RingGuard: Guard io_uring with eBPF

Wanning He†
Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
hewn2019@mail.sustech.edu.cn

Hongyi Lu‡
Department of Computer Science and Engineering, Southern University of Science and Technology, China
luhy2017@mail.sustech.edu.cn

Fengwei Zhang§¶
Shenzhen Key Laboratory of Safety and Security for Next Generation of Industrial Internet, Southern University of Science and Technology, China
zhangfw@sustech.edu.cn

Shuai Wang¶
Department of Computer Science and Engineering, Hong Kong University of Science and Technology, China
shuaiw@cse.ust.hk

1 INTRODUCTION

Asynchronous I/O significantly improves the efficiency of I/O-bound applications. Unlike synchronous I/O, which forces applications to block on time-consuming I/O operations, asynchronous I/O allows them to launch non-blocking I/O requests and be notified upon completion. At first, the asynchronous I/O of Linux is provided by aio [1], which does not support network sockets and offers only non-buffered direct I/O. These limitations significantly impede the development of asynchronous applications on Linux. Fortunately, a new asynchronous I/O interface named io_uring [16] was introduced to the kernel in 2019. This new interface unifies the previously divergent asynchronous I/O models adopted in the kernel and also overcomes the drawbacks of aio.

Specifically, io_uring defines two queues (submission queue and completion queue) to share data between applications and kernel. User applications first submit their I/O requests to the submission queue, which are then picked up and processed by the kernel. Upon completion, the kernel then pushes the responses of these requests back to the completion queue for user applications. This conceptual model brings io_uring encouraging flexibility, extensibility, and performance. The request submission of io_uring is not limited to simple file I/O operations (e.g., read). It also supports network operations like send and even device-specific operations such as ioctl [4]. Moreover, the shared queues largely reduce the number of costly context switches.

However, as pointed out by recent studies [9, 10, 12], this flexibility comes at the cost of security. The requests submitted to the queue of io_uring are not regulated by security mechanisms such as seccomp [6], even though these requests represent a substantial subset of I/O-related system calls. This loophole clearly allows certain applications to abuse io_uring to bypass these security mechanisms and causes potential security breaches. In addition, as a fast-developing feature, vulnerabilities are frequently discovered in io_uring. These vulnerabilities can lead to privilege escalation, memory corruption, and denial of service, thus posing a serious security threat to the kernel.

Given the increasing security threat of io_uring, we seek a reliable while flexible way of auditing the requests submitted via io_uring. In particular, we leverage Berkeley Packet Filter (BPF) [2]. BPF was originally introduced as a kernel infrastructure just for packet filtering [20], but soon became a powerful yet secure way...
to extend kernel functionality. To clarify, there are two variants of BPF, classic BPF (cBPF) and extended BPF (eBPF), and RingGuard is based on eBPF. eBPF allows a set of user-supplied programs to be attached to certain kernel hookpoints to extend kernel functionality. Moreover, an eBPF verifier [31] is adopted to perform static analysis to ensure the safety of these user-supplied programs. With the aid of eBPF, we present RingGuard, where users can supply their own eBPF programs to audit the submitted I/O requests inside io_uring or patch vulnerabilities if there is not yet an official patch for io_uring vulnerabilities.

Despite its encouraging potential, using eBPF on io_uring is not a trivial task. RingGuard is specifically designed to overcome two major technical obstacles. First, since asynchronous I/O is designed for performance in the first place, RingGuard should be sufficiently efficient and not impede the performance benefit brought by io_uring. In our experiments, RingGuard incurs limited overhead to io_uring in most cases. However, when many individual requests are submitted via io_uring, the overhead becomes noticeable due to the repetitive initialization and destruction of eBPF programs. To solve this hurdle, we propose a batching scheme that packs multiple requests together for auditing to reduce these costs. Second, as a kernel facility, RingGuard needs to ensure the security of the eBPF programs it inserts. To achieve this goal, we extend the existing eBPF verifier to support the verification of RingGuard eBPF programs.

To evaluate the overhead of RingGuard, we first examine the performance penalty of RingGuard under typical use cases of io_uring, such as file I/O. Second, we conduct thorough experiments to show the performance improvement brought by our batching scheme. Our experiments show that its worst-case performance overhead in all these experiments is 25%, and our batching scheme brought a 12% performance improvement. We also evaluate RingGuard’s security benefit by carefully analyzing all recent CVEs of io_uring. Our analysis and evaluation results show that they can all be effectively patched with RingGuard.

To summarize, we make the following contributions.

- This paper for the first time proposes addressing io_uring security risks with the use of eBPF programs.
- RingGuard is specifically designed to achieve two key properties: an efficient auditing process and the security of its eBPF programs.
- We implement a prototype of RingGuard on the v5.12 Linux kernel and conduct a thorough evaluation of its performance overhead and security benefits.

2 BACKGROUND AND RELATED WORK

2.1 Asynchronous I/O by io_uring

The io_uring subsystem is an asynchronous I/O interface for Linux introduced in 2019 [16]. The main components of io_uring are two queues serving as the communication channels to transmit I/O requests and their responses. This pair of queues are dubbed as submission queue and completion queue. To avoid expensive memory copying and context switching, both queues are mapped as shared memory between user space and kernel space.

As illustrated in Fig. 1, the overall workflow of io_uring can be divided into four steps:

1. User program launches an I/O request by submitting its operation type (e.g., read) and parameters (e.g., source and destination) to the tail of submission queue.
2. Kernel fetches the request from the head of submission queue and executes the corresponding operation.
3. When the operation completes, kernel puts its result to the tail of completion queue.
4. In the end, user program obtains the result of the I/O request from the head of completion queue. Note that the requests of io_uring normally have their equivalent system calls. For example, a request with IORING_OP_READ is equivalent to a read system call and is processed by the same system call handler.

**Operation Restriction.** To clarify, io_uring also comes with a built-in security policy [8]. The owner can specify a whitelist of permitted operations for an io_uring instance. Operations are checked based on their types, flags, and the file descriptors to be operated on. However, recent CVEs have demonstrated that attackers can still exploit certain I/O request parameters to launch attacks, sometimes combining multiple operations (as shown in Sec. 4.2). Thus, after a thorough analysis, we find that this security mechanism is insufficient in defending against most io_uring exploits. Furthermore, the whitelist approach may reject entire sets of operations to prevent such attacks, which results in the loss of functionality for many legitimate requests with valid parameters. In contrast, RingGuard is designed to address these limitations by providing flexible filtering policies with the help of eBPF programs.

2.2 Extended Berkeley Packet Filter (eBPF)

The eBPF subsystem enables developers to run customized programs to extend kernel functionalities. Fig. 2 depicts an overview of how eBPF programs [2] extend kernel functionalities. As shown in the right part of Fig. 2, the Linux kernel has predefined a set of BPF hookpoints, each of which can be attached with eBPF programs of a specific type. For example, a socket hookpoint can only be attached with SOCKET_FILTER eBPF programs. These hookpoints, depending on their locations, offer extensibility to different kernel functions. In this specific case of Fig. 2, an eBPF program is attached to the io_uring hookpoint to log and filter out suspicious I/O requests in the submission queue. Moreover, depending on the type of eBPF program, it can usually call a set of relevant helper functions and utilize BPF maps\(^1\) for data storage. For example, the eBPF program in Fig. 2 invokes dequeue_rreq helper to dequeue an I/O request from the submission queue, logs relevant information into a BPF map, and submits it to the kernel for further processing if it is safe.

\(^1\)BPF maps are data structures that are specific to eBPF programs. Depending on implementation, they can be divided into array maps and hash maps.
2.3 Related Work

2.3.1 System Call Filtering. Security Computing Mode (seccomp) is a security facility in the Linux kernel [6]. It aims to restrict the system calls a program can use and thus shares a similar goal as RingGuard. It can also be augmented with eBPF programs (i.e., seccomp-BPF [5]) to flexibly filter suspicious system calls launched by a process. However, it does not support filtering the system calls (i.e., I/O requests) submitted via io_uring, which has been regarded as a way to bypass seccomp for a long time [12].

2.3.2 BPF Security. BPF programs are widely used as a security enforcement for the Linux kernel. There is a line of works [7, 11, 13–15, 18, 29, 30, 32, 33] that utilize eBPF programs for security purposes, such as memory protection [33], DDoS mitigation [7], and access control [11, 13–15, 18]. Despite that we all leverage eBPF, the goals of these works are orthogonal to RingGuard. For instance, HotBPF [33] is a framework aiming to detect and isolate memory corruption on the fly using eBPF programs. Therefore, it focuses on a completely different subsystem (i.e., memory management) and solves a distinct set of domain-specific challenges compared to RingGuard. Similarly, Linux Security Module (LSM) [32] also provides its own set of eBPF hooks (i.e., LSM-BPF) that offers access control to various Linux subsystems. However, LSM-BPF only allows existing modules to be replaced with eBPF programs and does not offer new hooks within io_uring.

In the meantime, there are works [17, 19] focusing on enhancing the security of BPF itself. MOAT [19] leverages hardware features to prevent BPF from being exploited. Jia et al. [17] propose secure BPF using memory-safe language such as Rust. We deem these mechanisms can be used with RingGuard to further enhance security.

3 DESIGN

3.1 Overview

RingGuard leverages eBPF programs to audit and log I/O requests which are submitted via io_uring. Fig. 3 shows the overall workflow of RingGuard. In the beginning, the owner of the io_uring instance (Owner in Fig. 3) configures and deploys the request auditor and logger to the targeted io_uring in the form of eBPF programs. When a user program utilizes this io_uring to submit I/O requests, the auditor will check and modify (if necessary) these requests before they are processed by the kernel. In addition, the logger here is responsible for recording these requests for better observability.

3.2 Extending eBPF/io_uring

In this section, we introduce how we extend the current eBPF and io_uring subsystems to support RingGuard.
To comprehensively evaluate RingGuard’s performance, we conduct performance evaluations on different submission strategies. Our preliminary study shows that submitting numerous I/O requests individually results in significant overhead. As depicted in Table 2, submitting 512 requests one by one (512 × 1) takes over 7× longer than submitting them all at once (1 × 512). We presume this is due to the repetitive construction and destruction of eBPF runtime contexts as well as the overhead of context switching.

Based on this observation, we design a request batching scheme for RingGuard. When attaching a RingGuard eBPF program to the io_uring, the user may optionally supply a threshold and timeout value for RingGuard. threshold specifies the minimum number of I/O requests inside the submission queue to trigger RingGuard execution, which avoids the overhead of repeatedly setting up and tearing down the eBPF programs. timeout, on the other hand, defines the maximum latency RingGuard will wait if there are not enough I/O requests in the submission queue. Note that this waiting process is asynchronous and thus does not block other kernel tasks. In Sec. 4.1.2, we evaluate the performance benefits of different threshold and timeout values.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>1 × 512</th>
<th>4 × 128</th>
<th>16 × 32</th>
<th>64 × 8</th>
<th>512 × 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (ms)</td>
<td>20.3</td>
<td>21.5</td>
<td>24.2</td>
<td>35.8</td>
<td>150.8</td>
</tr>
</tbody>
</table>

4 IMPLEMENTATION & EVALUATION

RingGuard adds 273 LoC to the v5.12 Linux kernel. The small codebase indicates it can be easily customized and adapted to various use cases.

In this section, we evaluate the performance and security of RingGuard. In Sec. 4.1, we assess the overhead of RingGuard. In Sec. 4.2, we demonstrate its security benefits by analyzing recent vulnerabilities in io_uring and showing that they can indeed be patched using RingGuard. All experiments are conducted on an AMD 5800 8-core CPU. Each setup is run ten times to eliminate randomness. Time statistics are measured using the clock_gettime system call [3].

4.1 Performance Evaluation

4.1.1 Submission Latency. Note that RingGuard eBPF programs are only launched when the user program submits I/O requests to the kernel. Therefore, the main slowdown caused by RingGuard is reflected in the submission latency, which represents the time interval between the user enqueuing a request to the io_uring submission queue and the kernel fetching this request from the queue. To evaluate this overhead, we measure the average submission time of IORING_OP_NOP requests, which do not perform any actual I/O operations. These experiments involve RingGuard eBPF programs extracting requests from the submission queue and directly submitting them all to the kernel without any examination or modification. All requests are submitted to io_uring simultaneously, allowing
the eBPF program to run once and check all the requests. Fig. 6 shows the average submission latency of each request. From our observation, RingGuard imposes a moderate overhead of about 22% even in the worst case, which occurs in submitting 64 requests at once. We report that RingGuard’s slowdown decreases as the number of requests increases.

![Figure 6: Latency of submitting one request.](image)

4.1.2 Batching Optimization. Despite that RingGuard already shows a reasonable overhead in Sec. 4.1.1, there still exists a huge performance degradation when these I/O requests are submitted individually instead of simultaneously (see Sec. 3.4 for details). To evaluate the performance benefits of the batching mechanism proposed in Sec. 3.4, we measure the total submission latency of 512 requests under different configurations. The threshold values tested are 8, 32, 128, 512 and the timeout values tested are 2, 5, 10, 15 ms, respectively. All these requests are submitted one by one to better reflect the performance degradation. We report the results in Fig. 7 and Table 3.

Fig. 7 shows the time taken by io_uring to process 512 NOP requests under different configurations. We notice that our batching optimization improves the performance of io_uring by nearly 17% in this case. The improved processing time is even lower than that of a vanilla io_uring implementation (i.e., without RingGuard). We believe this is because batching reduces the internal processing time of the io_uring subsystem for these requests. We also observe that though with slight differences, the improvement brought by the batching mechanism is not affected by the exact threshold value. Instead, this optimization brings a notable improvement as long as the threshold is reasonable.

The complete experiment results are listed in Table 3. It can be seen that the timeout value also does not affect the performance by much as it falls in a reasonable range.

![Figure 7: Latency of submitting 512 requests with timeout = 2 ms. The statistic of vanilla io_uring is marked by the orange line.](image)

4.1.3 File Copying. To evaluate RingGuard overhead in the file I/O scenario, we compare three file copying test suites using synchronous I/O (system calls), io_uring, and io_uring with RingGuard. These programs utilize readv and writev system calls or equivalent io_uring operations to perform copying tasks. The submission queue depth of io_uring is set large enough to accommodate all the read or write operations required in these experiments so that the user can queue and submit them all at once. This setting is intended to minimize the number of system calls for request submission. Since this case does not involve many separate I/O requests, the batching optimization is disabled. As shown in Fig. 8, the worst overhead incurred by RingGuard is merely 7.3%. Moreover, this overhead gradually decreases as the copied file size increases, which drops to merely 1% when the copied file is 400 KB. We believe that handling I/O operations takes the largest proportion in file copying instead of the execution of eBPF programs. Besides, RingGuard does not impede the performance improvement brought by io_uring. In all test cases, io_uring with RingGuard is still about 2x more efficient than synchronous I/O and has comparable performance to vanilla io_uring.

![Figure 8: Latency of copying file.](image)

4.2 Security Evaluation

RingGuard reduces the attack interfaces in the io_uring subsystem by enforcing a customized auditing process for I/O requests submitted via io_uring. In io_uring, the most prevalent attack pattern is I/O request abuse, where attackers carefully prepare the parameter values of I/O requests to exploit io_uring vulnerabilities. This type of attack can be effectively prevented by RingGuard, which enforces parameter checking on all requests before they are passed to the

<table>
<thead>
<tr>
<th>Table 3: Latency of submitting 512 requests with different request batching configurations (ms).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

To evaluate the performance benefits of the batching mechanism, Table 3: Latency of submitting 512 requests with different request batching configurations. We notice that our batching optimization improves the performance of io_uring by nearly 17% in this case. The improved processing time is even lower than that of a vanilla io_uring implementation (i.e., without RingGuard). We believe this is because batching reduces the internal processing time of the io_uring subsystem for these requests. We also observe that though with slight differences, the improvement brought by the batching mechanism is not affected by the exact threshold value. Instead, this optimization brings a notable improvement as long as the threshold is reasonable.

The complete experiment results are listed in Table 3. It can be seen that the timeout value also does not affect the performance by much as it falls in a reasonable range.

<table>
<thead>
<tr>
<th>Table 3: Latency of submitting 512 requests with different request batching configurations (ms).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

Figure 7: Latency of submitting 512 requests with timeout = 2 ms. The statistic of vanilla io_uring is marked by the orange line.

4.1.3 File Copying. To evaluate RingGuard overhead in the file I/O scenario, we compare three file copying test suites using synchronous I/O (system calls), io_uring, and io_uring with RingGuard. These programs utilize readv and writev system calls or equivalent io_uring operations to perform copying tasks. The submission queue depth of io_uring is set large enough to accommodate all the read or write operations required in these experiments so that the user can queue and submit them all at once. This setting is intended to minimize the number of system calls for request submission. Since this case does not involve many separate I/O requests, the batching optimization is disabled. As shown in Fig. 8, the worst overhead incurred by RingGuard is merely 7.3%. Moreover, this overhead gradually decreases as the copied file size increases, which drops to merely 1% when the copied file is 400 KB. We believe that handling I/O operations takes the largest proportion in file copying instead of the execution of eBPF programs. Besides, RingGuard does not impede the performance improvement brought by io_uring. In all test cases, io_uring with RingGuard is still about 2x more efficient than synchronous I/O and has comparable performance to vanilla io_uring.

4.2 Security Evaluation

RingGuard reduces the attack interfaces in the io_uring subsystem by enforcing a customized auditing process for I/O requests submitted via io_uring. In io_uring, the most prevalent attack pattern is I/O request abuse, where attackers carefully prepare the parameter values of I/O requests to exploit io_uring vulnerabilities. This type of attack can be effectively prevented by RingGuard, which enforces parameter checking on all requests before they are passed to the
kernel for execution. With appropriate auditing rules, RingGuard can secure io_uring operations from being abused.

We analyze eight io_uring-related CVEs which fall within the application scope of RingGuard. These vulnerabilities typically lead to local privilege escalation and thus pose a severe security threat to the kernel. We report that all of them can be patched with RingGuard. Their CVE IDs and mitigations are listed in Table 4.

<table>
<thead>
<tr>
<th>CVE ID</th>
<th>Auditing Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020-29534</td>
<td>Check the provided file descriptor of FILES_UPDATE.</td>
</tr>
<tr>
<td>2021-2391</td>
<td>Check the buffer length of PROVIDE_BUFFERS.</td>
</tr>
<tr>
<td>2021-20226</td>
<td>Validate the existence of provided file in CLOSE.</td>
</tr>
<tr>
<td>2022-1976</td>
<td>Block a specific string of I/O requests.</td>
</tr>
<tr>
<td>2022-2327</td>
<td>Check the work flags of multiple I/O requests.</td>
</tr>
<tr>
<td>2022-4096</td>
<td>Check the work flags of SPLICE.</td>
</tr>
<tr>
<td>2022-29582</td>
<td>Block linked TIMEOUT and LINK_TIMEOUT.</td>
</tr>
<tr>
<td>2022-1508</td>
<td>Check multiple parameters in READ.</td>
</tr>
</tbody>
</table>

CVE Case Study. To better illustrate how RingGuard patches CVEs, we elaborate on the exploitation paths for two of them. CVE-2021-3491 is a buffer overflow vulnerability that could lead to privilege escalation. It is caused by assigning the length of a user-provided I/O buffer without validating its data type and size. This vulnerability allows local attackers to create a heap overflow and execute arbitrary code in the kernel. To prevent such exploits, RingGuard can be configured to check the length of PROVIDE_BUFFERS. We report that RingGuard effectively mitigates this vulnerability.

CVE-2022-29582 is a use-after-free flaw resulting from a race condition in io_uring. By exploiting such vulnerability, an unprivileged attacker can gain root privileges. This exploit relates to two timeout operators in io_uring - TIMEOUT and LINK_TIMEOUT, both of which are used to specify a timeout for io_uring tasks. If these two operators are coupled together (i.e., using LINK_TIMEOUT to specify a timeout for TIMEOUT), a race condition may occur, resulting in a use-after-free vulnerability. RingGuard can be configured to look for such coupled TIMEOUT and LINK_TIMEOUT and remove them in advance to protect against such exploits. Note that such TIMEOUT and LINK_TIMEOUT coupling is rarely (if ever) used by regular io_uring applications. Thus, such mitigation does not affect the original functionality of io_uring.

5 DISCUSSION

RingGuard for Virtual Machines. Apart from conventional scenarios, another use case of RingGuard is to improve the security of io_uring in virtual machines. An io_uring instance can be shared between guest virtual machines (or containers) and the host machine for better I/O performance. However, the existing security mechanism is not flexible enough in restricting these virtual machines from abusing the shared io_uring instances. RingGuard, on the other hand, can be used in this case to improve the security of these shared io_uring instances without compromising their efficiency.

RingGuard for Completion Queue. Although we only introduce RingGuard to audit and log I/O requests in the submission queue, it can also support logging I/O responses in the completion queue with minor engineering effort. This feature can be combined with other eBPF programs to achieve better kernel observability or to improve the efficiency of io_uring request scheduling.

Limitations. While RingGuard enhances the security of io_uring, it still has limitations. Patching vulnerabilities with RingGuard requires prior knowledge of the exploit characteristics, which poses a challenge for undisclosed vulnerabilities. In addition, RingGuard focuses on I/O requests within the io_uring submission queue. If attackers combine other kernel modules to exploit io_uring, RingGuard is unable to defend against it. However, RingGuard eBPF programs can detect a large number of suspicious io_uring operations, improving the security of the io_uring subsystem in general.

6 CONCLUSION

This paper presents RingGuard, an efficient and secure mechanism for regulating I/O requests within the io_uring subsystem using eBPF programs. We show that RingGuard effectively mitigates various I/O-related attacks with moderate performance overhead. Despite some limitations, RingGuard is a valuable tool for strengthening the security of the io_uring subsystem by providing flexible policies for auditing requests. Future research can explore ways to address the disclosed limitations and further extend RingGuard’s capabilities to adapt to evolving threats in the io_uring subsystem. Overall, RingGuard demonstrates the promising potential of eBPF programs to secure I/O operations and defend against sophisticated attacks in modern computing environments.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments. This work is partly supported by the National Natural Science Foundation of China under Grant No. 62002151 and Shenzhen Science and Technology Program under Grant No. SGDX20211030954608029 and No. ZDSYS20210623092007023.

REFERENCES


