Дикий Александр Борисович
Студент
РГУ нефти и газа (НИУ) имени И.М. Губкина

Журавлев Глеб Дмитриевич
Студент
РГУ нефти и газа (НИУ) имени И.М. Губкина

АНАЛИЗ СОСТОЯНИЙ ГОНКИ (RACE CONDITIONS) ПРИ ПАРАЛЛЕЛЬНОМ ДОСТУПЕ НА ЧТЕНИЕ/ЗАПИСЬ ЧЕРЕЗ /PROC: СТАТИЧЕСКИЙ АНАЛИЗ И ПРАКТИЧЕСКАЯ ДЕМОНСТРАЦИЯ УЯЗВИМОСТЕЙ

Аннотация: В работе рассматриваются проблемы синхронизации доступа к данным при параллельном доступе на чтение и запись через виртуальную файловую систему /proc в ядре Linux. Цель работы – оценить, способен ли статический анализатор обнаруживать race conditions в подсистеме /proc Linux на синтетических и реальных примерах, а также предложить комбинированную методику для повышения эффективности обнаружения. С помощью Smatch выполнен анализ обработчиков /proc в ядре дистрибутива ALT Linux, а также создан специальный уязвимый модуль с преднамеренным отсутствием синхронизации и многопоточное приложение-эксплойт. Для подтверждения актуальности проблемы проведён эксперимент с реальной уязвимостью CVE-2025-39866 (use-after-free использование после освобождения в подсистеме writeback). Результаты показывают, что Smatch способен выявлять отдельные классы ошибок синхронизации (например, missing unlock), но не обнаруживает гонки при полном отсутствии блокировок и в сложных межпроцедурных сценариях. Предложенная двухэтапная методика может быть использована при аудите безопасности модулей ядра и в образовательных целях.

Abstract: This paper addresses data access synchronization issues during concurrent read/write operations through /proc pseudo filesystem in the Linux kernel. The aim is to evaluate whether the static analyzers can detect race conditions in the /proc subsystem using synthetic and real-world examples, and to propose a combined methodology to improve detection effectiveness. Smatch is used to analyze /proc handlers in the ALT Linux kernel, and a deliberately unsynchronized kernel module with a multithreaded exploit is created. To demonstrate real-world relevance, an experiment with CVE-2025-39866 (a use-after-free in the writeback subsystem) is conducted. The results show that Smatch can find certain classes of synchronization errors (e.g., missing unlocks) but fails to detect races due to complete absence of locking or complex cross-function scenarios. The proposed two-step methodology can be used for kernel module security auditing and educational purposes.

Ключевые слова: Linux, ядро, состояние гонки, race condition, /proc, синхронизация, статический анализ, Smatch, CVE-2025-39866, уязвимость, ALT Linux.

Keywords: Linux, kernel, race condition, /proc, synchronization, static analysis, Smatch, CVE-2025-39866, vulnerability, ALT Linux.

Виртуальная файловая система /proc в операционной системе Linux предоставляет удобный механизм для обмена данными между ядром и пользовательским пространством. Многие драйверы и модули ядра экспортируют свои интерфейсы через файлы /proc, позволяя прикладным программам читать состояние подсистем или изменять параметры ядра с помощью стандартных системных вызовов read() и write(). Однако, если разработчик модуля явно не защищает разделяемые данные, при конкурентном доступе нескольких потоков пользовательского пространства к одному и тому же proc-файлу возникает состояние гонки.

Проблема усугубляется тем, что типичные обработчики read и write в модулях ядра часто работают с глобальными буферами, счётчиками и указателями. Отсутствие синхронизации приводит к повреждению данных ядра,

чтению несогласованной информации или, в худшем случае, к панике системы и отказу в обслуживании.

Для экспорта файла в виртуальную файловую систему /proc модуль ядра выполняет несколько стандартных действий:

- вызывает proc_create() с указанием имени файла, прав доступа и структуры struct file_operations;
- в file_operations заполняет указатели на функции-обработчики: .read (чтение из файла), .write (запись), иногда .open и .release;
- в этих обработчиках реализует логику обмена данными между ядром и пользовательским пространством.

Если два потока пользовательского процесса одновременно выполняют системные вызовы read() и write() к одному файлу /proc/myfile, то в ядре эти вызовы могут исполняться параллельно на разных ядрах процессора или быть прерваны по таймеру на одном ядре. Поскольку обработчики зачастую не используют блокировки, возможны несколько сценариев.

Первый сценарий — гонка чтения и записи. Писатель начинает копирование данных в shared_buffer, но в середине операции его вытесняет читатель. Читатель видит частично обновлённый буфер, содержащий новые и старые данные. При этом значение buffer_len может уже указывать новую длину, что приведёт к выходу за границы при копировании в пользовательское пространство.

Второй сценарий — гонка двух записей. Два писателя одновременно изменяют буфер и длину, в результате итоговое состояние оказывается несогласованным: длина от одного писателя, содержимое от другого.

Третий сценарий — гонка с частичным обновлением. Многие простые операции, например, buffer_len = len, не являются атомарными на многих архитектурах (требует нескольких инструкций), поэтому читатель может прочитать промежуточное значение.

Описанные сценарии являются следствием ошибочных представлений разработчиков, например, что файлы /proc обрабатываются последовательно или что атомарных операций достаточно для любых типов данных.

В реальности любое обращение к разделяемым изменяемым данным без блокировок создаёт состояние гонки. Блокировки, например, mutex_lock/mutex_unlock, гарантируют, что критическая секция выполняется только одним потоком в единицу времени, но все равно не защищают от неправильного использования самих блокировок, что может спровоцировать появление новых гонок.

Для демонстрации опасности состояния гонки проведены несколько экспериментов. На первом этапе эксперимента была предпринята попытка выявления реальных состояний гонки в коде ядра дистрибутива ALT Linux (версия ядра 6.12.74-6.12-altl) с использованием статического анализатора Smatch.

Выбор инструмента обусловлен его способностью выполнять глубокий анализ потоков данных и межпроцедурный анализ. Это позволяет Smatch отслеживать состояние переменных, блокировок и указателей не только внутри одной функции, но и при их передаче между разными частями кода. Такие возможности критически важны для обнаружения состояний гонки, где некорректная синхронизация может происходить при вызове одной функции из другой.

Однако у него есть существенные ограничения. Smatch успешно ищет несоответствия в использовании примитивов синхронизации, например, missing unlock, двойная блокировка, нарушение порядка, но не моделирует разделяемую память без блокировок полностью. Таким образом, гонки, возникающие при полном отсутствии защиты, не являются целевой областью Smatch. Это ограничение принципиально для всех статических анализаторов, не имеющих аннотаций о разделяемых данных таких как __shared__ или guarded_by.

На рисунке 1 представлен запуск smatch для анализа /proc и смежных с ней директорий и файлов.

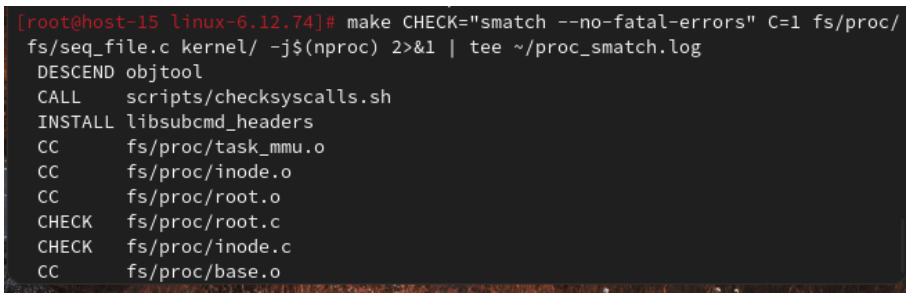

Рисунок 1 – Запуск smatch

При анализе полученных логов (рисунок 2), не было обнаружено явного указания на наличие потенциальной race condition.

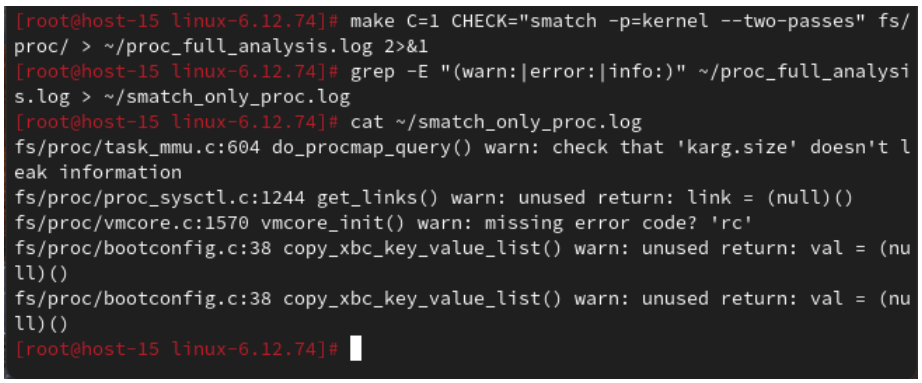

Рисунок 2 – Анализ полученных ошибок и предупреждений

Это не означает отсутствия уязвимостей типа race conditions в ядре, а объясняется особенностями современной реализации proc-файлов.

Начиная с ядер версии 5.6, инициализация структур file_operations для proc-файлов выполняется с помощью макросов, таких как DEFINE_PROC_OPS(), DEFINE_PROC_SHOW_ATTRIBUTE() и PROC_OPS_RW(). Эти макросы раскрываются в определение структуры на этапе препроцессора, ограничивая тем самым использование многих статических анализаторов, использующих поиск синтаксических и семантических шаблонов в коде, как например Coccinelle.

Помимо синтаксических трудностей, связанных с анализаторами на подобии Coccinelle, использование рассмотренного в данной работе Smatch, также не приводит к обнаружению состояний гонки в подсистеме /proc современных ядер. Это обусловлено несколькими причинами.

Во-первых, /proc является одним из наиболее отлаженных компонентов ядра, в котором классические блокировки на основе мьютексов и спин-

блокировок были в значительной степени заменены механизмами без блокировок (lock-free), в частности RCU (Read-Copy-Update) и внутренней синхронизацией интерфейса seq_file [15].

Во-вторых, как упоминалось выше, Smatch в основном ищет несоответствия в использовании блокировок, но не анализирует полностью несинхронизированный доступ, так как не имеет модели «разделяемые данные без защиты». Вследствие этого даже при корректной предобработке макросов статический анализ реального кода /proc не выдаёт значимых предупреждений о состояниях гонки.

Чтобы продемонстрировать способность Smatch выявлять ошибки синхронизации, был создан тестовый модуль. Он содержит общий буфер и счётчик длины, а обработчики read и write намеренно не используют синхронизацию. В него также был добавлен мьютекс, но намеренно допущена ошибка, а именно, в одном из путей обработки ошибок отсутствует вызов mutex_unlock(). Писатель копирует данные в shared_buffer, обновляет buffer_len, а читатель одновременно обращается к тем же переменным, что и вызывает гонку.

Для эксплуатации уязвимости было разработано многопоточное приложение пользовательского пространства на языке C с использованием pthread. Эксплойт создаёт два потока:

- поток-писатель непрерывно записывает в /proc/test_race строку фиксированного размера;
- поток-читатель непрерывно читает из того же файла.

Для увеличения вероятности пересечения потоков использовались циклы без синхронизации и вызовы sched_yield().

Smatch успешно обнаружил проблемный модуль и выдал предупреждение (рисунок 3).

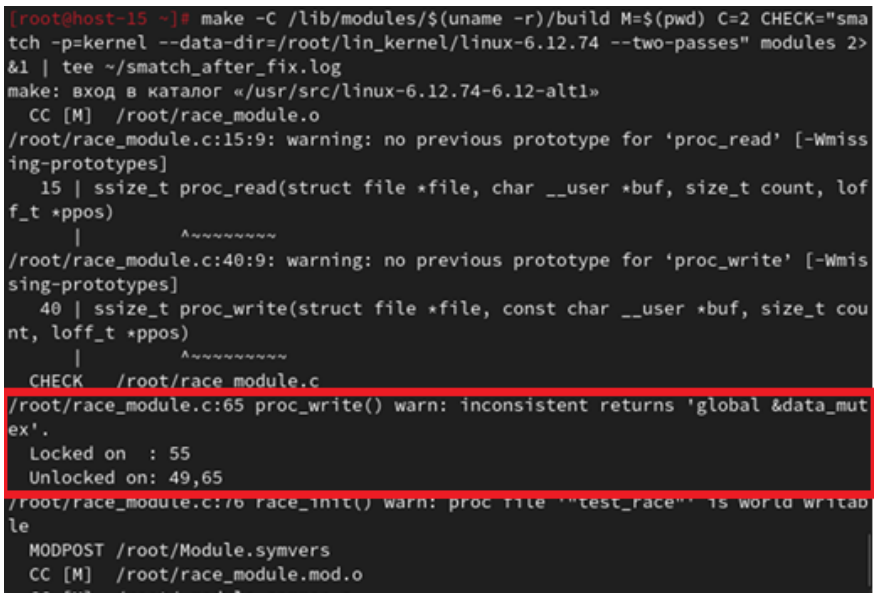

Рисунок 3 – Обнаружение Smatch ошибки синхронизации (неразблокированный мьютекс) в модифицированном модуле

Далее, был проведен тестовый запуск созданного эксплойта. Запуск эксплойта продемонстрирован на рисунке 4.

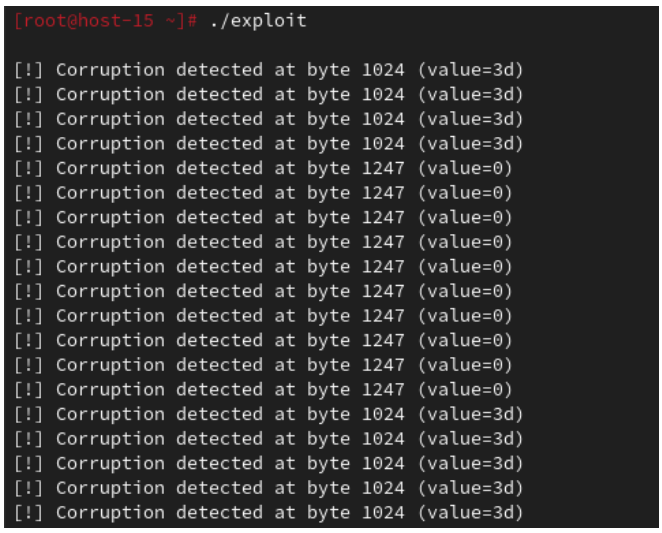

Рисунок 4 – Запуск эксплойта

Вызов эксплойта против тестового модуля спровоцировал механизм защиты CONFIG_HARDENED_USERCOPY, присутствующий в современных версиях ядра (рисунок 5).

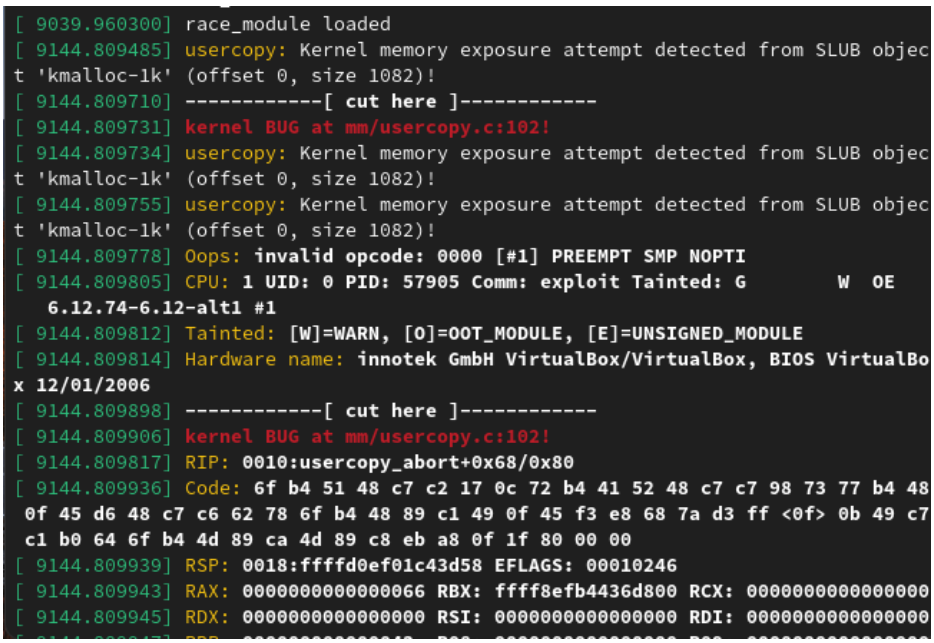

Рисунок 5 – Вывод dmesg

Это механизм безопасности, который проверяет, что при копировании данных из ядра в пользовательское пространство (copy_to_user) память, откуда читаются данные, действительно является выделенным объектом в куче и не является мусором или уже освобождённой областью.

В выводе сообщений ядра замечены многократные сообщения «Oops: invalid opcode» и «---[end trace]---« [14]. Это означает, что процесс, вызвавший ошибку, был убит. Также интересны сообщения вида «CPU: 1 UID: 0 PID: 57905», изображённые на рисунке 7.

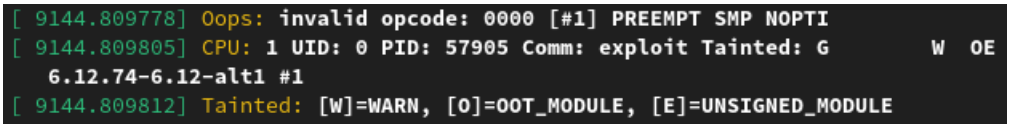

Рисунок 6 – Сообщения ядра

Система зарегистрировала фатальную ошибку в процессе exploit, но поскольку это произошло в контексте системного вызова read, ядро просто завершило поток с сигналом SIGSEGV или SIGKILL, а не уронило всю ОС. Это поведение характерно для современных ядер, позволяющее изолировать сбой в рамки процесса-нарушителя, если это возможно.

Для обоснования актуальности проблемы и демонстрации существования реальных состояний гонки в production-коде ядра Linux был проведён анализ

уязвимости CVE-2025-39866 (use-after-free в подсистеме writeback) [4]. Уязвимость проявляется в ядрах версий < 6.12.16 и возникает при конкурентном доступе к структуре bdi_writeback в функции __mark_inode_dirty() [13].

Гонка возникает между двумя соревнующимися потоками исполнения в подсистеме writeback, происходит это по следующему механизму:

1. Запись информации происходит при вызове функции __mark_inode_dirty(), который читает указатель на inode->i_wb(процесс записи информации о том какие «грязные» inode могут быть использованы для записи текущего потока)
2. Переключение и сбитие синхронизации исполнения потоков происходит при вызове inode_switch_wbs_work_fn() заменяет исполняемый поток записи по указателю на новый контекст writeback и освобождает старый через wb_put_many().

При отсутствии надёжной рабочей синхронизации между этими двумя потоками, может произойти сценарий, что при попытке записи __mark_inode_dirty() может обратиться к структуре уже освобожденной функцией wb_put_many(), из-за чего возникает ситуация использования страницы после освобождения.

Для демонстрации и вызова уязвимости был создан упрощенный модуль ядра [2], создающий два соревнующихся потока, работающих по описанному ранее принципу. Логика модуля следующая:

1. Поток cve_thread_1 циклически обращается к полю list_lock структуры bdi_writeback, предоставляя доступ к двуи конкурирующим путям
2. Поток cve_thread_1 вызывает __mark_inode_dirty(), который вызывает wb_put_many, который может создать уязвимость.

Модуль использует нативные средства и вызовы ядра Линукс из-за чего при попытке провести статический анализ обнаружение данной уязвимости может быть затруднено, потому что прямой работы с памятью и блокировками нет. Подобный паттерн может быть замаскирован под легитимный системный

модуль, например, оптимизатор записи fastwrite.ko, что демонстрирует серьёзность угрозы.

Сразу после загрузки модуля в ядро (рисунок 7) инициируются конкурентные потоки.

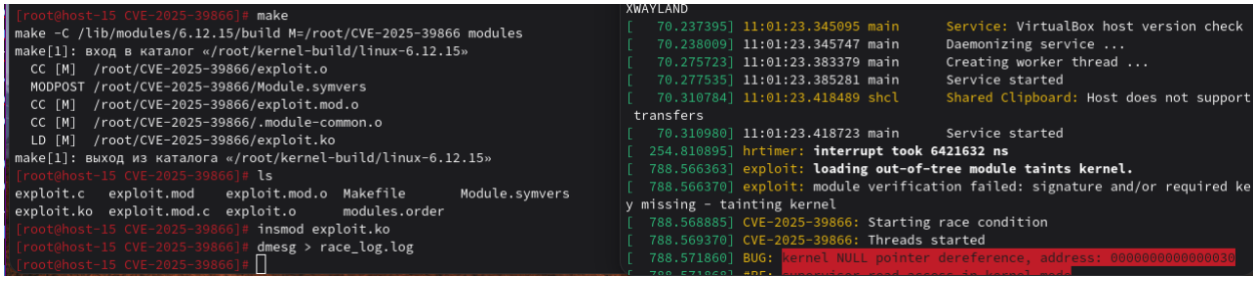

Рисунок 7 – Запуск эксплойта и инициализация потоков

Через несколько секунд выполнения система аварийно завершается (рисунок 8) с ошибкой NULL pointer dereference в функции __mark_inode_dirty() [14].

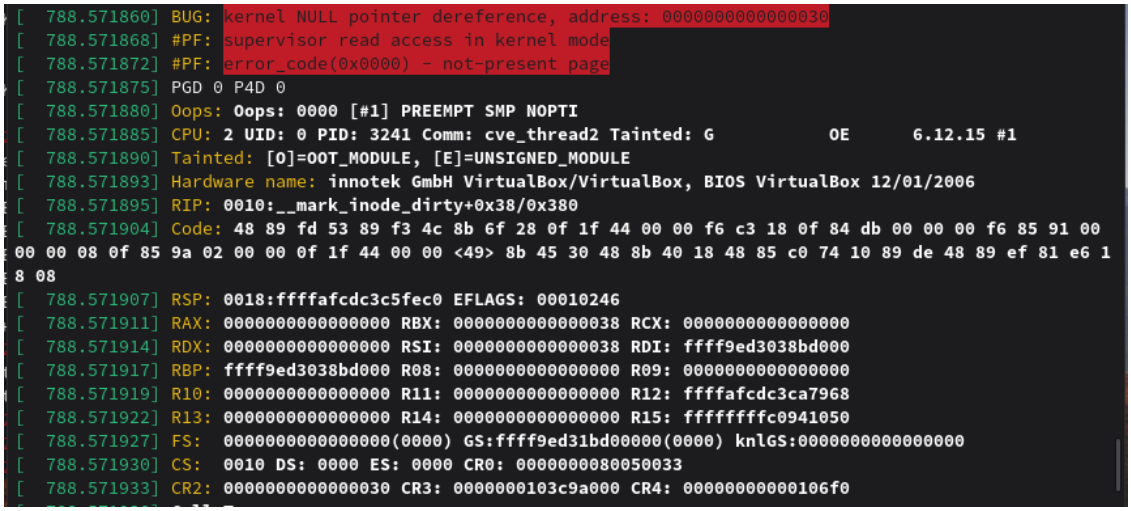

Рисунок 8 – Kernel NULL pointer dereference в __mark_inode_dirty

Анализ дампа показывает:

- RIP: __mark_inode_dirty+0x30/0x380 точка сбоя в ядерной функции;
- CR2: 0x0000000000000030 обращение по нулевому адресу;
- PID: поток cve_thread2 явился причиной сбоя;
- Tainted: OE загружен внешний модуль.

Call Trace демонстрирует цепочку вызовов от загруженного модуля к ядерной функции (рисунок 9), подтверждая, что гонка возникает именно между указанными конкурирующими путями.

Рисунок 9 – Call Trace путь от модуля к функции сбоя

Полученный сбой подтверждает наличие race condition в подсистеме writeback ядра 6.12.15. Поток, вызвавший mark_inode_dirty(), обратился к полю inode->i_wb, которое в момент вызова содержало недействительный указатель. Это произошло потому, что асинхронный механизм переключения writeback-контекста освободил старую структуру раньше, чем функция __mark_inode_dirty() завершила работу с ней.

Ввиду того что вызываемые функции и операции внутри описанного модуля являются легитимными, а сами функции не напрямую могут вызвать асинхронизацию и нарушение памяти, статический анализ существенных результатов не даст, как и в случае с proc-модулем. Нарушения работы модуля возможно обнаружить только при его фактическом запуске по опередленному времени исполнения атомарных операций, что требует точности и работы с динамикой. Из чего следует вывод в необходимости анализа разрабатываемых модулей и программ в динамике. В данной работе основной фокус был на попытке обнаружения данного семейства уязвимостей статическим анализом, ввиду распространённости практики избыточности статистического анализа при проверке на уязвимости, который, как показывают наши опыты, не достаточны для выявления определенных типов уязвимостей.

Использование динамического анализа не гарантирует обнаружения всех типов гонок, но значительно увеличивает вероятность обнаружения данной

уязвимости, потому как даёт более полную картину о состоянии исполняемых процессов и больше признаков для нахождения ошибки.

Заключение

В работе рассмотрены проблемы состояний гонки при параллельном доступе на чтение/запись к данным через файлы виртуальной файловой системы /proc в ядре Linux, а также в подсистеме записи страничного кэша в постоянное хранилище writeback на примере CVE-2025-39866.

Проведённые эксперименты продемонстрировали недостаточность статических анализаторов при попытке обнаружения гонки при полной асинхронной работе функций, но успешно выявляет ошибки использования мьютексов. Уязвимость CVE-2025-39866 осталась не обнаружена ввиду сложного межпроцедурного характера гонки, все случаи запуска уязвимых исполняемых модулей приводили к падению ядра.

Эксперименты подтвердили необходимость использования динамического анализа для обнаружения несинхронизированных потоков и вызовов функций. Рекомендуется двухэтапный анализ модулей ядра: статический анализ эффективен для быстрого выявления типовых ошибок, а также неправильного использования примитивов синхронизации, по завершению статического анализа проводить динамическое тестирование для обнаружения гонок и других процессов (RCU,seq_file) и ошибок, не покрываемых статикой.

Литература

1. The kernel’s command-line parameters [Электронный ресурс] // The Linux Kernel documentation. – URL: https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html (дата обращения: 13.04.2026).
2. Сборка модулей ядра [Электронный ресурс] // ALT Linux Wiki. – URL: https://www.altlinux.org/Сборка_модулей_ядра (дата обращения: 13.04.2026).
3. Linux Kernel Locking Documentation [Электронный ресурс] // The Linux Kernel documentation. – URL: https://origin.kernel.org/doc/html/v6.6/locking/index.html (дата обращения: 13.04.2026).

4. CVE-2025-39866 Detail // https://nvd.nist.gov/ URL: https://nvd.nist.gov/vuln/detail/CVE-2025-39866 (дата обращения: 12.04.2026).
5. Уймин, А. Г. Цифровые двойники сетевых инфраструктур: точность, методы и практические решения / А. Г. Уймин // Радиотехнические и телекоммуникационные системы. – 2023. – № 3(51). – С. 44-52. – DOI 10.24412/2221-2574-2023-3-44-52. – EDN QUSITK.
6. Dossche N., Abrath B., Coppens B. A Context-Sensitive, Outlier-Based Static Analysis to Find Kernel Race Conditions //arXiv preprint arXiv:2404.00350. – 2024.
7. Андрианов П. С. Анализ корректности синхронизации компонентов ядра операционных систем // Труды ИСП РАН. 2019. №5. URL: https://cyberleninka.ru/article/n/analiz-korrektnosti-sinhronizatsii-komponentov-yadra-operatsionnyh-sistem (дата обращения: 04.05.2026).
8. Андрианов П. С., Мутилин В. С., Хорошилов А. В. Метод легковесного статического анализа для поиска состояний гонок // Труды ИСП РАН. 2015. №5. URL: https://cyberleninka.ru/article/n/metod-legkovesnogo-staticheskogo-analiza-dlya-poiska-sostoyaniy-gonok (дата обращения: 04.05.2026)
9. Upadhyay A., Laxmi V., Naval S. Navigating the Concurrency Landscape: A Survey of Race Condition Vulnerability Detectors //arXiv preprint arXiv:2312.14479. – 2023.
10. Заборовский Н. В., Тормасов А. Г. Статическое обнаружение гонок в коде, содержащем ветвления и циклы // Прикладная информатика. 2011. №6 (36). URL: https://cyberleninka.ru/article/n/staticheskoe-obnaruzhenie-gonok-v-kode-soderzhaschem-vetvleniya-i-tsikly (дата обращения: 04.05.2026).
11. Андрианов П. С., Мутилин В. С., Хорошилов А. В. Конфигурируемый метод поиска состояний гонок в операционных системах с использованием предикатных абстракций // Труды ИСП РАН. 2016. №6. URL: https://cyberleninka.ru/article/n/konfiguriruemyy-metod-poiska-sostoyaniy-gonok-v-operatsionnyh-sistemah-s-ispolzovaniem-predikatnyh-abstraktsiy (дата обращения: 03.05.2026).

12. Li T. et al. SPATA: Effective OS bug detection with summary-based, alias-aware, and path-sensitive typestate analysis //ACM Transactions on Computer Systems. – 2024. – Т. 42. – №. 3-4. – С. 1-40.
13. Vulnerability CVE-2025-39866: Information // ALT LINUX TEAM URL: https://packages.altlinux.org/en/vuln/CVE-2025-39866 (дата обращения: 01.05.2026).
14. Oops! Debugging Kernel Panics // Linux Journal URL: https://www.linuxjournal.com/content/oops-debugging-kernel-panics-0 (дата обращения: 01.05.2026).
15. What is RCU? // The Linux Kernel URL: https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html (дата обращения: 01.05.2026).

Литература

1. The kernel's command-line parameters [Электронна ресурс] // The Linux Kernel documentation. - URL: https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html (атшазынархара амш: 13.04.2026).
2. Аядро амодульква радгалра [Электронна ресурс] / / ALT Linux Wiki. - URL: https://www.altlinux.org/Сборка_модулей_ядра (атшазынархара амш: 13.04.2026).
3. Linux Kernel Locking Documentation [Электронна ресурс] / / The Linux Kernel documentation. - URL: https://origin.kernel.org/doc/html/v6.6/locking/index.html (атшазынархара амш: 13.04.2026).
4. CVE-2025-39866 Detail // https://nvd.nist.gov / URL: https://nvd.nist.gov/vuln/detail/CVE-2025-39866 (атшазынархара амш: 12.04.2026).
5. Уймин, а.Г. Ацифра аглвазаква асеть инфраструктураква: атамамра, агламальква игьи апрактика щаквыргылраква / а. Г. Уймин // Радиотехническа игьи телекоммуникационна системаква. – 2023. – № 3(51). - 44-52 ан. – DOI 10.24412/2221-2574-2023-3-44-52. – EDN QUSITK.
6. Dossche N., Abrath B., Coppens B. A Context-Sensitive, Outlier-Based Static Analysis to Find Kernel Race Conditions //arXiv preprint arXiv:2404.00350. – 2024.

7. Андрианов П.С. операционна системаква рквта акомпонентква синхронизация тамамта Алыргара // ИСП РАН Анхараква. 2019. №5. URL: https://cyberleninka.ru/article/n/analiz-korrektnosti-sinhronizatsii-komponentov-yadra-operatsionnyh-sistem (атшазынархара амш: 04.05.2026).
8. Андрианов П.С., Мутилин В. С., Хорошилов А. В. апхъанысраква раъаща азыпшгlара ахъаз йласу астатика алыргара Агlамаль // ИСП РАН Анхараква. 2015. №5. URL: https://cyberleninka.ru/article/n/metod-legkovesnogo-staticheskogo-analiza-dlya-poiska-sostoyaniy-gonok (атшазынархара амш: 04.05.2026)
9. Upadhyay A., Laxmi V., Naval S. Navigating the Concurrency Landscape: A Survey of Race Condition Vulnerability Detectors //arXiv preprint arXiv:2312.14479. – 2023.
10. Заборовский Н. В`., Тормасов А`. Г`. Статическое обнаружение гонок в коде, содержащем ветвления и циклы // Прикладная информатика. 2011. №6 (36). URL: https://cyberleninka.ru/article/n/staticheskoe-obnaruzhenie-gonok-v-kode-soderzhaschem-vetvleniya-i-tsikly (дата обращения: 04`.05`.2026).
11. Андрианов П.С., Мутилин В. С., Хорошилов А. В. Конфигурировать зчпауа агlамаль апхъанысраква раъаща азыпшгlара операционна системаква рпны предикатна абстракцияква гladрысабапуамца // ИСП РАН Анхараква. 2016. №6. URL: https://cyberleninka.ru/article/n/konfiguriruemyy-metod-poiska-sostoyaniy-gonok-v-operatsionnyh-sistemah-s-ispolzovaniem-predikatnyh-abstraktsiy (атшазынархара амш: 03.05.2026).
12. Li T. et al. SPATA: Effective OS bug detection with summary-based, alias-aware, and path-sensitive typestate analysis //ACM Transactions on Computer Systems. – 2024. - Т.42. – №. 3-4. - Ан.1-40.
13. Vulnerability CVE-2025-39866: Information // ALT LINUX TEAM URL: https://packages.altlinux.org/en/vuln/CVE-2025-39866 (атшазынархара амш: 01.05.2026).
14. Oops! Debugging Kernel Panics // Linux Journal URL: https://www.linuxjournal.com/content/oops-debugging-kernel-panics-0 (атшазынархара амш: 01.05.2026).

15. What is RCU? // The Linux Kernel URL: https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html (атшазынархара амш: 01.05.2026).