

Gap Analysis of Linux io_uring for the The Linux kernel developers

Final Report and Management Summary

2026-11-17

PUBLIC - CC BY-SA 4.0 License

X41 D-Sec GmbH
Soerser Weg 20
D-52070 Aachen
Amtsgericht Aachen: HRB19989

https://x41-dsec.de/
info@x41-dsec.de



Organized by the Open Source Technology Fund

Revision	*Date*	*Change*	*Author(s)*
1	2026-06-05	Final Draft Report	Christian Reitter, Michael Rodler and Robert Femmer
2	2026-06-11	Final Report and Management Summary	Christian Reitter, Michael Rodler and Robert Femmer
3	2026-11-13	Public Release	Michael Rodler

PUBLIC - CC BY-SA 4.0 License

Contents

1 License **4**

2 Executive Summary **5**

3 Introduction **7**
3.1 Methodology 7
3.2 Findings Overview 9
3.3 Scope 9
3.4 Coverage 9
3.5 Recommended Further Tests 10

4 Rating Methodology **11**
4.1 CVSS 11
4.2 Severity Mapping 14
4.3 Common Weakness Enumeration 14

5 Threat Model **15**
5.1 Threat Actors 15
5.2 Trust Boundaries 16
5.3 Attack Vectors 16

6 Results **18**
6.1 Fuzzing Efforts 18
6.2 Findings 23
6.3 Informational Notes 33

7 About OSTIF **40**

8 About X41 D-Sec GmbH **41**

Dashboard

Target	
Customer	The Linux kernel developers
Name	io_uring subsystem
Type	Operating System
Version	v6.19

Engagement	
Type	White-Box Penetration Test
Consultants	3: Christian Reitter, Michael Rodler and Robert Femmer
Engagement Effort	45 person-days, 2026-04-06 to 2026-05-31

Total issues found	5

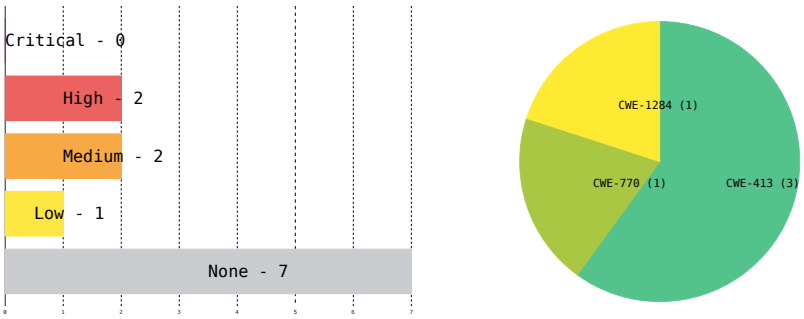


Figure 1: Issue Overview (l: Severity, r: CWE Distribution)

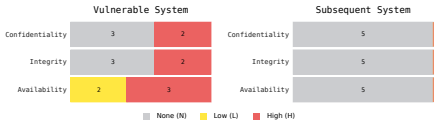


Figure 2: CVSS Impact and Exploitability Distribution

PUBLIC - CC BY-SA 4.0 License

1 License

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License (CC-BY-SA 4.0). This license permits anyone to copy, redistribute, transform, and build upon the material for any purpose, including commercial use, provided that appropriate credit is given to the original author. By applying the ShareAlike condition, derivative works must remain freely available under identical licensing terms, ensuring that the same freedoms to use and adapt are preserved for future recipients. The full legal text of the license is available at `https://creativecommons.org/licenses/by-sa/4.0/`.

Unless stated otherwise, this license does not apply to third-party materials contained in this work. These include but not limited to logos, fonts, images and code snippets, which may be subject to separate copyright or licensing terms.

2 Executive Summary

In April and May 2026, X41 D-Sec GmbH performed a source code audit against the *io_uring* subsystem of the Linux kernel to identify vulnerabilities and weaknesses in the interface.

Counting all Linux kernel issues across all subsystems, a total of five vulnerabilities were discovered during the test by X41. None were rated as having a critical severity, two as high, two as medium, and one as low. Additionally, seven issues without a direct security impact were identified.

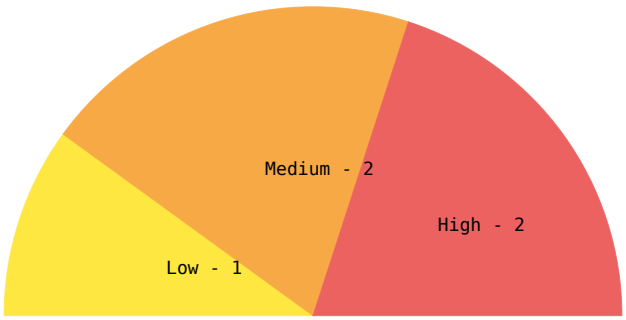


Figure 2.1: Issues and Severity

The *io_uring* subsystem provides an asynchronous high throughput I/O interface for applications running on Linux. Vulnerabilities in the subsystem would allow an attacker to elevate their privileges to root or leak sensitive memory from kernel space.

In a source code audit, all information about the system is made available. The test was performed by three experienced security experts between 2026-04-06 and 2026-05-31.

The most severe issue discovered allows an attacker to elevate their privileges to root by exploiting a Use-After-Free issue in *io_uring*. The root cause of the issue is a lack of serialized access to a shared resource, resulting in a race condition.

The project made contributions to the *Syzkaller* project which improved coverage of fuzzers and let *Syzkaller* reflect changes in latest Linux kernel releases.

Overall, the interface is very difficult to defend due to the inherent high complexity. Further development of this interface makes it likely that exploitable bugs will emerge in the future. Due to the asynchronous nature, bugs can be hard to reason about. Efforts that target *Syzkaller* in order to retain or increase coverage of *io_uring* are crucial to securing the subsystem in the future.

 PUBLIC - CC BY-SA 4.0 License

3 Introduction

X41 reviewed the `io_uring` subsystem of the Linux kernel, which facilitates asynchronous data transfer between kernel and user space while minimizing the amount of system calls. The name stems from the fact that two ring buffers are mapped into the process space of the user space program. One ring is used to submit requests and the other ring is used for completed requests. The interface is designed to provide high throughput data transfer to applications which can not afford to pay the costs of a context switch of a traditional system call. A successful attack against the interface may allow an attacker to elevate their privileges on the system to root. The complexity of the interface has produced a number of exploitable bugs that were abused in the past.

3.1 Methodology

The review was conducted as a source code audit. Additionally, the fuzzing efforts within the Syzkaller project were reviewed.

In a source code audit, all source code is available for static analysis and dynamic testing.

A manual approach for penetration tests and for code reviews is used by X41. This process is supported by tools such as static code analyzers and industry standard web application security tools.[1]

X41 adheres to established standards for source code reviewing and penetration testing. These are in particular the *CERT Secure Coding*[2] standards and the *Study - A Penetration Testing Model*[3] of the German Federal Office for Information Security.

The workflow of source code reviews is shown in figure 3.1. In an initial, informal workshop

[1] `https://portswigger.net/burp`

[2] `https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards`

[3] `https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Penetration/penetration_pdf.pdf?__blob=publicationFile&v=1`

regarding the design and architecture of the application a basic threat model is created. This is used to explore the source code for interesting attack surface and code paths. These are then audited manually and with the help of tools such as static analyzers and fuzzers. The identified issues are documented and can be used in a GAP analysis to highlight changes to previous audits.

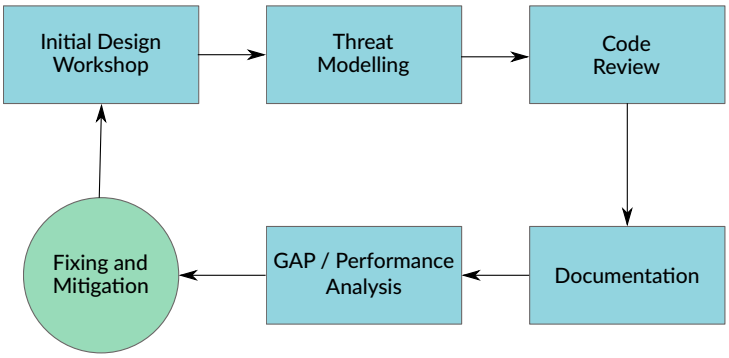


Figure 3.1: Code Review Methodology

 PUBLIC - CC BY-SA 4.0 License

3.2 Findings Overview

DESCRIPTION	SEVERITY	ID	REF
io_uring UAF in io_move_task_work_from_local()	HIGH	IOUR-CR-26-01	6.2.1
io_uring UAF during io_ring_exit_work() steps	HIGH	IOUR-CR-26-02	6.2.2
Soft lockup in io_uring due to NAPI busy polling	MEDIUM	IOUR-CR-26-03	6.2.3
Insufficient parameter validation in ublk driver	LOW	IOUR-CR-26-04	6.2.4
Use-after-free in ublk_cancel_cmd()	MEDIUM	IOUR-CR-26-05	6.2.5
Duplicate Code for Context Retrieval	NONE	IOUR-CR-26-100	6.3.1
Side Effects in Failure Path	NONE	IOUR-CR-26-101	6.3.2
Wrong Constant Used	NONE	IOUR-CR-26-102	6.3.3
Task Warning in ublk Driver	NONE	IOUR-CR-26-103	6.3.4
NULL Pointer Dereference in ublk Queue Commands	NONE	IOUR-CR-26-104	6.3.5
False Positive Lock Assertion in mm Subsystem	NONE	IOUR-CR-26-105	6.3.6
Use-after-free in NFS Exports	NONE	IOUR-CR-26-106	6.3.7

Table 3.1: Security-Relevant Findings

3.3 Scope

The scope of this project included the `io_uring/` sub-folder of the Linux kernel source code at version 6.19.11 and the current efforts of the Syzkaller project to fuzz the attack surface. Syzkaller is not a project that is released with version numbers. At the time of writing this report, X41 based a pull request for improvements on Syzkaller commit 3c0d21318d6dfab26815664c8ad20eaac007b3f3.

Communication with the maintainer of the subsystem was facilitated via email.

3.4 Coverage

A security assessment attempts to find the most important or sometimes as many of the existing problems as possible, though it is practically never possible to rule out the possibility of additional weaknesses being identified in the future.

The time allocated to X41 for this assessment was sufficient to yield a reasonable coverage of the given scope.

A light-weight threat model was created. It is described in chapter 5.

X41 reviewed the source code for the following vulnerability classes

- Kernel object lifetime issues
- Race conditions
- Double fetch
- Logic errors
- Buffer overflows
- Memory leaks
- possible Central Processing Unit (CPU) side-channel attacks

A focus was laid on running and improving *Syzkaller* and triage and analyze crashes produced by *Syzkaller*. The results of this work is described in sec. 6.1. Fuzzing opcodes also exercises kernel subsystems downstream from *io_uring*. Issues found in these systems are also reported as part of the results, even though their root cause may not be the *io_uring* code.

Suggestions for next steps in securing this scope can be found in section 3.5.

3.5 Recommended Further Tests

X41 recommends mitigating the issues described in this report. Afterwards, Common Vulnerabilities and Exposures (CVE) ID should be requested and customers be informed (e.g. via a changelog or a special note for issues with higher severity) to ensure that they can make an informed decision about upgrading or other possible mitigations.

X41 recommends applying the same testing strategy to any newly developed code and major changes.

Code in `io_uring/zcrx.c` is currently not being fuzzed due to the lack of virtualized hardware supporting zero-copy networking through direct memory access. While the feature is guarded by *CAP_NET_ADMIN*, it would be feasible to either implement a mock driver or a driver based on virtio so that this interface can be fuzzed in the future.

Another avenue of improvement is rooted in the disconnect of the *Syzkaller* code base and the kernel interface. As the feature set of the kernel interface moves, the *Syzkaller* definitions require active maintenance. Automated tooling could be created to streamline the maintenance process which currently is laborious and error-prone.

4 Rating Methodology

Security vulnerabilities are given a purely technical rating by the testers when they are discovered during a test. Business factors and financial risks for The Linux kernel developers are beyond the scope of a penetration test, which focuses entirely on technical factors. However, technical results from a penetration test may be an integral part of a general risk assessment. A penetration test is based on a limited time frame and only covers vulnerabilities and security issues which have been found in the given time, there is no claim for full coverage.

The Common Vulnerability Scoring System (CVSS) is used to score all findings relevant to security. The resulting CVSS score is mapped to qualitative ratings as shown below.

4.1 CVSS

Testers rate all security-relevant findings using the CVSS industry standard version 4.

Vulnerabilities scored with CVSS get a numeric value based on several metrics ranging from 0.0 (least worst) to 10.0 (worst).

The score captures different factors that express the impact and the ease of exploitation of a vulnerability among other factors. For a detailed description of how the scores are calculated, please see the CVSS version 4.0 specification.[1]

The metrics used to calculate the final score are grouped into three different categories.

[1] `https://www.first.org/cvss/v4.0/specification-document`

The *Base Metric Group* represents the intrinsic and fundamental characteristics of a vulnerability that are constant over time and user environments. It captures the following metrics:

- Attack Vector (AV)
- Attack Complexity (AC)
- Attack Requirements (AT)
- Privileges Required (PR)
- User Interaction (UI)
- Vulnerable/Subsequent System Confidentiality (VC/SC)
- Vulnerable/Subsequent System Integrity (VI/SI)
- Vulnerable/Subsequent System Availability (VA/SA)

The *Threat Metric Group* represents the current state of exploit techniques and the availability of proof of concepts. It captures the following metric:

- Exploit Maturity (E)

The *Environmental Metric Group* represents the characteristics of a vulnerability that are relevant and unique to a particular user's environment. It includes the following metrics:

- Confidentiality Requirement (CR)
- Integrity Requirement (IR)
- Availability Requirement (AR)
- Modified Attack Vector (MAV)
- Modified Attack Complexity (MAC)
- Modified Attack Requirements (MAC)
- Modified Privileges Required (MPR)
- Modified User Interaction (MUI)
- Modified Vulnerable System Confidentiality (MVC)
- Modified Vulnerable System Integrity (MVI)
- Modified Vulnerable System Availability (MVA)
- Modified Subsequent System Confidentiality (MSC)
- Modified Subsequent System Integrity (MSI)
- Modified Subsequent System Availability (MSA)

A CVSS vector defines a specific set of metrics and their values, and it can be used to reproduce and assess a given score. It is rendered as a string that exactly reproduces a score.

PUBLIC - CC BY-SA 4.0 License

For example, the vector `CVSS:4.0/AV:N/AC:H/AT:N/PR:L/UI:A/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N` defines a base score metric with the following parameters:

- Attack Vector: Network
- Attack Complexity: High
- Attack Requirements: None
- Privileges Required: Low
- User Interaction: Active
- Vulnerable System Confidentiality: High
- Vulnerable System Integrity: Low
- Vulnerable System Availability: None
- Subsequent System Confidentiality: None
- Subsequent System Integrity: None
- Subsequent System Availability: None

In this example, a network-based attacker performs a complex attack after gaining access to some privileges, by tricking a user into performing some actions. This allows the attacker to read confidential data and change some parts of that data.

The detailed scores are the following:

Metric	Score
Exploitability	Medium
Complexity	Medium
Vulnerable system	Medium
Subsequent system	Low
Exploitation	High
CVSS Score	5.8 (Medium)

CVSS vectors can be automatically parsed to recreate the score, for example, with the CVSS calculator provided by FIRST, the organization behind CVSS: `https://www.first.org/cvss/calculator/4.0#CVSS:4.0/AV:N/AC:H/AT:N/PR:L/UI:A/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N`.

PUBLIC - CC BY-SA 4.0 License

4.2 Severity Mapping

To help in understanding the results of a test, numeric CVSS scores are mapped to qualitative ratings as follows:

Severity Rating	CVSS Score
NONE	0.0
LOW	0.1–3.9
MEDIUM	4.0–6.9
HIGH	7.0–8.9
CRITICAL	9.0–10.0

4.3 Common Weakness Enumeration

The Common Weakness Enumeration (CWE) is a set of software weaknesses that allows vulnerabilities and weaknesses in software to be categorized. If applicable, X41 gives a CWE ID for each vulnerability that is discovered during a test.

CWE is a very powerful method for categorizing a vulnerability. It gives general descriptions and solution advice on recurring vulnerability types. CWE is developed by *MITRE*.[2] More information can be found on the CWE site at `https://cwe.mitre.org/`.

[2] `https://www.mitre.org`

5 Threat Model

The following light threat model was created during this security audit and should serve as a benchmark for rating the findings. It can be extended over time as new attack vectors emerge.

5.1 Threat Actors

The following lists potential threat actors and the capabilities they inherently possess.

5.1.1 Local Attacker

The local system attacker may use the Linux kernel *io_uring* API to create instances and use *io_uring* and manipulate their behavior and properties. They also have access to other Linux kernel API, whose functionality is adjacent or may have side-effects on the behavior of the *io_uring* system. This attacker is the most capable attacker of the described attackers, since they have unfettered access to the attack surface. A local attacker with access to the hardware is explicitly excluded here and considered out of scope.

5.1.2 CPU Adjacent Attacker

The attacker has access to a system which runs on the same CPU as the target system utilizing *io_uring*. They may attempt to exfiltrate information via CPU side-channel attacks (Spectre, Meltdown, etc.) An attacker with access to the hardware is explicitly excluded here and considered out of scope.

5.1.3 Remote Attacker

The attacker controls the remotely connected resource which an *io_uring* is communicating with. Usually this is a network resource, e.g. a server that was connected to by an instance of *io_uring* via ***connect()***. The remote attacker can control the data and the timing of some of the data transactions.

5.2 Trust Boundaries

The most prominent trust boundary is the shared memory used for rings between userspace and kernel. The kernel directly reads from attacker controlled memory. The kernel may offload processing of requests to a dedicated kernel task, which may run with a different set of credentials than the requesting task. *io_uring* must ensure that the request is only executed with the credentials of the original requesting task.

io_uring defines three system calls which make up the second trust boundary:

- ***io_uring_setup()***
- ***io_uring_enter()***
- ***io_uring_register()***

mmap() and ***close()*** can be called on the file descriptor and are therefore part of this trust boundary as well as the code exposing information under the `/proc` directory.

The most relevant trust boundaries in the context of *io_uring* inherent to the Linux kernel include task isolation and credentials, namespaces, SELinux, and capabilities. In the context of *io_uring*, seccomp filters for I/O operations are out of scope, because they do not apply to I/O requests issued through *io_uring*. A process relying on seccomp filters must also deny access to the three system calls provided by *io_uring*.

5.3 Attack Vectors

The following lists attack vectors, which can potentially be used to cause break down of trust boundaries.

5.3.1 Memory Corruption

This attack vector encompasses any bug that corrupts kernel memory in a way that userspace may seize control of the kernel. The outcomes include kernel object lifetime issues, buffer overflows or type confusions stemming from a myriad of bug classes like use-after-free, double-free, double-fetch, race conditions, integer overflows, etc.

5.3.2 Logic Errors Involving Credentials

An attacker could try to use a logic bug to execute a request with a set of credentials they would usually not have access to. The operations executed under the different set of credentials could be used to gain root privileges on the system. The attack vector includes breaking trust boundaries inherent to the Linux kernel.

5.3.3 Leak of Kernel Memory

Pointers in kernel memory can be used to bypass Address Space Layout Randomization (ASLR) and may aid in crafting reliable exploits. These can be attributed to unitialized memory or kernel object lifetime issues and out of bounds reads. Further, CPU side-channel attacks may be mounted against the *io_uring* code to leak kernel secrets.

5.3.4 Denial of Service

Attackers could try to overwhelm the processing queue of *io_uring* with operations that take longer than or allocate more memory than anticipated, which may result in denial of service of the system as a whole. Another vector may be tricking *io_uring* into a dead lock due to conflicting mutex lock acquisitions.

6 Results

This chapter describes the results of this audit. The security-relevant findings are documented in Section 6.2. Additionally, findings without a direct security impact are documented in Section 6.3.

6.1 Fuzzing Efforts

This chapter documents the current state of fuzzing of *io_uring* within *Syzkaller* in section 6.1.1. The efforts taken during this project are described in section 6.1.2.

6.1.1 The Current State of Fuzzing *io_uring*

This section briefly introduces the systems under investigation. A full description with all details is omitted here for brevity; more detailed information can be found in the respective project's documentation.

io_uring works by providing userspace and kernel with two ring buffers in shared memory, which facilitate passing of commands that (most of the time) can be mapped to kernel interfaces that are exposed through regular system calls (colloquially dubbed syscalls). The relevant system calls are typically handling I/O for storage devices or network interfaces. The aim of this design is to achieve high performance throughput by avoiding expensive syscalls as much as possible.

Syzkaller is a coverage-guided fuzzer which targets the Operating System (OS) kernel interface as exposed to userspace via system calls. To generate useful sequences of system calls, the fuzzer relies on a Domain-specific Language (DSL) called *syzlang* to specify the details of how system calls are performed. The specification is used to generate random, valid programs that issue various syscalls in a sequence. These programs are executed in a virtual machine with a kernel compiled for generating code coverage for syscalls using the *KCOV* mechanism. The coverage information gathered during executing of a generated program is used by the fuzzer to build a corpus of interesting test cases.

Syzbot is a project operated and funded by Google. Essentially it is a cluster of *Syzkaller* instances that use different configurations to achieve maximum code coverage. The *Syzbot* cluster runs fuzzer instances on multiple architectures, kernel versions, kernel configurations, with varying sanitizer and fuzzer settings. The results of the *Syzkaller* runs are publicly accessible at syzbot.org.

The design of *io_uring* avoids system calls and instead replaces them by asynchronously executed requests that are passed from userspace to the kernel using shared memory. At first glance, this sounds problematic for a system call oriented fuzzer. However, the DSL of *Syzkaller* is designed to allow defining pseudo-syscalls. These pseudo-syscalls are fuzzed like regular syscalls, but run custom logic, which can be used to set up resources before using them. This makes it more likely to generate useful syscalls by taking prior knowledge about the syscall interface into account. Furthermore, *Syzkaller* allows implementing pseudo-syscalls in C code in the syscall executor harness to perform multiple related syscalls in one step. For fuzzing *io_uring*, *Syzkaller* defines multiple pseudo-syscalls. Setting up a valid *io_uring* requires multiple system calls and mapping memory with the right addresses and flags. To achieve better fuzzing efficiency *Syzkaller* uses a single pseudo-syscall to create a valid *io_uring*. Additionally, to fuzz *io_uring* operations, *Syzkaller* defines a pseudo-syscall that puts requests into the submission queue of a previously set up ring buffer and then signals the kernel to work on these entries.

The code coverage of current efforts can be found on the *Syzbot* project page.[1]

Creating and maintaining *syzlang* definitions is a manual and error-prone task. Discrepancies between the definitions and the actual kernel interface are not detected automatically. Additions to the kernel interface are usually not automatically added for newer kernel versions. This is especially problematic for rapidly developing interfaces like *io_uring*. At the beginning of testing, support for new *io_uring* APIs was about 14 months behind stable kernel releases.

6.1.2 Improvements

During analysis of the current state of system call fuzzing, X41 noticed that *Syzkaller* fuzzing of *io_uring* is held back by the existing fuzzer grammar and coverage collection logic.

First, *Syzkaller* definitions for the *io_uring* interface were incomplete for current stable kernels. X41 improved the definitions by adding missing opcodes and specifying data structures more precisely.[2] This makes it more likely for *Syzkaller* to generate valid *io_uring* sycall sequences and reach more complex bugs. Second, *Syzkaller* gathers code coverage using the KCOV mechanism, which is not well suited for *io_uring*. X41 improved this by enabling KCOV

[1] https://syzbot.org/upstream/coverage/subsystems?period=month
[2] https://github.com/google/syzkaller/pull/7305

remote coverage, which allows the fuzzer to track code coverage of *io_uring* in background threads.

6.1.2.0.1 Improving Syzlang Definitions At the beginning of testing, the *syzlang* specification for *io_uring* supported 41 operations, whereas the target Linux kernel version defines 64. X41 added new *syzlang* definitions for the following opcodes:

- IORING_OP_BIND
- IORING_OP_EPOLL_WAIT
- IORING_OP_FIXED_FD_INSTALL
- IORING_OP_FSETXATTR
- IORING_OP_FTRUNCATE
- IORING_OP_FUTEX_WAIT
- IORING_OP_FUTEX_WAITV
- IORING_OP_FUTEX_WAKE
- IORING_OP_GETXATTR
- IORING_OP_LISTEN
- IORING_OP_NOP128
- IORING_OP_PIPE
- IORING_OP_READ_MULTISHOT
- IORING_OP_READV_FIXED
- IORING_OP_RECV_ZC
- IORING_OP_SENDMSG_ZC
- IORING_OP_SEND_ZC
- IORING_OP_SETXATTR
- IORING_OP_SOCKET
- IORING_OP_URING_CMD
- IORING_OP_URING_CMD128
- IORING_OP_WAITID
- IORING_OP_WRITEV_FIXED

This makes it easier for *Syzkaller* to reach the corresponding *io_uring* functionality with suitable parameter values. Similarly, X41 added several new *Syzkaller* targets for ***io_uring_register()*** operations.

During testing, the *Syzkaller* developers merged a pull request[3] from other authors which introduces *syzlang* grammar definitions for the *ublk* userspace block driver. This driver heavily relies on the *io_uring* API but is a separate component. X41 adopted the relevant *Syzkaller* changes

[3] https://github.com/google/syzkaller/pull/7039

into testing. While X41 primarily aimed at finding problems in *io_uring* logic through the code exercised by *ublk* workflows, the early and *io_uring*-focused testing of these definitions contributed to finding several bugs in the *ublk* driver as well.

In newer kernel versions, the *io_uring* interface optionally supports special zero-copy network operations via Direct Memory Access (DMA) to network interface cards if the *CONFIG_IO_URING_ZCRX* kernel configuration flag is enabled. The main purpose of this class of opcodes (indicated by a `_ZC` suffix) is higher networking performance through reduced memory operation overhead. End-to-end flows for this zero-copy functionality require `CAP_NET_ADMIN` privileges to set up, as well as the presence of network interfaces and drivers that support DMA. X41 expects that the required privileges and device availability make it more difficult for attackers to leverage any potential security flaws in this component. The hardware with these specifications is not common, and at the time of testing, X41 did not find stable virtio drivers that could emulate such capabilities in a *Syzkaller* VM environment. X41 still introduced *syzlang* grammar definitions for the corresponding opcodes and enabled *CONFIG_IO_URING_ZCRX* during testing, but test coverage was limited to the zero-copy code sections which are reachable without a suitable network card. Deeper testing of this functionality represents a future work item.

The *syzlang* grammar has a number of mechanisms to shape fuzzer syscall behavior and provide hints, ideally without over-constraining the syscall descriptions. X41 reviewed the existing *io_uring* definitions in *sys/linux/io_uring.txt* and compared them against the kernel source to spot definition problems which could make most of the generated syscall variations ineffective. For example, X41 found syscall parameters used in the wrong order, use of the wrong integer sizes for parameters, dynamic flags or pointers set to constant 0, insufficient handling of reserved fields, incorrect length calculation, and other details that make the generated syscalls unlikely to pass initial validation steps on the kernel side. Given the experimental nature of the definitions and the manual translation from kernel code to syzlang code, X41 considers that some issues are unavoidable and require incremental efforts to improve. However, an improvement in automated or semi-automated checks to effectively spot these discrepancies or validate new PR changes like X41's for regressions could make fuzzer maintenance and improvements efforts more efficient in the future.

The changes to *sys/linux/io_uring.txt* are available as a pull request.[4]

6.1.2.0.2 Improving Syzkaller During testing, X41 observed that *Syzkaller* penalizes or rejects a given type of syscall if it appears too frequently in the corpus, based on hardcoded thresholds. Due to the special design of *io_uring*, which gates many different pseudo-syscalls behind a small number of actual syscalls, the same syscalls are frequently needed to set up and submit operations. X41 considers that the current penalty logic can be detrimental to *io_uring* fuzzing

[4] `https://github.com/google/syzkaller/pull/7305`

on some fuzzer configurations, especially if the automatic description mode is enabled[5]. X41 performed testing with a modified *Syzkaller* instance with manually relaxed threshold values to avoid this issue.[6]

Additionally, X41 reported several other minor issues or limitations in *Syzkaller* itself related to runtime statistics, coverage reports and mutation probabilities.[7,8,9]

6.1.2.0.3 Enabling Remote Coverage *io_uring* can not complete all requests immediately. Some requests may take longer, for instance requests that rely on waiting for slow hardware, like hard disks. These requests are driven by the *io_uring* workqueue, which runs as a kernel task in the background. Some types of requests like *IORING_OP_FSYNC* are always pushed to the work queue. This is a problem for collecting coverage information by *Syzkaller*, because regular coverage is only collected for the task executing the system calls and not for background tasks handling work.

The same applies for the instances of *io_uring* that use *IORING_SETUP_SQPOLL*. In that case, a kernel tasks is polling for new submission queue entries and dispatches the requests without relying on a call to ***iouring_enter()***. However, coverage information for this task is also not available as feedback to *Syzkaller*.

The system that produces coverage information in the Linux kernel provides a way to annotate tasks in code that are to be included when generating coverage information. A call to ***kcov_remote_start()*** enables gathering coverage and ***kcov_remote_stop()*** stops gathering coverage. A handle per subsystem is used in order for userspace to be able to specify which coverage information to gather. This handle has to be specified and therefore be known in advance by userspace to collect the desired coverage information. Currently, the code for *io_uring* has no annotations for collecting remote coverage.

This patch has been merged into the Linux kernel.[10,11] The patch uses the tasks common handle and therefore does not require any modifications to *Syzkaller* to work.

[5] Such as via the configuration setting *descriptions_mode=any*.
[6] https://github.com/google/syzkaller/issues/7204
[7] https://github.com/google/syzkaller/issues/7206
[8] https://github.com/google/syzkaller/issues/7207
[9] https://github.com/google/syzkaller/issues/7209
[10] https://lore.kernel.org/io-uring/178286250200.176362.12773693148044387806.b4-ty@b4/
[11] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c905736a46892e4776efc7f50888d67715d6ec08

6.2 Findings

The following subsections describe findings with a direct security impact that were discovered during the test.

6.2.1 IOUR-CR-26-01: io_uring UAF in io_move_task_work_from_local()

Severity:	HIGH / 8.5
CVSS Vector:	CVSS:4.0/AV:L/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N
CWE:	413 – Improper Resource Locking
Component:	`io_uring/tw.c`:io_move_task_work_from_local()

6.2.1.1 Description

Once a request sent through *io_uring* completes, the completion queue entry in the user-space mapped completion queue ring buffer has to be filled in. This work is driven by adding it to so called task work, which runs before a system call returns to user space control, e.g. when the application calls ***io_uring_enter()***. To limit the amount of work per system call, a patch[12] introduced a retry list, which is updated with work items that would exceed a defined threshold and to be completed during the next system call.

When accessing the retry list, the ring context lock must be held. This is not the case when moving (and completing) task work to the local task while shutting down the ring context and therefore it can race with regular task work in another task. This can have a number of consequences. The most commonly observed is that a request is completed twice.

Requests are cached to avoid allocating and releasing their required memory each time a request is made. Upon completion, the allocation is returned to the cache and can be re-used by the next request. When a request is completed twice, the code operates on stale or reclaimed memory, which causes all the issues associated with a use-after-free condition.

Although crafting an exploit may be more involved than a classic use-after-free, X41 assumes that this bug is exploitable. This bug was found during fuzzing with *Syzkaller*. To trigger this bug, a high number of requests has to be issued that complete during the exit procedure of the ring context. The issue is present in Linux kernel versions 6.13 - 7.0.

[12] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=f46b9cdb22f7a167c36b6bcddaef7e8aee2598fa`

6.2.1.2 Remediation

A patch[13] was submitted during the audit by Jens Axboe and was backported to all vulnerable supported versions.

[13] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=17666e2d7592c3e85260cafd3950121524acc2c5

6.2.2 IOUR-CR-26-02: io_uring UAF during io_ring_exit_work() steps

Severity:	HIGH / 8.5
CVSS Vector:	CVSS:4.0/AV:L/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N
CWE:	413 – Improper Resource Locking
Component:	`io_uring/cancel.c`:io_uring_try_cancel_requests()

6.2.2.1 Description

Fuzzing with *Syzkaller* revealed a series of *io_uring* related kernel crashes and warnings which are consistent with a race condition and use-after-free access patterns. X41 regards this UAF as a separate vulnerability, since these issues were detected with patches for `io_uring/tw.c` ***io_move_task_work_from_local()*** already applied to the kernel under test.

Unfortunately, *Syzkaller* was unable to reliably reproduce any of the observed issue symptoms and build a proof-of-concept. This made it more difficult to pinpoint the root cause and characteristics of this issue.

Listings 6.1 and 6.2 show representative crash symptoms observed during fuzzing.

```
1 Oops: general protection fault, probably for non-canonical address 0xdead000000001159: 0000 [
  ↪ #1] SMP
2 [...]
3 Workqueue: iou_exit io_ring_exit_work
4 RIP: 0010:io_linked_nr+0x1a/0x40 io_uring/io_uring.c:518
```

Listing 6.1: General Protection Fault on Poisoned Pointer

```
1 refcount_t: underflow; use-after-free.
2 WARNING: lib/refcount.c:28 at refcount_warn_saturate+0x91/0xd0 lib/refcount.c:28, CPU#0: syz.3.13
  ↪ 109/86740
3 [...]
4 RIP: 0010:refcount_warn_saturate+0x91/0xd0 lib/refcount.c:28
```

Listing 6.2: Refcount Underflow and Use-after-free Warning

Based on the error logs provided by X41, Jens Axboe identified several code positions related to worker task link reference pointers and timeout handling which were insufficiently protected by existing mutexes. One example of such a location is listed in listing 6.3:

```
diff --git a/io_uring/cancel.c b/io_uring/cancel.c
index 5e5eb9cfc7cd..4aa3103ba9c3 100644
--- a/io_uring/cancel.c
+++ b/io_uring/cancel.c
@@ -561,8 +561,8 @@ __cold bool io_uring_try_cancel_requests(struct io_ring_ctx *ctx,
     ret |= io_waitid_remove_all(ctx, tctx, cancel_all);
     ret |= io_futex_remove_all(ctx, tctx, cancel_all);
     ret |= io_uring_try_cancel_uring_cmd(ctx, tctx, cancel_all);
-    mutex_unlock(&ctx->uring_lock);
     ret |= io_kill_timeouts(ctx, tctx, cancel_all);
+    mutex_unlock(&ctx->uring_lock);
     if (tctx)
         ret |= io_run_task_work() > 0;
     else
```

Listing 6.3: Patch 3/3 for Race Condition Behavior

The first patchset revision contained a regression, which X41 found during testing before inclusion into a stable release.

With the second revision of the patchset applied to the kernel under test, no use-after-free related symptoms were observed again.

Local experimentation suggests that triggering this issue likely involves an *IORING_OP_LINK_TIMEOUT* submission queue entry referencing another submission queue entry, for example a network operation like *IORING_OP_CONNECT*. However, X41 currently does not have a working proof of concept.

X41 classified this as a security vulnerability under the assumption that a low-privileged local user can deliberately trigger the race condition, which seems plausible given the *io_uring* API permissions. The scoring also expects that this vulnerability can be leveraged for impacts beyond a local Denial of Service (DoS).

6.2.2.2 Remediation

During the audit Jens Axboe developed patches.[14,15]

[14] https://lore.kernel.org/io-uring/20260511182217.226763-1-axboe@kernel.dk/T/
[15] https://lore.kernel.org/io-uring/6171c1b4-fa7c-478f-ae61-ea0c0e939e1d@kernel.dk/T/

6.2.3 IOUR-CR-26-03: Soft lockup in io_uring due to NAPI busy polling

Severity:	MEDIUM / 4.8
CVSS Vector:	CVSS:4.0/AV:L/AC:L/AT:N/PR:L/UI:N/VC:N/VI:N/VA:L/SC:N/SI:N/SA:N
CWE:	770 – Allocation of Resources Without Limits or Throttling
Component:	`io_uring/napi.c:io_napi_register_napi()`

6.2.3.1 Description

A low-privileged local user can deliberately trigger a "soft lockup" kernel bug condition, during which the kernel loops for more than 20s without giving other tasks a chance to run. A combination of ***io_uring_setup()***, ***io_uring_register()*** with *IORING_REGISTER_NAPI* and ***io_uring_enter()*** syscalls can reliably cause this issue resulting in the kernel log message shown in listing 6.4. The issue exists in the kernel versions 6.9 - 7.0.

```
watchdog: BUG: soft lockup - CPU#0 stuck for 26s!
```

Listing 6.4: Excerpt of Linux Diagnostics

6.2.3.2 Remediation

Jens Axboe identified an unbounded variable in the busy polling mechanism of the io_uring NAPI registration interface as responsible for the lockup. The patch in listing 6.5 limits ***napi->busy_poll_to*** to 10ms.[16]

```
--- a/io_uring/napi.c
+++ b/io_uring/napi.c
@@ -276,6 +276,8 @@ static int io_napi_register_napi(struct io_ring_ctx *ctx,
/* clean the napi list for new settings */
io_napi_free(ctx);
WRITE_ONCE(ctx->napi_track_mode, napi->op_param);
+    /* cap NAPI at 10 msec of spin time */
+    napi->busy_poll_to = min(10000, napi->busy_poll_to);
WRITE_ONCE(ctx->napi_busy_poll_dt, napi->busy_poll_to * NSEC_PER_USEC);
WRITE_ONCE(ctx->napi_prefer_busy_poll, !!napi->prefer_busy_poll);
return 0;
```

[16] https://lore.kernel.org/io-uring/ae34680f-4890-40a5-9d20-d99e719e9008@kernel.dk/T/

Listing 6.5: NAPI Patch for Soft Lockup

Due to the relatively low impact of this vulnerability, a backport of the patch is not planned.

 PUBLIC - CC BY-SA 4.0 License

6.2.4 IOUR-CR-26-04: Insufficient parameter validation in ublk driver

Severity:	LOW / 1.8
CVSS Vector:	CVSS:4.0/AV:L/AC:H/AT:P/PR:H/UI:N/VC:N/VI:N/VA:L/SC:N/SI:N/SA:N
CWE:	1284 – Improper Validation of Specified Quantity in Input
Component:	`drivers/block/ublk_drv.c`

6.2.4.1 Description

X41 identified multiple issues in the parameter validation of the ublk block driver. They can be triggered by a user with sufficient privileges to create udev rules.

Firstly, ***ublk_validate_params()*** missed limits on upper and lower bounds of several parameters.

During testing with *Syzkaller*, X41 observed undefined behavior in the Linux kernel using the Undefined Behavior Sanitizer (*UBSAN*) due to the incomplete parameter checks.

For larger values of *physical_bs_shift*, the left shift operation in listing 6.6 generates results that could not be expressed in the available size of the target variable. The resulting warning is shown in listing 6.7.

```
struct queue_limits lim = {
        .logical_block_size     = 1 << p->logical_bs_shift,
        .physical_block_size    = 1 << p->physical_bs_shift,
        .io_min                 = 1 << p->io_min_shift,
        .io_opt                 = 1 << p->io_opt_shift,
        .max_hw_sectors         = p->max_sectors,
        .chunk_sectors          = p->chunk_sectors,
        .virt_boundary_mask     = p->virt_boundary_mask,
        .max_segments           = USHRT_MAX,
        .max_segment_size       = UINT_MAX,
        .dma_alignment          = 3,
};
```

Listing 6.6: Possible Out of Bounds Shift

```
UBSAN: shift-out-of-bounds in drivers/block/ublk_drv.c:2912:28
shift exponent 44 is too large for 32-bit type 'int'
```

Listing 6.7: UBSAN shift-out-of-bounds Warning

In the available patch, the undefined behavior is prevented via limits on *physical_bs_shift*.

Similarly, X41 triggered a kernel warning via an undersized *max_sectors* parameter, which should be prevented with mandatory minimum limits, as shown in listing 6.8.

```
1 lim->max_hw_sectors < PAGE_SECTORS
2 WARNING: block/blk-settings.c:402 at blk_validate_limits+0x645/0xac0 block/blk-settings.c:402, CP
  ↪ U#1: iou-wrk-16977/16978
```

Listing 6.8: Warning on hw_sectors Assert

Secondly, ***ublk_validate_params()*** had insufficient error handling logic on parameter validation failure, effectively persisting some of the parameter changes instead of rejecting them if errors occur in ***copy_from_user()*** or ***ublk_validate_params()***. Under some circumstances, this made the existing validation mechanisms ineffective and allowed users to create ublk block devices with illegal parameter values.

6.2.4.2 Remediation

Ming Lei created patches for both issues.[17,18]

[17] https://lore.kernel.org/linux-block/20260506082238.22363-1-tom.leiming@gmail.com/
[18] https://lore.kernel.org/linux-block/20260510144843.769031-1-tom.leiming@gmail.com/

6.2.5 IOUR-CR-26-05: Use-after-free in ublk_cancel_cmd()

Severity:	MEDIUM / 5.6
CVSS Vector:	CVSS:4.0/AV:L/AC:H/AT:P/PR:H/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:N
CWE:	413 – Improper Resource Locking
Component:	`drivers/block/ublk_drv.c`

6.2.5.1 Description

X41 discovered a use-after-free issue in the ublk block driver using *Syzkaller*.

As part of a race condition between ***ublk_reset_ch_dev()*** and ***ublk_cancel_cmd()***, the ***ublk_cancel_cmd()*** function can operate on a stale pointer. This causes use-after-free accesses in ***__io_uring_cmd_done()*** and other related impact, such as kernel crashes due to illegal pointer accesses, and warnings on violated checks.

In its default permission configuration, the ublk control APIs are only accessible to users with administrative privileges, who have the privileges to deploy a udev rule. However, the sections `Unprivileged mode` and `Use unprivileged ublk in Docker` in the ublksrv documentation[19] suggest that there are popular setup configurations in which ublk control is granted to lower-privileged local users or users in container environments. Under these conditions, a denial-of-service (DoS) through kernel exceptions or other potential impact of the UAF crosses trust boundaries and therefore represents a security issue.

The observed crash behavior varied, as shown in listing 6.9 through listing 6.13.

```
1 BUG: kernel NULL pointer dereference, address: 0000000000000000
2 #PF: supervisor read access in kernel mode
3 [...]
4 RIP: 0010:__io_uring_cmd_done+0xfa/0x470 io_uring/uring_cmd.c:161
```

Listing 6.9: Kernel NULL Pointer Dereference

```
1 BUG: unable to handle page fault for address: 0000000004655225 ,
2 #PF: supervisor read access in kernel mode
3 [...]
4 RIP: 0010:__io_uring_cmd_done+0xfa/0x470 io_uring/uring_cmd.c:161
```

[19] `https://github.com/ublk-org/ublksrv`

Listing 6.10: Page Fault during Read Access

```
Oops: general protection fault, probably for non-canonical address 0xdead000000001091: 0000 [#1] SMP
CPU: 1 UID: 0 PID: 56040 Comm: iou-wrk-56028 Not tainted 6.19.11-dirty #2 PREEMPT(voluntary)
[...]
RIP: 0010:__io_uring_cmd_done+0xfa/0x470 io_uring/uring_cmd.c:161
```

Listing 6.11: General Protection Fault on Poisoned Pointer

```
DEBUG_LOCKS_WARN_ON(lock->magic != lock)
WARNING: kernel/locking/mutex.c:593 at __mutex_lock_common kernel/locking/mutex.c:593 [inline], CPU#0: iou-wrk-22567/22573
WARNING: kernel/locking/mutex.c:593 at __mutex_lock+0x5aa/0x1070 kernel/locking/mutex.c:776, CPU#0: iou-wrk-22567/22573
[...]
RIP: 0010:__mutex_lock_common kernel/locking/mutex.c:593 [inline]
RIP: 0010:__mutex_lock+0x5b1/0x1070 kernel/locking/mutex.c:776
```

Listing 6.12: Debug Assert in Mutex Handling

```
req->flags & REQ_F_APOLL_MULTISHOT
WARNING: io_uring/uring_cmd.c:146 at __io_uring_cmd_done+0x410/0x470 io_uring/uring_cmd.c:146, CPU#0: iou-wrk-104082/104096
Modules linked in:
CPU: 0 UID: 0 PID: 104096 Comm: iou-wrk-104082 Not tainted 6.19.11-dirty #2 PREEMPT(voluntary)
[...]
RIP: 0010:__io_uring_cmd_done+0x410/0x470 io_uring/uring_cmd.c:146
```

Listing 6.13: Assert Warning in io_uring Command Handling

6.2.5.2 Remediation

Ming Lei identified the underlying issue and created a patch.[20]

[20] https://lore.kernel.org/linux-block/20260508123746.242018-1-tom.leiming@gmail.com/T/

6.3 Informational Notes

The following observations do not have a direct security impact, but are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

6.3.1 IOUR-CR-26-100: Duplicate Code for Context Retrieval

Component:	`io_uring/register.c`, `io_uring/io_uring.c`

6.3.1.1 Description

The syscalls ***io_uring_register()*** and ***io_uring_enter()*** both need to retrieve the ring context from the file structure associated with the passed file descriptor. Both implement their own logic to accomplish this. ***io_uring_enter()*** inlines the code directly in the syscall entry function and ***io_uring_register()*** utilizes ***io_uring_register_get_file()***. The latter also takes an additional reference to files that were registered through *`IORING_ENTER_REGISTERED_RING`*. This reference is superfluous.

6.3.1.2 Remediation

X41 recommends deduplicating the code and implementing a single function to retrieve the ring context from a file descriptor to improve readability and maintainability.

Patches addressing this issue were authored during the audit.[21]

[21] `https://lore.kernel.org/io-uring/20260408193023.397746-1-axboe@kernel.dk/`

6.3.2 IOUR-CR-26-101: Side Effects in Failure Path

Component: `io_uring/tctx.c`

6.3.2.1 Description

The function ***__io_uring_add_tctx_node()*** allocates a task specific context and attaches it to the current task by calling ***io_uring_alloc_task_context()***. This action is not reverted should adding the task-specific context to the ring context fail due to an allocation error later in the function as can be seen in listing 6.14.

```
1  int __io_uring_add_tctx_node(struct io_ring_ctx *ctx)
2  {
3          struct io_uring_task *tctx = current->io_uring;
4          struct io_tctx_node *node;
5          int ret;
6          if (unlikely(!tctx)) {
7                  ret = io_uring_alloc_task_context(current, ctx);
8                  if (unlikely(ret))
9                          return ret;
10                 tctx = current->io_uring; // Side effect
11                 // [..] omitted for brevity
12         }
13         if (!xa_load(&tctx->xa, (unsigned long)ctx)) {
14                 node = kmalloc(sizeof(*node), GFP_KERNEL);
15                 if (!node)
16                         return -ENOMEM; // Failure path
17                 // [..] omitted for brevity
18         }
19         return 0;
20 }
```

Listing 6.14: Side Effects during Resource Allocation

6.3.2.2 Remediation

X41 recommends cleaning up the logic so that the task-specific context is either set up properly or the corresponding pointer in the task is `NULL`.

Patches addressing this issue were authored by Jens Axboe during the audit.[22]

[22] `https://lore.kernel.org/io-uring/20260408192711.396827-2-axboe@kernel.dk/`

6.3.3 IOUR-CR-26-102: Wrong Constant Used

Component: `io_uring/zcrx.c`

6.3.3.1 Description

The function ***io_allocate_rbuf_ring()*** computes the mmap offset by shifting the variable *id* by the constant *IORING_OFF_PBUF_SHIFT* as can be seen in listing 6.15.

```
1     mmap_offset += id << IORING_OFF_PBUF_SHIFT;
```

Listing 6.15: Usage in ***io_allocate_rbuf_ring()***

The *id* variable should be shifted by *IORING_OFF_ZCRX_SHIFT* instead. There is no practical impact since both constants are currently defined with an identical value of *16*.

6.3.3.2 Remediation

The kernel developers replaced the use of the wrong constant with the correct constant.[23]

[23] `https://lore.kernel.org/io-uring/31f58c90bd1222b420f24a71f2b9eea4cccefe45.1774444007.git.asml.silence@gmail.com/T/`

6.3.4 IOUR-CR-26-103: Task Warning in ublk Driver

Component: `drivers/block/ublk_drv.c`

6.3.4.1 Description

During fuzzing with *Syzkaller*, X41 triggered a kernel warning regarding task handling in the ublk block driver, as shown in listing 6.16.

```
1 task && task != io->task
2 WARNING: drivers/block/ublk_drv.c:1938 at ublk_uring_cmd_cancel_fn drivers/block/ublk_drv.c:1938
  ↪ [inline], CPU#1: kworker/u8:18/5309
3 WARNING: drivers/block/ublk_drv.c:1938 at ublk_ch_uring_cmd+0xe1/0x2c0 drivers/block/ublk_drv.c:2
  ↪ 548, CPU#1: kworker/u8:18/5309
```

Listing 6.16: Excerpt of Warning Message

Jens Axboe identified the corresponding edge case involving task cancellation conditions.

The ublk block driver uses *io_uring* functionality, but is a separate component and not in scope of the *io_uring* subsystem.

6.3.4.2 Remediation

Jens Axboe provided a patch to fix this issue.[24]

[24] https://git.kernel.org/pub/scm/linux/kernel/git/axboe/linux.git/commit/?h=block-7.1&id=845db023a8aeba8b14315a846dcfba31ee727fb1

6.3.5 IOUR-CR-26-104: NULL Pointer Dereference in ublk Queue Commands

Component: `drivers/block/ublk_drv.c`

6.3.5.1 Description

During fuzzing with *Syzkaller*, X41 triggered kernel crashes in the ***ublk_queue_rq()*** and ***ublk_queue_rqs()*** functions of the ublk block driver, as shown in listing 6.17.

```
BUG: kernel NULL pointer dereference, address: 0000000000000018
#PF: supervisor write access in kernel mode
#PF: error_code(0x0002) - not-present page
PGD 0 P4D 0
Oops: Oops: 0002 [#1] SMP
```

Listing 6.17: Excerpt of ublk Crash Log

The affected logic is related to the partition scanning functionality, which cannot be triggered by unprivileged users according to the maintainer. X41 therefore does not regard this as a security issue, and did not further investigate the bug.

The ublk block driver uses *io_uring* functionality, but is a separate component and not in scope of the *io_uring* subsystem.

6.3.5.2 Remediation

During the audit, Ming Lei created a patch fixing this issue.[25] The patch synchronizes the two paths using *cancel_lock* of the *ublk_queue* data structure.

[25] https://lore.kernel.org/linux-block/20260508123746.242018-1-tom.leiming@gmail.com/T/#u

6.3.6 IOUR-CR-26-105: False Positive Lock Assertion in mm Subsystem

Component: `mm/hugetlb.c`

6.3.6.1 Description

During fuzzing with *Syzkaller*, X41 triggered a custom ***assert()*** in the *mm* memory management subsystem under special conditions involving hugepages. X41 forwarded a proof-of-concept C program for the issue to the *mm* maintainers. The issue was caused by a change that led to a race condition where a code path could trigger the assertion. However, the race condition does not pose a security vulnerability and merely uncovered that assumptions around the assertion have changed.

The issue is present in the Linux kernel versions 6.16 - 7.0.

6.3.6.2 Remediation

The patch[26] implemented by the maintainers asserts the locks conditionally.

[26] `https://lore.kernel.org/linux-mm/20260513085658.45264-1-ljs@kernel.org/`

6.3.7 IOUR-CR-26-106: Use-after-free in NFS Exports

Component: `fs/nfsd/export.c`

6.3.7.1 Description

During fuzzing with *Syzkaller* on kernel 7.1-rc4 to test upstreamed patches, X41 triggered a use after free condition caused by a change in the *nfs* subsystem. X41 forwarded a proof-of-concept C program for the issue to the *nfs* maintainers, who were already aware of the issue.[27]

Since this is a known issue and a fix is available, X41 decided to report this as an informational note.

6.3.7.2 Remediation

The *nfs* maintainers already posted a patch fixing a similar issue as discussed on the *nfs* mailing list.[28] X41 has verified that the issue does not reproduce on kernel versions 7.1.3 and 7.3.

[27] `https://lore.kernel.org/all/10019b42-4589-4f9f-8d5b-d8197db1ce3c@huawei.com/`
[28] `https://lore.kernel.org/all/10019b42-4589-4f9f-8d5b-d8197db1ce3c@huawei.com/`

7 About OSTIF

The Open Source Technology Improvement Fund (OSTIF) is dedicated to resourcing and managing security engagements for open source software projects through partnerships with corporate, goverment, and non-profit donors. We bridge the gap between resources and security outcomes, while supporting and championing the open source community whose efforts underpin our digital landscape.

Over the past ten years, OSTIF has been responsible for the discovery of over 800 vulnerabilities, (121 of those being Critical/High), over 13,000 hours of security work, and millions of dollars raised for open source security. Maximizing output and security outcomes while minimizing labor and cost for projects and funders has resulted in partnerships with multi-billion dollar companies, top open source foundations, government organizations, and respected individuals in the space. Most importantly, we've helped over 120 projects and counting improve their security posture.

Our directive is to support and enrich the open source community through providing public-facing security audits, educational resources, meetups, tooling, and advice. OSTIF's experience postions us to be able to share knowledge of auditing with maintainers, developers, foundations, and the community to further secure our infrastructure in a sustainable manner.

We are a small team working out of Chicago, Illinois. Our website is `ostif.org`. You can follow us on social media to keep up to date on audits, conferences, meetups, and opportunities with OSTIF, or feel free to reach out directly at contact@ostif.org or our Github.

Derek Zimmer, Executive Director
Amir Montazery, Managing Director
Helen Woeste, Communications and Community Manager
Tom Welter, Project Manager

8 About X41 D-Sec GmbH

X41 D-Sec GmbH is an expert provider for application security and penetration testing services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world-class security experts enables X41 D-Sec GmbH to perform premium security services.

X41 has the following references that show their experience in the field:

- Source code audit of ISC BIND9 DNS server[1]
- Source code audit of the Git source code version control system[2]
- Review of the Mozilla Firefox updater[3]
- X41 Browser Security White Paper[4]
- Review of Cryptographic Protocols (Wire)[5]
- Identification of flaws in Fax Machines[6,7]
- Smartcard Stack Fuzzing[8]

The testers at X41 have extensive experience with penetration testing and red teaming exercises in complex environments. This includes enterprise environments with thousands of users and vendor infrastructures such as the Mozilla Firefox Updater (Balrog).

Fields of expertise in the area of application security encompass security-centered code reviews, binary reverse-engineering and vulnerability-discovery. Custom research and IT security consulting, as well as support services, are the core competencies of X41. The team has a strong technical background and performs security reviews of complex and high-profile applications such as Google Chrome and Microsoft Edge web browsers.

X41 D-Sec GmbH can be reached via `https://x41-dsec.de` or `mailto:info@x41-dsec.de`.

[1] `https://x41-dsec.de/news/security/research/source-code-audit/2024/02/13/bind9-security-audit/`
[2] `https://x41-dsec.de/security/research/news/2023/01/17/git-security-audit-ostif/`
[3] `https://blog.mozilla.org/security/2018/10/09/trusting-the-delivery-of-firefox-updates/`
[4] `https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf`
[5] `https://www.x41-dsec.de/reports/Kudelski-X41-Wire-Report-phase1-20170208.pdf`
[6] `https://www.x41-dsec.de/lab/blog/fax/`
[7] `https://2018.zeronights.ru/en/reports/zero-fax-given/`
[8] `https://www.x41-dsec.de/lab/blog/smartcards/`

Acronyms

API Application Programming Interface 15, 31

ASLR Address Space Layout Randomization 17

CPU Central Processing Unit 10, 15, 17

CVE Common Vulnerabilities and Exposures 10

CVSS Common Vulnerability Scoring System 11, 12, 13, 14

CWE Common Weakness Enumeration 14

DMA Direct Memory Access 21

DoS Denial of Service 26

DSL Domain-specific Language 18, 19

ID Identifier 10

OS Operating System 18

PUBLIC - CC BY-SA 4.0 License