

CVE	# of Checkpoints	Public Exploit	Generated Exploit	
			Without Checkpoints	Use Checkpoints
CVE-2015-8550	✕	✕	✕	✕
CVE-2016-6516	4	✕	✕	✕
CVE-2016-8655	22	✓	✕	✓
CVE-2017-10661	2	✕	✕	✓
CVE-2017-11176	35	✓	✕	✓

Table 2. Cont.

CVE	# of Checkpoints	Public Exploit	Generated Exploit	
			Without Checkpoint	Use Checkpoint
CVE-2017-15265	7	✗	✓	✓
CVE-2017-15649	22	✗	✗	✓
CVE-2017-15951	✗	✗	✗	✗
CVE-2017-17712	55	✗	✗	✗
CVE-2017-2636	4	✓	✗	✓
CVE-2017-2671	34	✗	✗	✗
CVE-2017-7533	5	✗	✗	✓
CVE-2019-11599	✗	✗	✗	✗
CVE-2019-11815	5	✗	✗	✗
CVE-2019-18683	6	✗	✓	✓
CVE-2019-6974	24	✗	✗	✓
CVE-2021-26708	1	✗	✓	✓
7341... [23]	3	✗	✓	✓
4d33... [24]	24	✗	✗	✗
62d9... [25]	10	✗	✗	✓
def3... [26]	22	✗	✗	✗
e010... [27]	4	✗	✓	✓
9824... [28]	5	✗	✗	✗
Overall	294	3	5	13

Among these 23 vulnerabilities, exploits of only 3 vulnerabilities were publicly available and ERACE helped 13 vulnerabilities to achieve exploitation. We found a total of 294 checkpoints in the modules where the 23 vulnerabilities were located, which can be used by researchers to check whether race conditions or heap spraying has been satisfied. Among the 13 successfully exploited vulnerabilities, we compared whether the vulnerabilities could be successfully exploited by using and not using checkpoints. We found that eight of them could not be successfully exploited without using checkpoints. This shows that it is necessary to consider the use of checkpoints when exploiting race vulnerabilities. If heap spraying is performed without confirming whether the race condition has been satisfied, it consumes a limited amount of heap-spraying space and time and also makes the system unstable. Then, if it goes to the next exploitation stage without confirming whether the heap spraying has succeeded, it will further reduce the exploitation success rate.

There were five vulnerabilities that were successfully exploited without using checkpoints, namely CVE-2017-15265, CVE-2019-18683, 2021-26708, 7341 [23], and e010 [27]. As fewer checkpoints were identified in the modules where the three vulnerabilities were located, no suitable checkpoints could help to exploit these three vulnerabilities. The main reason is that some kernel modules are implemented with less code and fewer functionalities and they contain fewer interfaces for users to access kernel data.

We analyzed 10 vulnerabilities that failed the exploitation. Four of them failed because the race point was not identified and the other six failed because their exploit primitives were restricted. For example, CVE-2016-6516 could only overwrite a continuous 12 bytes to 0, but modifying the credentials required 28 bytes to escalate privilege. Hence, we could not exploit the vulnerability at the time. We also made some interesting discoveries. The CVE-2016-8655

and CVE-2017-15649 vulnerabilities were both located in the `/net/packet/af_packet.c` file so the numbers of checkpoints were the same. CVE-2017-17712 was located in the `/net/ipv4` module, which was very large with many functions, so the number of checkpoints was the largest. The CVE-2017-11176 and CVE-2017-2671 vulnerabilities involved multiple modules so the numbers of checkpoints found were relatively large.

7. Related Works

As mentioned above, the objective of this paper is to assist in the process of exploiting kernel race vulnerabilities. Therefore, the related works are mainly divided into three categories, namely race vulnerability detection, automatic exploit generation in the user space, and kernel vulnerability exploitation. Below, we discuss the existing works in these three categories in detail.

Race vulnerability detection. There are a lot of studies on detecting race vulnerabilities, and feature modeling and dynamic detection techniques for race vulnerabilities are very helpful for race exploitation. These techniques are mainly divided into two categories: static analysis and dynamic analysis.

Refs. [30–32] adopted static analysis techniques. Among them, Relay [31] proposed the concept of a relative lockset, which makes the function summary independent of the calling context so that it can perform easy-to-parallel modular and bottom-up analyses. Ref. [30] was based on the Coccinelle engine and used static pattern matching to detect a special kind of data race, called a double-fetch vulnerability. It matched the kernel function that calls `copy_from_user()` or `get_user()` multiple times to read data from the user space at the same address and then manually analyzed these matched functions. Ref. [32] formalized the double-fetch vulnerability, of which the two read operations have a spatial overlap and the overlapping variables have control dependence or data dependence. Then, static analysis and symbolic execution were performed on the LLVM intermediate representation. The advantage of static analysis is that it is scientific in race vulnerability feature modeling and has great scalability. The disadvantage is that it has a high false-positive rate and relies heavily on manual confirmations.

Refs. [10,33–39] applied dynamic analysis techniques. BochsPwn [38] is a system-wide memory monitoring tool implemented on a Bochs $\times 86$ emulator, which is capable of detecting kernel double-fetch and information leaks. ThreadSanitizer [39] is an execution tool integrated into gcc [40] and clang [41] compilers that can detect multi-thread data race vulnerabilities at runtime. The goal of [33,34] was to detect concurrency vulnerabilities. The former proposed the coarse-grained interleaving hypothesis, which improves detection accuracy and reduces time costs. The latter defined relaxed exchangeable events, which can detect three types of concurrency vulnerabilities, the UAF, null-pointer-dereference, and double-free vulnerabilities. The innovation in [10,35] was to explore thread interleaving, which applies modified QEMU that can be used to set breakpoints to control thread interleaving. However, the former does not first determine the suspicious race instructions, resulting in a large search space. The latter proposed a deterministic thread interleaving technique, which modifies qemu and implements three hypercalls to schedule several virtual CPU cores to run and interrupt. However, it can only control thread interleaving on different CPU cores, whereas our exploit must run on one specific CPU to improve the success rate of heap spraying. Thus, the hypervisor-based solution does not promote our exploit debugging. The innovation in [30,37] was to improve the traditional fuzzing technique to adapt to detecting race vulnerabilities. Krace [36] introduced a new coverage tracking metric, namely alias coverage. MUZZ [37] introduced a new instrumentation mechanism and optimized the process of seed selection. The mechanism gives priority to the seed that can trigger new code coverage or a new thread context. The advantage of dynamic analysis is that it can directly generate input that triggers vulnerabilities, but the disadvantage is that it has poor scalability and slow speeds.

Automatic exploit generation. Kernel vulnerabilities are more difficult to exploit than vulnerabilities in the user space. This is because the kernel environment is more complicated

and requires more system calls to complete the exploitation. However, the research on automatic exploit generation originates from vulnerabilities in the user space and the techniques used are similar.

APEG [42] studied the automatic exploitation of real software vulnerabilities for the first time, but APEG needs software that contains vulnerabilities and a relatively patched version. Exploits generated by APEG can only trigger vulnerabilities. AEG [43] and Mayhem [44] are both end-to-end systems that can fully exploit vulnerabilities. The former relies on the source code and the latter only requires binary files. The latter applies concolic symbolic execution and index-based memory modeling. FLOWSTITCH [45] first proposed a model based on data-flow stitching, which can generate valid inputs to bypass DEP, CFI, and even ASLR mitigations. The Q [46] scheme is based on ROP and uses a small amount of unrandomized code to automatically generate an ROP chain, which can effectively bypass W \oplus X and ASLR mitigations. For heap vulnerabilities, one of the most difficult vulnerabilities to exploit, [47–50] each proposed their own solutions. Revery [47] is able to explore exploitable states that are different from the crash path based on the PoC. HeapHopper [48] is based on model checking and symbolic execution and it can automatically analyze the exploitability of the heap implementation in the case of memory corruption. SHRIKE [49] can automatically arrange a heap layout on the PHP interpreter and perform control-flow hijacking. ARCHEAP [50] can systematically search undeveloped heap exploit primitives without considering their underlying implementations.

Kernel vulnerability exploitation. This paper refers to previous research on two aspects, which are not the focus of this paper. The first bypasses mainstream mitigation mechanisms such as SMAP, SMEP, and KASLR [19,51,52]. The second applies the follow-up exploitation stage after determining the triggering type of the race vulnerability [13–16,53–55]. Next, we introduce the existing techniques for kernel vulnerabilities.

Refs. [14,53,54] each proposed new exploitation techniques, among which KEPLER [14] discovered a new ROP construction method for vulnerabilities that can only hijack the control flow once. Ref. [53] used the shared space physmap between the kernel space and user space to bypass SMAP and SMEP mitigations. Ref. [54] proposed a new kernel heap-spraying mechanism for when the physmap page is not executable. Refs. [19,51,52] studied how to bypass the kernel address space layout randomization. ELOISE [19] used an elastic object in the kernel. Refs. [51,52] used the side-channel technique to leak the kernel address. Ref. [55] proposed a deterministic stack-spraying technique and memory-consuming stack-spraying technique to control the uninitialized variables on the kernel stack. FUZE [13] is a framework for facilitating a kernel UAF exploitation and adopts the kernel fuzzing and symbolic execution techniques to explore system calls that are useful for UAF exploitation and mitigation bypassing. SLAKE [15] modeled three kernel vulnerabilities: UAF, double-free, and OOB vulnerabilities. It can identify kernel objects and the corresponding system calls that are useful for heap spraying, and reorganize slabs to obtain the expected slab layout. KOUBE [16] summarized the challenges in exploiting OOB access vulnerabilities. A new technique for capability-oriented fuzzing was proposed to search for certain capabilities. Symbolic tracing was used to automatically analyze OOB vulnerabilities and identify the appropriate target objects.

The process of locating race points in this paper is similar to that of FUZE [13], but FUZE aims at exploiting UAF vulnerabilities. Memory allocation points and release points can be located according to the KASAN warning report. However, there is no existing tool for locating race points. EXPRACE [56] uses an interrupt mechanism to slow down the execution of the specified core to improve the success rate of competition for multi-variable race vulnerabilities, whereas we use program analysis techniques to assist in the exploit generation of general race vulnerabilities. The problems we face are different.

8. Conclusions

In this paper, we show that it is challenging to exploit kernel race vulnerabilities. Aiming to address the problem of the exploiting process taking considerable time and requiring extremely high professional knowledge, we developed ERACE, an effective framework for facilitating the process of exploiting kernel race vulnerabilities. To be more specific, we introduce the implementation of ERACE. The race-point detector uses low-level virtual machine (LLVM) static analysis to search for all possible race instruction pairs and `kcov` [17] to dynamically record the execution path of the PoC. The vulnerability analyzer uses the instrumentation mechanism to identify the timing relationships among the race instructions, uses static analysis to identify the triggering type of the vulnerability, and records the vulnerability context. The checkpoint detector uses LLVM static analysis to identify the checkpoints and records the system calls to which the checkpoints belong. Finally, based on the vulnerability information collected above, we finish generating the exploit.

We choose 23 real-world kernel race vulnerabilities to demonstrate the utility of ERACE. We successfully detect the race points of 19 vulnerabilities, the timing relationships among the race instructions, and the triggering types of 17 vulnerabilities, and assist in generating exploits for 13 vulnerabilities. Based on these findings, we can safely conclude that ERACE can effectively help security researchers simplify the analysis process of kernel race vulnerabilities, select appropriate exploitation methods, and use checkpoints to increase the success rates of exploitations. Furthermore, the relaxed alias analysis also helps to identify more race candidates, which may be beneficial for detecting race vulnerabilities.

In the future, we will extend the functionality of ERACE to adapt to more scenarios. First, when identifying race points, we did not perform lockset-based analysis on the alias variables, which may have led to false positives. ERACE uses dynamic execution paths to filter out unexecuted race instructions to reduce false positives. By performing lockset-based analysis, we can exclude race instructions with correct locking, further reduce false alarms, and reduce manual analysis. For example, refs. [57–59] each applied lockset-based analysis to detect data race vulnerabilities. Second, ERACE is only suitable for single-variable race vulnerabilities. For multi-variable race vulnerabilities, the execution order of the race instructions is so complicated that it was impossible for ERACE to identify the exact timing relationships. Next, we will improve ERACE to enable it to analyze multi-variable race vulnerabilities.

Author Contributions: Conceptualization, D.L. and P.W.; methodology, D.L.; software, D.L.; validation, P.W., X.Z. and B.W.; formal analysis, B.W.; investigation, D.L.; resources, X.Z.; data curation, X.Z.; writing—original draft preparation, D.L.; writing—review and editing, D.L.; visualization, P.W.; supervision, X.Z.; project administration, X.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Natural Science Foundation of China (61902412, 61902405, 62272472), the National High-Level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013), the Natural Science Foundation of Hunan Province of China under grant no. 2021JJ40692, and the Research Project of the National University of Defense Technology (ZK20-17, ZK20-09).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Corporation, T.M. Common Vulnerability and Exposures. 2021. Available online: <https://cve.mitre.org/cve/> (accessed on 17 November 2022).
2. Google. Syzbot: Google Continuously Fuzzing The Linux Kernel. 2018. Available online: <https://syzkaller.appspot.com/upstream> (accessed on 17 November 2022).

3. CWE. CWE-787: Out-of-Bounds Write. 2021. Available online: <https://cwe.mitre.org/data/definitions/787.html> (accessed on 17 November 2022).
4. CWE. CWE-416: Use After Free. 2021. Available online: <https://cwe.mitre.org/data/definitions/416.html> (accessed on 17 November 2022).
5. CWE. CWE-415: Double Free. 2021. Available online: <https://cwe.mitre.org/data/definitions/415.html> (accessed on 17 November 2022).
6. Corporation, T.M. CVE-2016-8655. 2016. Available online: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655> (accessed on 17 November 2022).
7. Corporation, T.M. CVE-2019-11815. 2019. Available online: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11815> (accessed on 17 November 2022).
8. Guo, P.J.; Zimmermann, T.; Nagappan, N.; Murphy, B. Characterizing and predicting which bugs get fixed. In Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering, Cape Town, South Africa, 1–8 May 2010; Volume 10.
9. Penalver, C.M. How to Triage Bugs. 2016. Available online: <https://wiki.ubuntu.com/Bugs/Importance> (accessed on 17 November 2022).
10. Jeong, D.R.; Kim, K.; Shivakumar, B.; Lee, B.; Shin, I. Razer: Finding kernel race bugs through fuzzing. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–23 May 2019; pp. 754–768.
11. Google. The Kernel Address Sanitizer (Kasan). 2017. Available online: <https://github.com/google/kasan> (accessed on 17 November 2022).
12. Google. Kernel Concurrency Sanitizer (KCSAN). 2020. Available online: <https://github.com/google/ktsan/wiki/KCSAN> (accessed on 17 November 2022).
13. Wu, W.; Chen, Y.; Xu, J.; Xing, X.; Gong, X.; Zou, W. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 781–797.
14. Wu, W.; Chen, Y.; Xing, X.; Zou, W. KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), San Antonio, TX, USA, 26–29 January 2019; pp. 1187–1204.
15. Chen, Y.; Xing, X. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the Linux kernel. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2019; pp. 1707–1722.
16. Chen, W.; Zou, X.; Li, G.; Qian, Z. KOUBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Washington, DC, USA, 13–17 August 2020; pp. 1093–1110.
17. Yamada, M.; Nikula, J. Kcov: Code Coverage for Fuzzing. 2019. Available online: <https://github.com/torvalds/linux/blob/master/Documentation/devtools/kcov.rst> (accessed on 17 November 2022).
18. Popov, A. How STACKLEAK Improves Linux Kernel Security. 2018. Available online: <https://a13xp0p0v.github.io/2018/11/04/stackleak.html> (accessed on 17 November 2022).
19. Chen, Y.; Lin, Z.; Xing, X. A systematic study of elastic objects in kernel exploitation. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual, 9–13 November 2020; pp. 1165–1184.
20. Sui, Y.; Xue, J. SVF: Interprocedural static value-flow analysis in LLVM. In Proceedings of the 25th International Conference on Compiler Construction, Barcelona, Spain, 17–18 March 2016; pp. 265–266.
21. Lu, K.; Pakki, A.; Wu, Q. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), San Antonio, TX, USA, 26–29 January 2019; pp. 1769–1786.
22. Gens, D.; Schmitt, S.; Davi, L.; Sadeghi, A.R. K-Miner: Uncovering Memory Corruption in Linux. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2018, San Diego, CA, USA, 18–21 February 2018.
23. Syzbot. General Protection Fault in skb_queue_tail. 2019. Available online: <https://syzkaller.appspot.com/bug?id=73411c725071d5153a0080116c05fc68e0ea195a> (accessed on 17 November 2022).
24. Syzbot. KASAN: Use-after-Free Read in n_tty_Receive_buf_Common. 2020. Available online: <https://syzkaller.appspot.com/bug?id=4d331631d20a7ffcb1ef5b77eb0d91cfaad1e7> (accessed on 17 November 2022).
25. Syzbot. KASAN: Use-after-free Read in usbhid_Close (3). 2020. Available online: <https://syzkaller.appspot.com/bug?id=62d96ed19ad346333a20b6cac86441bfff7c4719> (accessed on 17 November 2022).
26. Syzbot. General Protection Fault in Packet_Lookup_Frame. 2019. Available online: <https://syzkaller.appspot.com/bug?id=def39b541a4d5fdd258ec710d2a159010c8601f3> (accessed on 17 November 2022).
27. Syzbot. WARNING in io_Link_Timeout_fn. 2021. Available online: <https://syzkaller.appspot.com/bug?id=e0104ebcf6c138cf60f0af0ee1a0772016f5aa33> (accessed on 17 November 2022).
28. Syzbot. KASAN: Use-after-Free Write in ip6_Hold_Safe. 2018. Available online: <https://syzkaller.appspot.com/bugid=9824322d106666c1f8eb8f329b91dea75b390> (accessed on 17 November 2022).
29. Nikolenko, V. Linux Kernel ROP—Ropping Your Way to # (Part 1). 2016. Available online: <https://www.trustwave.com/en-us/resources/blogs/spiderlabsblog/linux-kernel-rop-roping-your-way-to-part-1/> (accessed on 17 November 2022).

30. Wang, P.; Krinke, J.; Lu, K.; Li, G.; Dodier-Lazaro, S. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; pp. 1–16.
31. Voung, J.W.; Jhala, R.; Lerner, S. RELAY: Static race detection on millions of lines of code. In Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, Sacramento, CA, USA, 8–13 November 2007; pp. 205–214.
32. Xu, M.; Qian, C.; Lu, K.; Backes, M.; Kim, T. Precise and scalable detection of double-fetch bugs in OS kernels. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 21–23 May 2018; pp. 661–678.
33. Cai, Y.; Zhang, J.; Cao, L.; Liu, J. A deployable sampling strategy for data race detection. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; pp. 810–821.
34. Cai, Y.; Zhu, B.; Meng, R.; Yun, H.; He, L.; Su, P.; Liang, B. Detecting concurrency memory corruption vulnerabilities. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; pp. 706–717.
35. Fonseca, P.; Rodrigues, R.; Brandenburg, B.B. SKI: Exposing kernel concurrency bugs through systematic schedule exploration. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), Broomfield, CO, USA, 6–8 October 2014; pp. 415–431.
36. Xu, M.; Kashyap, S.; Zhao, H.; Kim, T. Krace: Data race fuzzing for kernel file systems. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1643–1660.
37. Chen, H.; Guo, S.; Xue, Y.; Sui, Y.; Zhang, C.; Li, Y.; Wang, H.; Liu, Y. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 12–14 August 2020; pp. 2325–2342.
38. Googleprojectzero. Bochspxn. 2017. Available online: <https://github.com/googleprojectzero/bochspxn> (accessed on 17 November 2022).
39. Serebryany, K.; Iskhodzhanov, T. ThreadSanitizer: Data race detection in practice. In Proceedings of the Workshop on Binary Instrumentation and Applications, New York, NY, USA, 12 December 2009; pp. 62–71.
40. Stallman, R. GCC, the GNU Compiler Collection. 1987. Available online: <https://www.gnu.org/software/gcc/> (accessed on 17 November 2022).
41. Group, L.D. Clang: A C Language Family Frontend for LLVM. 2007. Available online: <https://clang.llvm.org/> (accessed on 17 November 2022).
42. Brumley, D.; Poosankam, P.; Song, D.; Zheng, J. Automatic patch-based exploit generation is possible: Techniques and implications. In Proceedings of the 2008 IEEE Symposium on Security and Privacy (sp 2008), Oakland, CA, USA, 18–19 May 2008; pp. 143–157.
43. Avgerinos, T.; Cha, S.; Hao, B.; Brumley, D. AEG: Automatic Exploit Generation. *Commun. ACM* **2014**, *57*, 74–84. [\[CrossRef\]](#)
44. Cha, S.K.; Avgerinos, T.; Rebert, A.; Brumley, D. Unleashing mayhem on binary code. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012; pp. 380–394.
45. Hu, H.; Chua, Z.L.; Adrian, S.; Saxena, P.; Liang, Z. Automatic generation of data-oriented exploits. In Proceedings of the 24th USENIX Security Symposium (USENIX Security 15), Washington, DC, USA, 12–14 August 2015; pp. 177–192.
46. Schwartz, E.J.; Avgerinos, T.; Brumley, D. Q: Exploit hardening made easy. In Proceedings of the USENIX Security Symposium, San Francisco, CA, USA, 8–12 August 2011; Volume 10.
47. Wang, Y.; Zhang, C.; Xiang, X.; Zhao, Z.; Li, W.; Gong, X.; Liu, B.; Chen, K.; Zou, W. Revery: From proof-of-concept to exploitable. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 1914–1927.
48. Eckert, M.; Bianchi, A.; Wang, R.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Heaphopper: Bringing bounded model checking to heap implementation security. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 99–116.
49. Heelan, S.; Melham, T.; Kroening, D. Automatic heap layout manipulation for exploitation. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Berkeley, CA, USA, 15–17 August 2018; pp. 763–779.
50. Yun, I.; Kapil, D.; Kim, T. Automatic techniques to systematically discover new heap exploitation primitives. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 12–14 August 2020; pp. 1111–1128.
51. Gruss, D.; Maurice, C.; Fogh, A.; Lipp, M.; Mangard, S. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 August 2016; pp. 368–379.
52. Jang, Y.; Lee, S.; Kim, T. Breaking kernel address space layout randomization with intel tsx. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 August 2016; pp. 380–392.
53. Kemerlis, V.P.; Polychronakis, M.; Keromytis, A.D. Ret2dir: Rethinking kernel isolation. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USA, 20–22 August 2014; pp. 957–972.
54. Xu, W.; Li, J.; Shu, J.; Yang, W.; Xie, T.; Zhang, Y.; Gu, D. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 414–425.

55. Lu, K.; Walter, M.T.; Pfaff, D.; Nümberger, S.; Lee, W.; Backes, M. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In Proceedings of the NDSS, San Diego, CA, USA, 26 February–1 March 2017.
56. Lee, Y.; Min, C.; Lee, B. EXPRACE: Exploiting kernel races through raising interrupts. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Virtual, 11–13 August 2021.
57. Chen, Q.L.; Bai, J.J.; Jiang, Z.M.; Lawall, J.; Hu, S.M. Detecting data races caused by inconsistent lock protection in device drivers. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering (SANER), Hangzhou, China, 24–27 February 2019; pp. 366–376.
58. Savage, S.; Burrows, M.; Nelson, G.; Sobalvarro, P.; Anderson, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. (TOCS)* **1997**, *15*, 391–411.
59. Engler, D.; Ashcraft, K. RacerX: Effective, static detection of race conditions and deadlocks. *ACM SIGOPS Oper. Syst. Rev.* **2003**, *37*, 237–252.