DOI: 10.15514/ISPRAS-2024-36(3)-12

Платформа автоматизации фаззинг-тестирования компонентов операционной системы

Е.П. Сураев, ORCID: 0009-0005-8454-3241 <*esuraev@astralinux.ru*>
В.В. Егорова, ORCID: 0000-0003-1286-3631 <*vegorova@astralinux.ru*>
А.С. Панов, ORCID: 0000-0002-0046-0766 <*apanov@astralinux.ru*>

ПАО Группа Астра,
117105, г. Москва, Варшавское ш., д.26, стр.11

Аннотация. Автоматизация процессов тестирования и анализа безопасности играет важную роль при разработке программного обеспечения, поскольку позволяет обнаруживать и устранять уязвимости на ранних этапах. В данной статье представлены результаты разработки автоматизированной платформы фаззинг-тестирования, а также ее интеграция с платформой обработки и хранения результатов различных средств анализа безопасности. Разработанная платформа интегрирует инструменты анализа безопасности в единую систему тестирования, встраиваемую в непрерывные процессы интеграции (CI). Предложенная платформа не только упрощает и ускоряет процессы тестирования и анализа, но и повышает точность обнаружения уязвимостей за счет агрегации результатов и применения алгоритмов машинного обучения для разметки и приоритизации обнаруженных ошибок. Такой подход позволяет разработчикам своевременно идентифицировать и исправлять уязвимости, что способствует созданию более надежных и безопасных программных продуктов.

Ключевые слова: динамический анализ; анализ безопасности; автоматизация фаззинг-тестирования; фаззинг.

Для цитирования: Сураев Е.П., Егорова В.В., Панов А.С. Платформа автоматизации фаззинг-тестирования компонентов операционной системы. Труды ИСП РАН, том 36, вып. 3, 2024 г., стр. 167–188. DOI: 10.15514/ISPRAS–2024–36(3)–12.

Platform for automatic fuzzing of OS components

E.P. Suraev, ORCID: 0009-0005-8454-3241 <*esuraev@astralinux.ru*>
V.V. Egorova, ORCID: 0000-0003-1286-3631 <*vegorova@astralinux.ru*>
A.S. Panov, ORCID: 0000-0002-0046-0766 <*apanov@astralinux.ru*>

PJSC Astra Group,
26, Varshavskoe, Moscow, 117105, Russia

Abstract. Automation of security analysis processes plays an important role in software development, because it allows vulnerabilities to be detected and fixed at an early stage. This article presents the development outcomes of an automated fuzz-testing platform, as well as its integration with a platform for processing and storing the results of various security analysis tools. The developed platform integrates security analysis tools into a single testing system embedded in the continuous integration process. The proposed platform not only simplifies and speeds up the testing and analysis processes, but also increases the accuracy of vulnerability detection through results aggregation and the application of machine learning algorithms for marking and prioritizing detected errors. This approach allows developers to identify and correct vulnerabilities in a timely manner, contributing to the creation of more reliable and secure products.

Keywords: dynamic analysis; security analysis; fuzz-testing automation; fuzzing.

For citation: Suraev E.P., Egorova V.V., Panov A.S. Platform for automatic fuzzing of OS components. *Trudy ISP RAN/Proc. ISP RAS*, vol. 36, issue 3, 2024. pp. 167-188 (in Russian). DOI: 10.15514/ISPRAS-2024-36(3) - 12.

1. Введение

При разработке операционной системы, сертифицированной по первому, наивысшему, уровню доверия, безопасность разрабатываемых и поддерживаемых компонентов является критически важным аспектом. В связи с этим, подходы по анализу безопасности программного кода, такие как статический и динамический анализ, включая фаззинг-тестирование, выделяются как ключевые методы обнаружения ошибок и уязвимостей на ранних этапах разработки. Именно комбинация данных подходов покрывает широкий спектр задач тестирования на безопасность.

Несмотря на преимущества перечисленных подходов, они также имеют и ряд недостатков: ручной запуск фаззинг-тестирования, особенно при большом количестве фаззинг-оберток, анализ полученных результатов, а также его масштабирование, может быть сложной задачей, когда речь идет о большом количестве тестируемых компонентов. Статический анализ, в свою очередь, является легко масштабируемым, однако часто приводит к высокому количеству ложных срабатываний (ошибки первого рода) или наоборот, не обнаруживает определенные ошибки (ошибка второго рода), которые, тем не менее, способны обнаружить инструменты динамического анализа. Помимо этого, ручной разбор результатов статического анализа также является достаточно ресурсозатратной задачей, когда речь идет о таком масштабном программном продукте, как операционная система. По этим причинам исследователи уделяют большое внимание интеграции статического и динамического анализа с целью расширения охвата и упрощения процессов разметки результатов.

В ответ на эти вызовы, актуальность автоматизации процессов тестирования и анализа безопасности программного обеспечения также неуклонно возрастает. Автоматизация и совместное применение статического и динамического анализа, агрегация всех результатов, их взаимная приоритизация, позволяет ускорить и облегчить процессы запуска и анализа обнаруженных результатов. Особенно значимо это в контексте интеграции в непрерывный цикл безопасной разработки, где автоматизация может играть решающую роль в обеспечении безопасности, способствуя своевременному обнаружению и устранению ошибок и уязвимостей.

В данной статье мы представляем платформу автоматизации фаззинг-тестирования, встраиваемую в CI и интегрированную с платформой хранения и обработки результатов статического анализа, которая, в свою очередь, предоставляет различный функционал по приоритизации, дедупликации и разметке результатов, в том числе частично автоматизированный с помощью алгоритмов машинного обучения. Помимо этого, платформа хранения и агрегации результатов предоставляет возможность сопоставления результатов различных средств анализа, что положительно влияет на процессы разметки.

Предложенная в данной статье платформа автоматизации фаззинг-тестирования предоставляет возможность унифицировать и объединить под собой циклы тестирования всех разрабатываемых и поддерживаемых компонентов операционной системы в единую систему тестирования и предоставлять все необходимые результаты тестирования различными способами. Помимо этого, разрабатываемая платформа масштабируема и позволяет встроить процессы фаззинга в CI-конвейеры, что является критически важным фактором для обнаружения ошибок на ранних этапах.

Такой подход упрощает процессы анализа безопасности и интегрируется в процессы разработки, что является крайне важным вопросом при разработке надежных и безопасных программных продуктов, соответствующих высшим стандартам безопасности.

2. Обзор существующих решений

На сегодняшний день существует не так много специализированных решений, предоставляющих полноценные платформы по автоматизации фаззинг-тестирования и внедрения этих процессов в цикл безопасной разработки. Зачастую процессы автоматизации анализа безопасности ограничиваются внедрением проверок в CI-конвейеры в рамках системы контроля версий, однако такой подход неудобен в первую очередь для специалистов в области информационной безопасности, которые анализируют полученные результаты после каждого такого прогона, следят за результативностью проведенных процедур анализа и при необходимости вносят коррективы и улучшения с целью повышения эффективности процесса. Помимо этого, при таком подходе сложно оценивать эффективность фаззинга и подготовленных для него оберток, обнаруживать зависания, снижение стабильности работы и другие аспекты, влияющие на результативность проводимой процедуры анализа.

В рамках данного раздела будет приведено сравнение существующих инструментов автоматизации фаззинг-тестирования, при этом предлагается выделить следующие важные критерии сравнения:

- наличие возможности проведения регрессионного тестирования обнаруженных ранее ошибок;
- наличие проверок, определяющих необходимый набор оберток для запуска фаззинга изменений в коде;
- локальный запуск, сторонние зависимости, возможность работы с закрытыми репозиториями с кодом (для проприетарных продуктов);
- поддерживаемые языки программирования (возможность тестирования системного и прикладного программного обеспечения (ПО), входящего в состав ОС семейства Linux);
- удобство использования и настройки (включая создание новых оберток и интеграцию в существующие процессы);
- возможность гибкой настройки параметров фаззинг-тестирования: времени – от коротких запусков до длительных, выбора санитайзеров, инструментов и других опций;

- возможность запуска вручную и по триггерам в системе контроля версий;
- масштабируемость;
- открытый исходный код (учитывается гибкость и возможность самостоятельной настройки и доработки);
- используемые техники и алгоритмы фаззинга, используемые фаззеры, добавление собственных фаззеров, возможность интеграции с другими средствами анализа, системами отслеживания задач, собственными хранилищами, экспорт результатов в стандартных форматах (JSON/SARIF) и др.;
- наличие графического интерфейса для отображения результатов по запускам, статистики фаззинга, покрытия по отдельным целям и компонентам целиком, базы данных ошибок и др.;
- сбор покрытия по конкретным оберткам и консолидированного покрытия по всему проекту;
- длительное хранение всех артефактов фаззинга;
- минимизация и дедупликация обнаруженных ошибок.

Одной из популярных платформ автоматизации фаззинг-тестирования является платформа OSS-Fuzz [1] от Google, использующая в своей основе ClusterFuzz [6]. Этот инструментальный комплекс обеспечивает интеграцию с системами контроля версий, автоматически запуская фаззинг-тестирование на непродолжительное время после каждой фиксации. Благодаря открытому исходному коду, OSS-Fuzz активно модифицируется и адаптируется разработчиками под конкретные задачи и потребности их проектов. Существует также проект OSS-Sydr-Fuzz [2], разработанный ИСП РАН, который расширяет возможности стандартного фаззинга с помощью AFL [3] и libFuzzer [4], доступного в исходном проекте, добавлением собственного инструмента Sydr-fuzz [5] для динамического анализа. Репозиторий OSS-Sydr-Fuzz содержит проекты с обертками из OSS-Fuzz, дополненные также обертками для запуска гибридного фаззинга с помощью Sydr-fuzz.

В платформе OSS-Fuzz отсутствует возможность проведения регрессионного тестирования отдельно от фаззинга, однако зачастую такая необходимость все же возникает, например, для тестирования изменений, устраняющих конкретную, ранее обнаруженную ошибку, с целью убедиться, что она более не воспроизводится. В качестве альтернативы используется обработка всех сохраненных входных данных, в том числе обнаруженных ошибок, перед началом фаззинга. Эти проверки по умолчанию проводятся фаззерами (AFL++ и libFuzzer) с целью построения начальной конфигурации фаззинга и приоритизации входных тестовых примеров. Артефакты, полученные во время предыдущих циклов фаззинга, такие как обнаруженные ошибки, корпуса, начальные тестовые примеры, сохраняются в течение 30 дней и используются для последующих запусков.

Количество оберток для одного проекта может превышать сотни и не всегда оправдано при небольших изменениях в коде запускать даже те из них, что внесенные изменения никоим образом не затрагивают. При наличии поддержки покрытия кода в тестируемом проекте, для фаззинга будут использоваться только те обертки, которые непосредственно взаимодействуют с измененными участками кода. В противном случае, отведенное на фаззинг время будет разделено на все обертки в проекте.

ClusterFuzz, лежащий в основе OSS-Fuzz, возможно запускать локально, однако в связи с тем, что этот проект является разработкой Google, в своей работе он также использует сервисы Google Cloud. Локальный запуск возможен с помощью эмуляторов Google Cloud для тех, кому не нужны предлагаемые им сервисы, такие как веб-интерфейс для доступа к обнаруженным ошибкам, статистике и планировщику задач, облачное хранилище, bigQuery

и Stackdriver. Стоит отметить, что функционал, зависящий от этих сервисов, также может быть ограничен из-за отсутствия поддержки эмулятора. Помимо этого, по той причине, что фаззинг-тестирование проводится по умолчанию на вычислительных мощностях компании Google, а платформа, даже при локальном запуске, не работает с закрытыми репозиториями, проводить тестирование проприетарного ПО с закрытой кодовой базой не представляется возможным.

OSS-Fuzz поддерживает следующие языки программирования:

- C/C++
- Go,
- Rust,
- Python,
- JVM-based (Java, Kotlin, Scala и др.),
- Swift,
- Javascript,

что в полной мере охватывает необходимый набор языков, использующихся при разработке системного ПО и компонентов операционных систем.

Платформа предоставляет возможность добавления собственных проектов с открытым исходным кодом, а также дополнительных оберток для существующих проектов посредством добавления их в основной репозиторий проекта OSS-Fuzz. Предлагается достаточно простая процедура для добавления новых проектов и фаззинг-целей: разработчики могут самостоятельно интегрировать свои проекты, следуя документации и используя предоставляемые в открытом доступе шаблоны и конфигурационные файлы.

На платформе OSS-Fuzz длительность фаззинг-тестирования ограничена параметрами в конфигурационном файле, и не может превышать 6 часов. Это ограничение соответствует максимально возможному времени выполнения задачи в GitHub Actions. Однако практика показывает, что для проектов с большой кодовой базой короткие интервалы тестирования могут быть недостаточными. Фаззеры могут не успевать обеспечить достаточное покрытие кода и сгенерировать необходимые входные данные для выявления всех потенциальных ошибок и уязвимостей. Помимо этого, в AFLplusplus реализован флаг, позволяющий завершать тестирование, если за указанное в качестве параметра количество часов не были обнаружены новые пути. Это полезный функционал, который позволяет производить всеобъемлющее фаззинг-тестирование и при этом минимизировать затраты ресурсов. Также в платформе реализован выбор санитайзеров в соответствии со спецификой и требованиями тестируемого проекта, возможно добавление собственных словарей, настройка переменных окружения. Однако нигде в документации к OSS-Fuzz и ClusterFuzz не указывается возможность настройки параметров фаззинга, таких как ограничение по памяти, времени и другие параметры, обычно передаваемые в качестве аргументов команде запуска фаззинга (запуска цели libFuzzer или непосредственно команде afl-fuzz).

OSS-Fuzz ориентирован в основном на автоматизацию и интеграцию в существующие CI-конвейеры, но также позволяет разработчикам ПО запускать тестирование локально, например, для отладки. Это может быть реализовано с помощью специальных скриптов и подготовленных Docker-контейнеров, предоставляемых OSS-Fuzz, что позволяет имитировать среду, используемую на платформе, и запускать тестирование вручную на своей машине. OSS-Fuzz интегрируется в GitHub Actions, что позволяет автоматически запускать фаззинг-тестирование при создании запроса на слияние или при отправке изменений в отслеживаемую ветку. Платформа автоматически управляет запуском фаззинга для множества целей в рамках одного проекта, однако не позволяет совершить массированный

запуск вручную, в случае необходимости. ClusterFuzz, в свою очередь, предоставляет возможность настройки и запуска фаззинг-тестов вручную, хоть это и не является основным сценарием для использования платформы. На платформе ClusterFuzz возможность ручного запуска фаззинг-целей, не привязанного к автоматизированным триггерам CI, реализуется через веб-интерфейс управления. Необходимо указывать конкретные цели для тестирования, параметры запуска, тестовые корпуса и другие настройки. Также в ClusterFuzz реализована возможность группового управления задачами для ручного запуска сразу всех целей, относящихся к отдельному проекту. В контексте разработки и анализа безопасности операционной системы, включающей в себя большое количество тестируемых проектов, может также понадобиться возможность запуска всех оберток для всех проектов (например, в случае подготовки документов для сертификации или полномасштабного тестирования очередного релиза), однако, судя по доступной документации, такой возможности не предоставляется.

Инфраструктура Google Cloud позволяет автоматически масштабировать количество вычислительных ресурсов в зависимости от потребностей фаззинг-задач. Это обеспечивает достаточную производительность даже для самых требовательных проектов. Помимо этого, использование ресурсов оптимизируется путем динамического распределения задач между доступными вычислительными узлами, гарантируя, что каждая фаззинг-сессия имеет доступ к необходимым ресурсам без излишнего расхода. Система автоматически приоритизирует запуск фаззинг-оберток, уделяя больше внимания новым и недавно измененным участкам кода, что позволяет эффективнее вызывать потенциальные уязвимости на ранних этапах.

ClusterFuzz и OSS-Fuzz являются проектами с открытым исходным кодом, что позволяет пользователям изучать и модифицировать код платформ. Несмотря на это, ни одна из этих платформ не работает для проприетарных проектов с закрытым исходным кодом, так как сначала необходимо настроить систему сборки, в которую затем встраивается одна из платформ.

Данные платформы предоставляют возможность работы с фаззерами AFL++, libFuzzer, Jazzer, Atheris (хотя единственный поддерживаемый движок фаззинга – libFuzzer) и несмотря на то, что эти инструменты покрывают большинство задач по фаззингу проектов, написанных на указанных языках программирования, этого может быть недостаточно в специфических задачах, в том числе с целью повышения эффективности фаззинга, реализации направленного фаззинга для подтверждения результатов статического анализа с использованием специализированных инструментов. При этом добавление собственного инструмента в масштабную инфраструктуру данных проектов затруднительно, так как вся платформа спроектирована именно под работу с движком libFuzzer. Помимо этого, в ClusterFuzz существует ограничение при использовании AFL: платформа не поддерживает функционал минимизации корпуса и минимизации входных данных, вызывающих падения, для этого фаззера. В остальном, никаких дополнительных опций для модификации существующих алгоритмов фаззинга не предоставляется. В качестве инструмента для сбора покрытия используется llvm-cov, и OSS-Fuzz, и ClusterFuzz предоставляют возможность сбора консолидированного покрытия по проекту. Помимо этого, для анализа результатов фаззинга в данных проектах используется Fuzz Introspector [9], который, однако, приспособлен только для работы с фаззинг-целями, адаптированными под libFuzzer. Для целей, созданных специально для фаззинга с использованием AFL++ данный инструмент не подходит.

В качестве хранилища результатов по умолчанию используется Google Cloud Storage, также доступен экспорт данных для внешнего хранилища, однако это требует сложной дополнительной настройки, в том числе для приведения данных к единому формату экспорта. OSS-Fuzz может автоматически создавать отчеты об обнаруженных ошибках в публичных системах отслеживания задач и ошибок, что может быть не применимо для проприетарного ПО. Для интеграции с собственными системами отслеживания задач и ошибок необходимо отдельно модифицировать код и разрабатывать свою интеграцию. Интеграция с другими

средствами анализа (статический, динамический анализ, базы данных уязвимостей и др.) по умолчанию не предоставляется.

Помимо OSS-Fuzz и его аналогов, существуют и другие значимые работы, которые предлагают свои уникальные идеи для улучшения безопасности и надежности программного обеспечения. Так, например, статья [8] рассматривает нюансы интеграции процессов фаззинг-тестирования в CI/CD, акцентируя внимание на достижении оптимального баланса между глубиной тестирования безопасности и поддержанием эффективности процессов разработки. В ней описываются стратегии для селективного фаззинга, чтобы избежать лишних тестов и сэкономить вычислительные ресурсы. Кроме того, в рамках данной работы проведено исследование, как изменение продолжительности кампаний фаззинга влияет на эффективность обнаружения ошибок в контексте CI/CD, предполагая, что даже кратковременные систематические запуски могут выявить значительное количество в том числе критичных уязвимостей. Исследование подчеркивает важность стратегического планирования задач фаззинга в рамках CI/CD для повышения безопасности программного обеспечения без ущерба для скорости разработки.

Другое решение для автоматизации фаззинга – CIFuzz [7] от Code Intelligence, который является частью платформы фаззинга CI Sense [13]. Ранее данный инструмент распространялся свободно и представлял собой продукт с открытым исходным кодом, однако на текущий момент его использование доступно исключительно по платной подписке. Локальный запуск при оплате подписки возможен. Вместе с продуктом в одной из опций подписки поставляется также дополнительный модуль CI Spark [12]. Данный инструмент позволяет находить точки входа в тестируемую программу и генерировать обертки для них, используя большие языковые модели (LLM). Стоит заметить, что Google также работает над подобным проектом – OSS-Fuzz-Gen [39], который, используя специальный конструктор запросов к LLM, позволяет генерировать обертки для непокрытых критичных участков кода. Для определения таких участков используется Fuzz Introspector.

Данный инструмент, в отличие от OSS-Fuzz, реализует возможность проведения регрессионного тестирования отдельно от фаззинга. В таком случае обертки будут запущены со всеми входными данными, сохраненными после предыдущих прогонов фаззинга. Запуск возможен из командной строки или IDE.

CIFuzz по умолчанию запускает все обертки для проекта или только те, что указаны вручную. При этом нет функционала, отслеживающего возможность запуска исключительно тех оберток, которые затрагивают измененные участки кода.

CIFuzz поддерживает следующие языки программирования:

- C/C++,
- Java,
- Javascript/Typescript,
- Go (только для фаззинга API-интерфейсов REST и GRPC на основе Go),

фаззинг реализуется с помощью libFuzzer, Jazzer [10], Jazzer.js [11]. Однако, для фаззинга компонентов операционной системы такого набора недостаточно. Необходима по меньшей мере возможность фаззинга Go-проектов вне контекста Web-интерфейсов, а также Python-проектов. Платформа не предоставляет возможность добавления собственных фаззеров и из-за того, что исходный код является закрытым, это невозможно.

Платформа предоставляет возможность добавления собственных проектов, а также дополнительных оберток для существующих проектов. Предлагается достаточно простая процедура для добавления новых проектов и фаззинг-целей: разработчики могут самостоятельно интегрировать свои проекты, следуя документации и используя

предоставляемые в открытом доступе шаблоны и конфигурационные файлы, а также добавляя дополнительные директивы в систему сборки своего проекта.

В CIFuzz отсутствует гибкая настройка параметров фаззинга, а в документации не отражена возможность даже добавления или отключения определенных санитайзеров. Время фаззинга по умолчанию не регламентировано и никак не настраивается отдельно. Встраивание в CI происходит посредством GitHub Actions, так что здесь также действует ограничение в 6 часов, как максимальное время выполнения задачи.

Ручной запуск целей возможен через IDE или с помощью интерфейса командной строки для проекта на локальной машине, однако для запуска локально в закрытом репозитории необходимо загружать архивы с фаззинг-целями отдельно на платформу. В случае изменения кода проекта или оберток необходимо создавать и загружать эти архивы заново вручную. Данный процесс не автоматизирован.

Что касается принципов организации масштабирования на данной платформе, на официальном сайте и в документации не отражены эти аспекты, однако можно предположить, что задачи фаззинга также динамически распределяются между множеством машин для обработки крупных проектов.

Интеграция данной платформы с собственными хранилищами данных или сторонними инструментами обработки и анализа затруднительна. Простой графический интерфейс предоставляет возможность добавлять проекты, а также отслеживать обнаруженные ошибки, а данные возможно выгрузить в форматах PDF, Word и Excel. Преобразование в общепринятые форматы, такие как JSON или SARIF, невозможно. Сбор покрытия возможен только по одной цели, консолидация покрытия по всему проекту не реализована.

Обнаруженные ошибки не минимизируются и не дедуплицируются. Они отображаются в том числе в графическом интерфейсе. Для обнаруженных ошибок автоматически выставляется уровень критичности. Помимо этого, есть возможность проводить ручное управление обнаруженными ошибками, возможность фильтрации и отслеживания. Для каждой ошибки можно поставить статус и присоединить ее к соответствующей задаче в системе отслеживания задач.

Длительность хранения артефактов фаззинга не регламентирована, однако указано, что метрики, собираемые по конкретному проекту (информация о покрытии, количество существующих оберток, количество запусков фаззинга за последние 7 дней, количество пользователей, вошедших в систему, сумма новых результатов, общее количество устраненных ошибок, которые более не обнаруживаются, общее количество проектов и др.), хранятся 7 дней. Эти метрики отправляются непосредственно в Code Intelligence, с целью дальнейшего улучшения их проектов.

Еще одним известным проектом является Fuzzit [13], который на текущий момент является частью GitLab Security [14]. Ранее этот инструмент распространялся свободно, но сейчас входит только в GitLab Ultimate, исходный код проекта закрыт. Данный проект интегрируется непосредственно в CI-конвейеры GitLab (в том числе в закрытые репозитории) и автоматизируется с их помощью. Фаззинг можно запускать вручную (только одну обертку) или по фиксациям (количество запускаемых оберток зависит от описания конкретного конвейера разработчиком или специалистом по безопасности). Запуск вне GitLab или встраивание в другие платформы для разработки невозможно.

Регрессионное тестирование предыдущих срабатываний возможно только с использованием функционала libFuzzer, позволяющего сделать прогон всех сохраненных корпусов без непосредственного запуска фаззинга. Данная функция доступна только для него и не работает с AFL.

По умолчанию не поддерживается функция определения оберток, затрагивающих измененный участок. При настройке конвейеров можно указать вручную необходимые обертки, однако этот метод является не слишком надежным.

Данный инструмент поддерживает следующие языки программирования:

- C/C++,
- Go,
- Swift,
- Rust,
- Java,
- JavaScript,
- Python.

с помощью фаззеров libFuzzer, Javafuzz [16], jsfuzz [17], pythonfuzz [18], AFL, что в полной мере охватывает необходимый набор языков, использующихся при разработке системного ПО и компонентов операционных систем. Добавление других фаззеров или их модификаций невозможно.

Платформа предоставляет возможность интеграции фаззинга в собственные проекты, добавление оберток и интеграцию фаззинга в CI/CD. Предлагается достаточно простая процедура для добавления новых фаззинг-конвейеров и фаззинг-целей: разработчики могут самостоятельно интегрировать свои проекты, следуя документации и используя предоставляемые в открытом доступе шаблоны и конфигурационные файлы.

Длительность фаззинг-тестирования ограничена параметрами и есть возможность выбора из двух вариантов: короткий фаззинг на 10 минут (по умолчанию) и запуск на 60 минут, который рекомендуется использовать для ветвей, в которых на текущий момент ведется процесс разработки, и запросов на слияние. Настройка фаззинга достаточно гибкая и зависит от описания сборки в конвейерах. Также есть возможность передавать аргументы для фаззинга, доступна возможность добавлять собственные словари, настраивать переменные окружения. Параллельный фаззинг недоступен, как следствие, возможно только переиспользование корпуса, но не обмен между различными экземплярами фаззера. Длительность хранения артефактов настраивается вручную и можно задать любое значение.

Так как данная платформа встроена в GitLab и функционирует посредством конвейера непрерывной интеграции, то так как GitLab поддерживает масштабируемость и распределение нагрузки по агентам «GitLab Runner», это также распространяется и на агенты, отвечающие за фаззинг-тестирование.

Интеграция с другими платформами для хранения результатов или другими средствами анализа возможна посредством настройки конвейеров. Выгрузка результатов фаззинга доступна в различных форматах, в том числе JSON с информацией об ошибках. Информационная панель, встроенная в GitLab, отображает информацию обо всех обнаруженных ошибках, также каждая обнаруженная ошибка автоматически сопоставляется с CVE или CWE. Информация о покрытии не отображается, покрытие не собирается по умолчанию ни по отдельным оберткам, ни по всему проекту в целом. Обнаруженные ошибки не дедуплицируются, минимизация данных, вызывающих ошибку, и корпусов также не производится.

Еще одно проприетарное решение для автоматизации процессов анализа безопасности в рамках непрерывного цикла безопасной разработки – Mayhem [19]. Данный инструмент является инструментом с закрытым исходным кодом и предоставляет несколько вариантов запуска тестирования (в том числе возможен локальный запуск): с использованием графического веб-интерфейса, с помощью интерфейса командной строки, его также возможно интегрировать в CI-конвейеры используемой платформы для разработки. Этот инструмент объединяет в себе использование фаззинга с символьным исполнением.

Масштабирование происходит эффективно, включая работу с большим количеством узлов [21].

В данном инструменте реализовано регрессионное тестирование: все тестовые примеры, вызывающие ошибку, сохраняются, чтобы затем автоматически воспроизводить их на новых версиях ПО и сообщать разработчику или специалисту по безопасности об успешности или неуспешности устранения. Регрессионное тестирование также можно инициировать вручную через графический интерфейс или интерфейс командной строки.

Данный инструмент не производит отбора оберток, которые затрагивают изменения в коде. Эту функцию необходимо настраивать самостоятельно в собственных конвейерах или указывать, какие фаззинг-цели необходимо запускать и при каких обстоятельствах.

Mayhem поддерживает следующие языки программирования:

- C/C++,
- Go,
- Rust,
- Java,
- Python,
- Ada,

с помощью фаззеров libFuzzer, hongfuzz [20] Jazzer, Atheris, AFL, что в полной мере охватывает необходимый набор языков, использующихся при разработке системного ПО и компонентов операционных систем. Добавление других фаззеров или их модификаций невозможно.

Платформа предоставляет возможность интеграции фаззинга в собственные проекты, добавление оберток, запуск фаззинга и анализ обнаруженных ошибок. Предлагается достаточно простая процедура для добавления новых фаззинг-целей, однако в документации не отражена информация о добавлении инструмента в свои конвейеры, так что их подготовка и конфигурация остается за специалистами. Как следствие, автоматизированный запуск в конвейерах зависит от собственных настроек, инструмент, в отличие от, например, OSS-Fuzz и ClusterFuzz не предоставляет готовых шаблонов для настройки. В остальном для запуска фаззинга и добавления фаззинг-целей предоставляются шаблоны и конфигурационные файлы.

Для настройки фаззинга есть ряд параметров в конфигурационных файлах, можно настраивать длительность, способ передачи аргументов в обертку и непосредственно фаззеру, однако, нигде не указано, что можно выбирать санитайзеры отдельно под конкретную цель, это необходимо указывать самостоятельно, непосредственно при сборке и нельзя указать в конфигурациях, таким образом, для каждого санитайзера необходимо вручную готовить свою сборку. Также нет явной возможности использования только определенных санитайзеров, а также спецификации, какие санитайзеры применимы для конкретной фаззинг-цели. Более того, с целью оптимизации было бы эффективнее явно указывать, с каким фаззером (фаззерами), санитайзером (санитайзерами) и инструментацией запускается конкретная обертка, чтобы запускать фаззинг в автоматизированных конвейерах.

Mayhem предоставляет информацию о результатах тестирования и обо всех обнаруженных ошибках в графическом интерфейсе, есть возможность выгрузить все обертки и артефакты к ним, однако обнаруженные ошибки нельзя выгрузить в стандартных форматах с целью загрузки их в собственные базы и интеграции с другими инструментами. Также нет возможности импортировать в этот интерфейс дополнительную информацию об ошибках от других инструментов анализа. Взаимодействие с системами отслеживания задач также не реализовано. Покрытие по результатам собирается для каждой обертки в отдельности и не

консолидируется по всему проекту. Обнаруженные ошибки дедуплицируются, также минимизируется тестовый пример для воспроизведения обнаруженной ошибки. Ошибки сопоставляются с CWE. Минимизация корпусов отсутствует. Длительное хранение артефактов также доступно и настраивается отдельно.

Все рассмотренные решения предоставляют широкие возможности для автоматизации процессов анализа безопасности, включая интеграцию в существующие системы CI/CD. По результатам сравнения составлена сводная таблица (табл. 1), отражающая описанное выше.

При составлении таблицы использовалась следующая система оценок:

- «+» – требование удовлетворено полностью;
- «+/-» – требование удовлетворено частично;
- «-» – функционал полностью отсутствует или не соответствует требованию;
- «?» – наличие функционала не отражено в документации.

Табл. 1. Результат сравнения инструментов автоматизации фаззинг-тестирования
Table 1. Results of the comparison of fuzzing automation tools

Критерий	OSS-Fuzz	Cluster Fuzz		CIFuzz	FuzzIt	Mayhem
Регрессионное тестирование	-	-		+	+/-	+
Запуск соответствующих оберток, затрагивающих только изменения	+	+		-	-	-
Локальный запуск	-	+/-		+	-	+
Поддерживаемые ЯП	+	+		+/-	+	+
Удобство использования и простота настройки	+	+		+	+	+
Гибкая настройка параметров фаззинга	+/-	+/-		-	+	+/-
Возможность запуска вручную	-	+		+/-	+	+
Возможность запуска по триггерам в CI-конвейерах	+	+		+	+	-
Масштабируемость	+	+		?	+	+
Открытый исходный код	+	+		-	-	-
Добавление новых фаззеров	-	-		-	-	-
Интеграция с другими системами, экспорт результатов	+/-	+/-		-	+	-
Графический интерфейс	+	+		+	+/-	+
Сбор покрытия для одной обертки	+	+		+	-	+
Консолидация покрытия по всему проекту	+	+		-	-	-
Минимизация и дедупликация	+/-	+/-		-	-	+

обнаруженных ошибок						
Длительное хранение артефактов фаззинга	-	-		?	+	+

3. Предлагаемое решение

В этом разделе будет описана разрабатываемая платформа для автоматизации тестирования безопасности, нацеленная в первую очередь на процессы динамического анализа. Помимо этого, также описана платформа хранения и агрегации результатов анализаторов, производящая дедупликацию, приоритизацию и кластеризацию результатов статического анализа, в том числе с помощью алгоритмов машинного обучения. Предлагаемые клиент-серверные приложения интегрированы между собой, что позволяет облегчить процесс анализа безопасности исходного кода и оценки защищенности для таких больших программных продуктов, как операционная система.

В разрабатываемых платформах учтены недостатки существующих разработок и добавлены нововведения для упрощения процессов динамического анализа. Разработка отдельного сервиса не ограничивает добавление дополнительных функций, необходимых как для улучшения процессов анализа безопасности, так и для исследовательских задач. Помимо этого, возможность системы хранения формировать отчеты об обнаруженных ошибках позволяет повысить операционную эффективность и упростить по меньшей мере процессы подготовки сертификационных документов. Кроме того, при большом количестве используемых инструментов и подходов по анализу безопасности, платформа позволит агрегировать все результаты и упростить их анализ. Также, разработка собственного подхода открывает возможности для будущих расширений и настроек с учетом конкретных потребностей команд разработки и специалистов по анализу безопасности.

В рамках данной главы будут описаны обе платформы и их совместная работа, а также отражены плюсы от их интеграции и взаимодействия.

3.1 Архитектура платформы автоматизации фаззинг-тестирования

Ключевая цель разработки платформы автоматизации фаззинг-тестирования – это объединение циклов тестирования всех разрабатываемых компонентов операционной системы в единую систему тестирования. При этом важна адаптивность системы под специфику продуктов и их компонентов, а также возможность предоставления результатов тестирования различными способами, удобными как для разработчиков, так и для специалистов по безопасности, занимающихся как анализом обнаруженных уязвимостей, так и адаптацией и улучшением существующих подходов анализа безопасности. Помимо этого, важна возможность объединения нескольких инструментов динамического анализа не только для увеличения скорости, но и для обнаружения специфических ошибок, которые нельзя обнаружить с использованием стандартных инструментов. Когда разрабатываемым продуктом является операционная система с собственными средствами защиты, необходимо использовать различные инструменты, учитывающие специфику отдельных компонентов и их функциональные особенности. Здесь также применимо использование собственных отладочных аллокаторов, направленных на проверку корректности состояния средств защиты и настроенных в операционной системе политик безопасности во время фаззинг-тестирования, так как ряд компонентов, взаимодействие с ними и обращение к определенным модулям, может непосредственно влиять на состояние защищенности системы. Также с целью обнаружения ошибок на ранних этапах цикла непрерывной разработки необходимо внедрять фаззинг-тестирование в систему контроля версий, используемую в организации, что поможет как разработчикам, так и специалистам по безопасности быстро отследить возникновение новых ошибок или успешность устранения ранее обнаруженных.

Общая архитектура предлагаемой платформы изображена на рис.1.

Пользовательский интерфейс представляет собой адаптивное одностраничное приложение (SPA). Для разработки клиентской части был использован язык программирования JavaScript, фреймворки Vue.js [22] и Vuetify [23]. Vue.js позволяет ускорить разработку пользовательского интерфейса веб-приложения, благодаря возможности использования и переиспользования готовых компонент в качестве шаблонов. Библиотека vue-router [24] отвечает за организацию маршрутизации для сопоставления запросов к приложению с определенными компонентами интерфейса. Библиотека Axios [25] используется для формирования HTTP-запросов к API платформы. Фреймворк Pinia [26] используется в качестве хранилища состояния приложения для Vue.js.

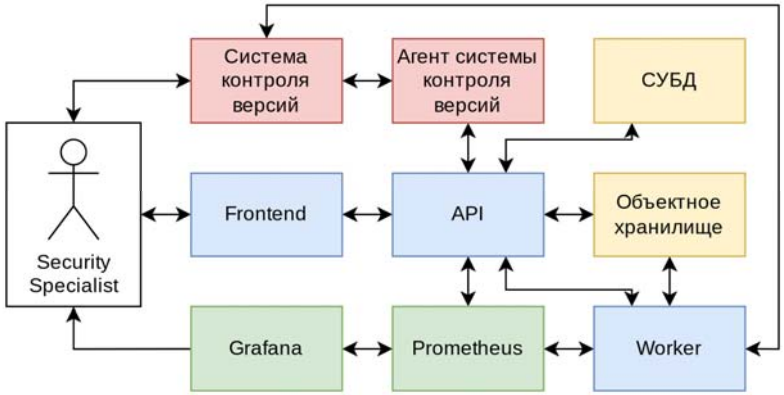

Рис.1. Общая архитектура разрабатываемой платформы автоматизации фаззинг-тестирования
Fig.1. General architecture of the developed fuzzing automation platform

Всю серверную составляющую приложения можно разбить на следующие сервисы: API, Worker, СУБД и объектное хранилище. Серверная часть (API) реализована на языке Python с использованием фреймворка Django REST Framework [27] и асинхронной очереди задач Celery [28]. Хранение данных осуществляется в PostgreSQL [29] и S3-совместимом объектном хранилище MinIO [30].

За задачу выполнения фаззинг-тестирования отвечает компонент Worker. Данный компонент запрашивает необходимые данные у серверной части через REST API.

Компоненты Prometheus и Grafana отвечают за сбор и визуализацию метрик состояния платформы.

На рис. 2 представлено подробное отображение компонентов платформы и их функционал.

Взаимодействие в рамках платформы в случае работы с CI осуществляется следующим образом:

1) Разработчик ПО вносит изменения в исходный код отслеживаемых проектов и ветвей в систему контроля версий.
2) По заданным требованиям производится проверка необходимости запуска Агента системы контроля версий. Если изменение воспринято системой как критичное, то Агент определяет набор фаззинг-оберток, покрывающих соответствующее изменение и запускает задачу на фаззинг.

3) Worker выгружает из системы контроля версии образы контейнеров, исходный код ПО, фаззинг-обертки, конфигурации и информацию о сборке, и запускает соответствующие контейнеры для проведения фаззинг-тестирования.
4) В рамках контейнеров производится сборка оберток в составе проекта. Артефакты предыдущего фаззинг-тестирования выгружаются из объектного хранилища для переиспользования. В зависимости от конфигураций сборки проекта (информация хранится в репозиториях в YAML-файлах) и конфигурации текущего запуска фаззинга, запускается несколько стендов с разными инструментами фаззинг-тестирования и разными санитайзерами, которые совершают обмен корпусами между собой. Результаты фаззинга отправляются на API в режиме реального времени и отображаются в графическом интерфейсе платформы. Перед началом фаззинга всегда проводится регрессионное тестирование предыдущих обнаруженных срабатываний, информация о результатах сохраняется. Помимо этого, есть возможность также проводить регрессионное тестирование отдельно от фаззинга с целью проверки устранения конкретных ошибок.

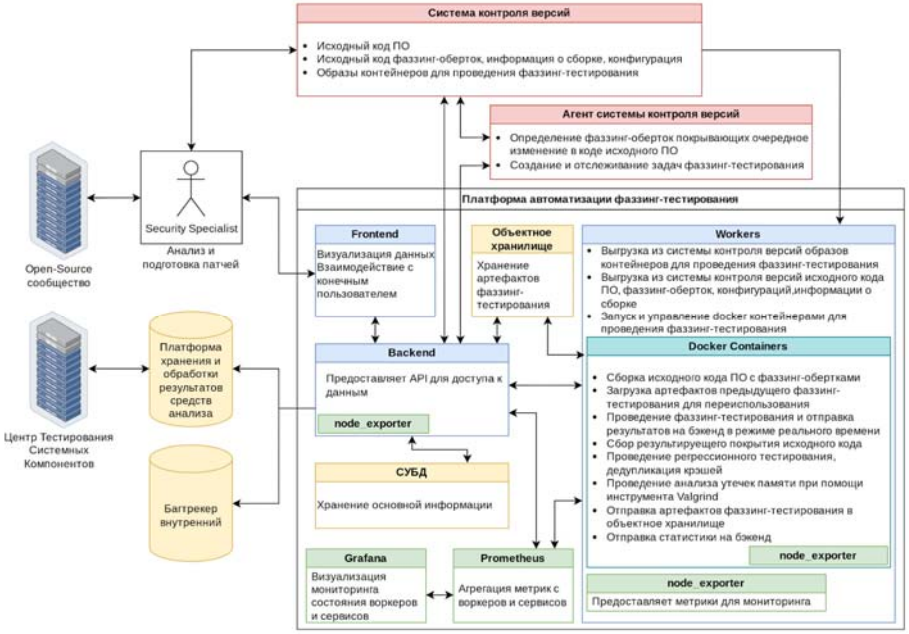

Рис.2. Описание компонентов платформы
Fig.2. Description of the platform's components

5) По результатам фаззинга собирается результирующее покрытие для каждой обертки в отдельности, а также консолидированное по всем запущенным на фаззинг оберткам.
6) Для каждой запущенной обертки дополнительно проводится анализ с помощью инструментов Valgrind [31], Dr.Memory [40]. Для тестовых примеров, вызывающих ошибки в тестируемой программе, также формируются выводы strace/ltrace.

7) По результатам фаззинга для всех обнаруженных падений производится минимизация входных данных и дедупликация с помощью CASR [32]. Содержание отчетов для дедуплицированных ошибок отображается в графическом интерфейсе.
8) Все артефакты фаззинга отправляются в объектное хранилище, результирующая статистика отправляется на API и отображается в графическом интерфейсе.

Помимо этого, в платформе также возможен ручной запуск задач с гибкой настройкой параметров фаззинга через графический интерфейс. Доступен выбор инструментов, санитайзеров, настройка переменных окружения и аргументов, передаваемых непосредственно при запуске, доступна загрузка собственных словарей, настройка времени фаззинга и условий его останова (например, достижение определенного количества запусков), выбор ветви с исходным кодом, когда необходимо протестировать конкретные изменения в версии, находящейся в активной разработке, выбор базового образа для контейнера. Возможен как точечный запуск задач для фаззинга с помощью одной обертки, так и массовый для одного или нескольких проектов. В случае запуска фаззинга через графический интерфейс, процесс будет происходить по алгоритму, описанному выше, за тем лишь исключением, что будет пропущена стадия проверки в системе контроля версий и запуска соответствующего агента.

Все полученные результаты (информация о покрытии, информация об обнаруженных ошибках и любая другая важная информация для воспроизведения, исправления и отладки обнаруженного падения) автоматически передается на платформу хранения и обработки результатов и во внутреннюю систему отслеживания задач. Информация о номере задачи в системе отслеживания сохраняется на платформе, ее статус автоматически проверяется, и в случае исправления проводится регрессионное тестирование соответствующей исправленной версии.

3.2 Функциональные возможности платформы автоматизации фаззинг-тестирования

Разработанная платформа реализует регрессионное тестирование по ранее обнаруженным ошибкам. Такой вид тестирования может быть запущен вручную через графический интерфейс специалистом по анализу безопасности или разработчиком, а также в случае, если соответствующая задача в системе отслеживания изменила статус. В таком случае будет запущен тот же алгоритм сборки обертки и исходного проекта, сбор данных об устраненной ошибке из объектного хранилища, в том числе входные данные, ее вызывающие. В случае, если ошибка успешно устранена, такая информация будет сохранена на платформе и передана в систему отслеживания для закрытия задачи.

Определение целевых оберток, затрагивающих только измененный в системе контроля версий код, осуществляется следующим образом:

1) При добавлении нового проекта, новой обертки для существующего проекта на платформу или при внесении изменений в существующие обертки, осуществляется запуск длительного процесса фаззинга (по умолчанию 8 часов или 4 часа, в течение которых не обнаруживаются новые пути). Это тестовый запуск для сбора информации о результативности фаззинга.
2) По результатам тестового запуска для каждой из оберток собирается покрытие кода по строкам и ветвям, после чего информация сохраняется в объектном хранилище для каждой обертки.
3) В дальнейшем при запуске новой задачи на фаззинг, проверяется, относится ли измененный код к покрытию, достигаемому той или иной оберткой.

При этом предложенный подход отличается от подхода, предлагаемого в OSS-Fuzz тем, что в OSS-Fuzz все обертки сначала собираются в составе проекта, а после этого происходит определение целевых оберток по достигаемому ими покрытию. В предложенном же подходе эти действия происходят в обратном порядке: проверка, а после сборка. Это связано с тем, что сборка всех оберток (для больших проектов их количество может превышать сотни, особенно в случае, если используется несколько технологий фаззинга и инструментов, требующих определенной специфики при подготовке оберток) может быть ресурсозатратной задачей, в связи с этим, необходимо собирать лишь те, что будут непосредственно участвовать в текущем процессе фаззинга. Помимо этого, в зависимости от указанных в конфигурации санитайзеров, каждая обертка собирается в нескольких экземплярах, при этом при фаззинге разные экземпляры фаззеров обмениваются накопленным корпусом между собой. Такое решение связано с тем, что некоторые санитайзеры являются конфликтующими, а также совместное использование может замедлять фаззинг.

На текущий момент на платформе используются следующие инструменты: AFL++, libFuzzer, KLEE [33], в проработке также находятся Crusher [34] и Sydr-fuzz. При необходимости доступно подключение дополнительных инструментов. Поддерживаются те языки программирования, фаззинг которых доступен текущему набору инструментов.

Разработанная платформа предоставляет возможность добавления новых проектов, а также дополнительных оберток для существующих посредством добавления их в основной репозиторий проекта. Предлагается достаточно простая процедура для добавления новых проектов и фаззинг-целей: разработчики могут самостоятельно интегрировать свои проекты, следуя документации и используя предоставляемые в открытом доступе шаблоны и конфигурационные файлы. При этом доступна возможность конфигурирования собственных проектов: фиксация доступных фаззеров (например, в случае специфических оберток, подготовленных специально для libFuzzer), санитайзеров (в случае, когда проект не удается корректно сконфигурировать с какими-то из них), способов инструментации и определенных флагов и конфигураций фаззера (например, когда необходимо выделить больше времени на таймауты в связи со скорость работы тестируемого ПО). Помимо этого, для локального тестирования, например, для отладки подготовленных оберток, предоставляются скрипты для запуска и подготовленные контейнеры, что позволяет имитировать среду, используемую на платформе, и запускать тестирование вручную на своей машине.

Как уже было описано выше, разработанная платформа предлагает несколько способов запуска фаззинга: запуск после внесении изменений в системе контроля версий, принудительный запуск агента в системе контроля версий, а также запуск из графического интерфейса. При этом из графического интерфейса возможно запустить как одну обертку, так и все обертки в проекте или все проекты, например, в случае выхода обновления.

Основная инфраструктура была спроектирована таким образом, чтобы предоставлять возможность автоматического масштабирования объема вычислительных мощностей в точном соответствии с текущими потребностями, которые могут возникать в процессе выполнения различных задач фаззинга. Это важно, так как позволяет обеспечивать достаточный уровень производительности, необходимый для успешного выполнения даже самых сложных и ресурсозатратных задач. Более того, платформа нацелена на максимально эффективное использование доступных вычислительных ресурсов. Это достигается за счет умного динамического распределения задач между всеми доступными вычислительными узлами. Такой подход позволяет гарантировать, что каждый процесс фаззинга получает доступ к необходимому объему ресурсов, исключая при этом излишнее расходование выделенных мощностей. Кроме того, система использует собственные алгоритмы для автоматического определения приоритетов задач. Помимо этого, автоматически выбирается соответствующий Worker, в зависимости от версии пакета. Это связано с тем, что многие тестируемые средства защиты зависят от модулей подсистемы безопасности в ядре ОС, и их функционал и поведение могут меняться от версии к версии. Дополнительно, подсистема

динамического распределения задач автоматически выделяет больше времени и ресурсов тем фаззинг-оберткам, которые затрагивают критичные участки и модули, недавно модицифированные или те, в которых было внесено большее количество изменений. При этом для таких задач выделяются дополнительные ресурсы. Этот процесс значительно увеличивает шансы на обнаружение и устранение потенциальных уязвимостей на ранних этапах разработки, тем самым способствуя созданию более надежных и безопасных программных продуктов.

Разработанная платформа взаимодействует с внутренней системой отслеживания задач, отправляя туда информацию об ошибках и получая информацию об их устранении. Помимо этого, выгрузка отчетов об ошибках и архивов с покрытием возможна из графического интерфейса. Вся информация об обнаруженных ошибках в формате SARIF также отправляется на платформу хранения и обработки данных, где анализируется и сопоставляется с результатами других инструментов.

3.3. Платформа хранения и обработки результатов анализа

Разработанная платформа представляет собой инструмент для специалистов анализа безопасности и разработчиков, позволяющий:

- производить разметку ошибок статического анализа компонентов как в рамках релизного цикла, так и в рамках процессов разработки;
- автоматически приоритизировать обнаруженные ошибки;
- сопоставлять результаты от различных средств анализа с целью развития инструментальных средств и подходов к анализу кода;
- хранить и обрабатывать статистику об обнаруженных ошибках;
- распределять задачи между специалистами, отправлять конкретные задачи по разметке на проверку;
- производить перекрестную разметку с целью верификации полученных ранее результатов;
- автоматизировано формировать отчеты об обнаруженных ошибках в рамках пакета или продукта;
- предоставлять централизованный доступ к результатам анализа, их выгрузку и загрузку.

Общая архитектура разработанной платформы отражена на рис. 3 и включает в себя следующие основные модули:

- брокер сообщений RabbitMQ [35], получающий результаты из различных источников с помощью соответствующих «производителей» (producer) и обрабатывающий с помощью «потребителей» (consumer);
- база данных PostgreSQL для хранения результатов анализа;
- графический интерфейс для отображения результатов и интерфейс разметки результатов;
- модуль машинного обучения для разметки, кластеризации и приоритизации;
- система контроля версий в качестве хранилища кода.

Взаимодействие в рамках платформы в случае добавления новых результатов анализа осуществляется следующим образом:

1) Ошибки дедуплицируются с уже загруженными в базу данных сущностями.

2) В случае, если такая ошибка уже была загружена и подтверждена ранее, но какие-либо поля отличаются, сущность в базе обновляется с новыми данными. Информация о добавлении ошибки отправляется на платформу фаззинга.
3) В случае, если ошибка не соответствует ранее обнаруженным, на платформе регистрируется новое событие, и ошибка загружается в базу данных. Информация о добавлении ошибки отправляется на платформу фаззинга.
4) С помощью алгоритмов машинного обучения выдается предсказание по критичности (в дополнение к критичности, выданной анализатором), ошибка сопоставляется с CWE и размечается с помощью модели.
5) Задача присваивается специалисту по приоритетам (в зависимости от пакета, критичности и других критериев).
6) В случае, если ошибка подтверждена специалистом (статус Confirmed), автоматически заводится задача в системе отслеживания (для ряда компонентов). К ошибке привязывается уникальный идентификатор задачи в системе отслеживания.
7) После устранения (задача перешла в соответствующий статус), информация обновляется на платформе.

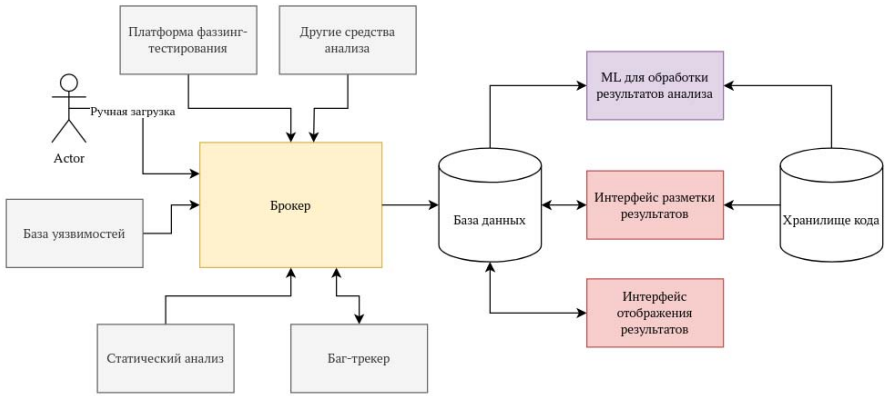

Рис.3. Общая архитектура разрабатываемой платформы хранения и обработки результатов
Fig.3. General architecture of the developed platform for storing and processing results

Ранее в статье [36] был предложен подход по подтверждению результатов статического анализа ядра ОС с помощью направленного фаззинга с использованием инструмента syzkaller [37]. Аналогичная стратегия применима и для других компонентов, входящих в пространство пользователя. Более того, так как разработанная платформа хранения результатов позволяет также хранить информацию и о покрытии кода, нет необходимости использовать сторонние средства хранения и корреляции результатов для этих целей. Помимо этого, алгоритмы машинного обучения, встроенные в платформу, используют покрытие фаззинга и обнаруженные с его помощью ошибки, как один из критериев для принятия решения по обнаруженным ошибкам.

Для подтверждения обнаруженных ошибок используется обученная на собственных наборах данных инновационная модель идентификации уязвимостей DefectHunter [38], которая базируется на механизме «Conformer». Эта модель объединяет в себе преимущества механизма внимания (self-attention) и сверточных нейронных сетей, тем самым обеспечивая

захват как локальных, так и глобальных признаков, которые являются ключевыми для точного и эффективного определения потенциальных уязвимостей. Использование автоматизированных методов наряду с ручной разметкой позволяет не только расставить приоритеты для задач, но и упростить работу специалистов, позволяя им акцентировать внимание в первую очередь не только на наиболее критичных срабатываниях, но и на тех, что подтверждены моделью.

4. Заключение и дальнейшие перспективы исследования

Совокупность описанных подходов, применяемых в рамках процессов анализа безопасности компонентов операционной системы, включающих в себя как системное ПО, в том числе реализующее средства защиты информации, так и прикладное ПО пространства пользователя, позволяет минимизировать участие специалистов и аналитиков в настройке средств анализа и запуске стендов фаззинг-тестирования, а также сократить время, затрачиваемое на первичный анализ полученных результатов и дедупликацию срабатываний, полученных в результате тестирования. Более того, предложенный подход автоматизации процессов фаззинг-тестирования в конечном итоге исключает для специалиста необходимость запускать стенды фаззинга самостоятельно при внесении изменений разработчиком в систему контроля версий. Помимо этого, реализованы подходы, направленные на эффективное использование вычислительных ресурсов, позволяющие снизить затраты при фаззинге крупных проектов. Реализованы подходы по масштабированию процесса фаззинга, динамическое распределение нагрузки и мониторинг состояния, что позволяет эффективно распределять ресурсы между задачами и отслеживать их загруженность. Разработанная платформа автоматизации позволяет существенно сократить время обнаружения ошибок в новых фиксациях, что является существенным критерием для оценки результативности. В результате внедрения всех описанных подходов, время от изменения кода до обнаружения ошибки сократилось в десятки раз по той причине, что специалистам по безопасности больше нет необходимости отслеживать изменения и тестировать выпущенные версии компонентов, вместо этого платформа уведомляет о новых обнаруженных ошибках в фиксациях автоматически.

Предложенная платформа хранения и обработки результатов предоставляет исчерпывающую статистику по всем результатам анализа, а благодаря синхронизации платформ и их совместной работе, позволяет коррелировать и пересчитывать полученные ранее результаты на новых данных. Более того, внедрение алгоритмов машинного обучения позволило оптимизировать работу специалистов по разметке, так как с внедрением описанных методов все задачи приоритизируются не только на основе данных от анализаторов, но и с использованием всей остальной имеющейся информации.

Таким образом, предложенные подходы позволяют не только минимизировать количество ошибок в коде системного и прикладного ПО на протяжении всего жизненного цикла разработки, выявляя возможные уязвимости на ранних этапах, но и предоставляет исчерпывающую статистику по результатам, обнаруженным за все время анализа. Такая статистика позволяет анализировать состояние компонентов в контексте безопасности и принимать решения о переходе на новые версии, сравнивая обнаруженные ошибки. Помимо этого, платформа предоставляет возможность коррелировать средства анализа между собой, что также ускоряет работу специалистов, занимающихся анализом полученных в ходе тестирования результатов. Платформы интегрированы со внутренней системой отслеживания задач, что позволяет в автоматическом режиме передавать информацию о результатах анализа разработчикам и также ускоряет процессы анализа безопасности.

В качестве направления исследований можно выделить следующие направления:

- Интеграция Fuzz Introspector;

- Добавление на платформу среды для воспроизведения и детального анализа обнаруженных падений и среды для разработки фаззинг-целей (взаимодействие с изолированном окружением через ssh-подключение или напрямую через графический интерфейс платформы);
- Внедрение LLM для генерации фаззинг-оберток (по аналогии с CI Spark и OSS-Fuzz-Gen);
- Внедрение других моделей машинного обучения для обнаружения и подтверждения ошибок в исходном коде, например, VulBERTA [41];
- Добавление функционала сравнения результатов статического анализа в разных версиях ПО в интерфейс разметки.

Список литературы / References

[1]. Проект OSS-Fuzz / OSS-Fuzz project. Available at: https://github.com/google/oss-fuzz/, accessed 12.04.2024.
[2]. Проект OSS-Sydr-Fuzz / OSS-Sydr-Fuzz project. Available at: https://github.com/ispras/oss-sydr-fuzz, accessed 12.04.2024.
[3]. Инструментальное средство фаззинг-тестирования AFLplusplus / AFLplusplus. Available at: https://github.com/AFLplusplus/AFLplusplus, accessed 12.04.2024.
[4]. Инструментальное средство фаззинг-тестирования libFuzzer / libFuzzer. Available at: https:// llvm.org/docs/LibFuzzer.html, accessed 12.04.2024.
[5]. A. Vishnyakov et al., “Sydr-Fuzz: Continuous Hybrid Fuzzing and Dynamic Analysis for Security Development Lifecycle,” in 2022 Ivannikov ISPRAS Open Conference (ISPRAS), 2022, pp. 111–123. doi: 10.1109/ISPRAS57371.2022.10076861.
[6]. Расширяемая инфраструктура для фаззинга ClusterFuzz / ClusterFuzz. Available at: https://google.github.io/clusterfuzz, accessed 12.04.2024.
[7]. Инструментальное средство фаззинг-тестирования CIFuzz / CIFuzz. Available at: https://www.code-intelligence.com/product-ci-fuzz, accessed 12.04.2024.
[8]. T. Klooster, F. Turkmen, G. Broenink, R. ten Hove, and M. Böhme, Effectiveness and Scalability of Fuzzing Techniques in CI/CD Pipelines. 2022.
[9]. Проект Fuzz Introspector / Fuzz Introspector. Available at: https://github.com/ossf/fuzz-introspector, accessed 12.04.2024.
[10]. Инструментальное средство фаззинг-тестирования Jazzer / Jazzer. Available at: https://github.com/CodeIntelligenceTesting/jazzer, accessed 12.04.2024.
[11]. Инструментальное средство фаззинг-тестирования Jazzer.js / Jazzer.js. Available at: https://github.com/CodeIntelligenceTesting/jazzer.js, accessed 12.04.2024.
[12]. Инструментальное средство CI Spark / CI Spark. Available at: https://www.code-intelligence.com/product-ci-spark, accessed 12.04.2024.
[13]. Платформа управления и автоматизиации фаззинга CI Sense / CI Sense. Available at: https://www.code-intelligence.com/product-ci-sense, accessed 12.04.2024.
[14]. Инструментальное средство FuzzIt / FuzzIt. Available at: https://github.com/fuzzitdev, accessed 12.04.2024.
[15]. Платформа Gitlab Security / GitLab Security. Available at: https://about.gitlab.com/solutions/security-compliance/, accessed 12.04.2024.
[16]. Инструмент фаззинг-тестирования Javafuzz / Javafuzz. Available at: https://gitlab.com/gitlab-org/security-products/analyzers/fuzzers/javafuzz, accessed 12.04.2024.
[17]. Инструмент фаззинг-тестирования jsfuzz / jsfuzz. Available at: https://gitlab.com/gitlab-org/security-products/analyzers/fuzzers/jsfuzz, accessed 12.04.2024.

[18]. Инструмент фаззинг-тестирования pythonfuzz / pythonfuzz. Available at: https://gitlab.com/gitlab-org/security-products/analyzers/fuzzers/pythonfuzz, accessed 12.04.2024.
[19]. Платформа для автоматизации процессов анализа безопасности Mayhem / Mayhem. Available at: https://www.mayhem.security/, accessed 12.04.2024.
[20]. Инструмент фаззинг-тестирования hongfuzz / hongfuzz. Available at: https://github.com/google/honggfuzz, accessed 12.04.2024.
[21]. Mayhem, the Machine That Finds Software Vulnerabilities, Then Patches Them. Available at: https://spectrum.ieee.org/mayhem-the-machine-that-finds-software-vulnerabilities-then-patches-them, accessed 12.04.2024.
[22]. Фреймворк Vue.js / Vue.js. Available at: https://vuejs.org/, accessed 12.04.2024.
[23]. Фреймворк Vuetify / Vuetify. Available at: https://vuetifyjs.com/en/, accessed 12.04.2024.
[24]. Библиотека vue-router / vue-router. Available at: https://router.vuejs.org/, accessed 12.04.2024.
[25]. Библиотека Anxious / Anxious. Available at: https://axios-http.com/, accessed 12.04.2024.
[26]. Фреймворк Pinia / Pinia. Available at: https://pinia.vuejs.org/, accessed 12.04.2024.
[27]. Django REST Framework. Available at: https://www.django-rest-framework.org/, accessed 12.04.2024.
[28]. Инструмент для асинхронной обработки задач Celery / Celery. Available at: https://github.com/celery/celery/, accessed 12.04.2024.
[29]. Система управления базами данных PostgreSQL / PostgreSQL. Available at: https://www.postgresql.org/, accessed 12.04.2024.
[30]. Объектное хранилище MinIO / MinIO. Available at: https://min.io/, accessed 12.04.2024.
[31]. Инструментальное средство Valgrind / Valgrind. Available at: https://valgrind.org/, accessed 12.04.2024.
[32]. CASR: инструмент формирования отчетов об ошибках / CASR. Available at: https://www.ispras.ru/technologies/casr/, accessed 12.04.2024.
[33]. Инструмент фаззинг-тестирования Crusher / Crusher. Available at: https://www.ispras.ru/technologies/crusher/, accessed 12.04.2024.
[34]. Комплекс гибридного фаззинга и динамического анализа Sydr / Sydr. Available at: https://www.ispras.ru/technologies/sydr/, accessed 12.04.2024.
[35]. Брокер сообщений RabbitMQ / RabbitMQ. Available at: https://www.rabbitmq.com/, accessed 12.04.2024.
[36]. Егорова В.В., Панов А.С., Тележников В.Ю., Девянин П.Н. Подходы, направленные на повышение эффективности фаззинг-тестирования компонентов защищенной ОС. Труды Института системного программирования РАН, том 34, вып. 4, 2022г., стр. 21-34 / Egorova V.V., Panov A.S., Telezhnikov V.Y., Devyanin P.N. Approaches for improving the efficiency of protected OS components fuzzing. 2022;34(4):21-34. https://doi.org/10.15514/ISPRAS-2022-34(4)-2.
[37]. Инструмент фаззинг-тестирования syzkaller / syzkaller. Available at: https://github.com/google/syzkaller, accessed 12.04.2024.
[38]. J. Wang, Z. Huang, H. Liu, N. Yang, and Y. Xiao, DefectHunter: A Novel LLM-Driven Boosted-Conformer-based Code Vulnerability Detection Mechanism. 2023.
[39]. Фреймворк OSS-Fuzz-Gen / OSS-Fuzz_gen. Available at: https://github.com/google/oss-fuzz-gen, accessed 12.04.2024.
[40]. Инструмент Dr.Memory / Dr.Memory. Available at: https://drmemory.org/, accessed 12.04.2024.
[41]. VulBERTa / VulBERTa. Available at: https://github.com/ICL-ml4csec/VulBERTa, accessed 12.04.2024.

Информация об авторах / Information about authors

Егор Петрович СУРАЕВ – студент Института кибербезопасности и цифровых технологий РТУ МИРЭА, специалист по безопасности в отделе динамического анализа. Область научных интересов: фаззинг, динамический анализ, машинное обучение, поиск уязвимостей в программном обеспечении.

Egor Petrovich SURAEV – student at the Institute of Cybersecurity and Digital Technologies in RTU MIREA, Information Security Specialist of Dynamic Analysis Unit. Field of Interest: fuzzing, dynamic analysis, machine learning, vulnerability research.

Виктория Вячеславовна ЕГОРОВА – заместитель директора департамента анализа безопасности, аспирант ВМК МГУ. Область научных интересов: анализ программ, динамический анализ, фаззинг.

Victoriia Vyacheslavovna EGOROVA – Deputy Head of Security Analysis Department, postgraduate student at the Faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. Field of Interest: program analysis, dynamic analysis, fuzzing.

Алексей Сергеевич ПАНОВ – руководитель направления динамического анализа, аспирант ИСП РАН. Область научных интересов: поиск уязвимостей в ПО, анализ защищенности ИС.

Alexey Sergeevich PANOV – Head of Dynamic Analysis Unit, postgraduate student at the ISP RAS. Field of Interest: vulnerability research, penetration testing.