DOI: 10.15514/ISPRAS-2022-34(4)-2

Подходы, направленные на повышение эффективности фаззинг-тестирования компонентов защищенной ОС

В.В. Егорова, ORCID: 0000-0003-1286-3631 <vegorova@astralinux.ru>
А.С. Панов, ORCID: 0000-0002-0046-0766 <apanov@astralinux.ru>
В.Ю. Тележников, ORCID: 0000-0002-6192-2856 <vtelezhnikov@astralinux.ru>
П.Н. Девянин, ORCID: 0000-0003-2561-794X <pdevyanin@astralinux.ru>

ООО «РусБИТех-Астра»,
117105, г. Москва, Варшавское ш., д. 26, стр.11

Аннотация. Фаззинг-тестирование в рамках цикла непрерывной разработки является необходимым инструментом, направленным в первую очередь на обеспечение доверия к разрабатываемому ПО. При этом при наличии значительных объемов кодовой базы фаззинг-тестирование становится ресурсозатратной задачей и именно поэтому важным направлением исследований является повышение эффективности фаззинг-тестирования с целью наиболее быстрого достижения интересующих участков кода без снижения качественных показателей. В рамках данной статьи рассмотрены подходы, направленные на повышение эффективности фаззинг-тестирования как для ядерных модулей, так и для ПО, входящего в пространство пользователя. С другой стороны, на отмеченных объемах программного кода инструментальные средства статического анализа выдают колоссальное количество предупреждений о возможных ошибках, при этом основные ресурсы для такого тестирования требуются не для получения результатов работы анализаторов, а для их полноценной аналитической обработки. В связи с этим, в статье значительное внимание уделяется подходу корреляции результатов статического и динамического анализа с помощью разработанного авторами инструментального средства, позволяющего в том числе реализовать направленное фаззинг-тестирование с целью подтверждения срабатываний статических анализаторов, что значительно повышает эффективность проведения тестирования компонентов защищенной ОС Astra Linux.

Ключевые слова: динамический анализ; направленное фаззинг-тестирование; фаззинг; операционная система; Astra Linux

Для цитирования: Егорова В.В., Панов А.С., Тележников В.Ю., Девянин П.Н. Подходы, направленные на повышение эффективности фаззинг-тестирования компонентов защищенной ОС. Труды ИСП РАН, том 34, вып. 4, 2022 г., стр. 21-34. 10.15514/ISPRAS-2022-34(4)-2

Approaches for improving the efficiency of protected OS components fuzzing

V.V. Egorova, ORCID: 0000-0003-1286-3631 <vegorova@astralinux.ru>
A.S. Panov, ORCID: 0000-0002-0046-0766 <apanov@astralinux.ru>
V.Y. Telezhnikov, *ORCID: 0000-0002-6192-2856 <vtelezhnikov@astralinux.ru>*
P.N. Devyanin, ORCID: 0000-0003-2561-794X <pdevyanin@astralinux.ru>

RusBITech-Astra
26, Varshavskoe, Moscow, 117105, Russia

Abstract. Fuzzing as a part of the continuous integration is a necessary tool, aimed primarily at the providing confidence in the software being developed. At the same time, in the presence of significant amounts of the source code, fuzzing becomes a resource-intensive task. That's why increasing the efficiency of fuzzing to reach needed code sections more quickly without reducing quality becomes an important line of research. The article deals with approaches to improve the efficiency of fuzzing both for kernel and for user-space software. On the other hand, on these amounts of program code, static code analysis produces a huge number of warnings about possible errors, and the main resources within this type of analysis are required not to obtain to result, but for analytical processing. In this regard, in the article considerable attention is paid to the approach of correlating the results of static and dynamic code analysis using the developed tool, which also allows to implement directed fuzzing in order to confirm the warnings of static analyzer, which significantly increases the efficiency of testing components of the protected OS Astra Linux.

Keywords: dynamic analysis; directed fuzzing; fuzzing; operating system; Astra Linux

For citation: Egorova V.V., Panov A.S., Telezhnikov V.Y., Devyanin P.N. Approaches for improving the efficiency of protected OS components fuzzing. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 4, 2022. pp. 21-34 (in Russian). DOI: 10.15514/ISPRAS-2022-34(4)-2

1. Введение

Согласно утвержденной ФСТЭК России «Методике выявления уязвимостей и недекларированных возможностей в программном обеспечении» [1] одним из этапов разработки безопасного ПО для обеспечения соответствия ее требованиям, начиная с минимального уровня доверия, является выполнение динамического анализа программного кода объекта оценки, в том числе с применением фаззинг-тестирования. Данный метод динамического анализа является наиболее распространенным и эффективным, благодаря чему появилось множество инструментальных средств, реализующих различные технологии тестирования, а также множество инструментов, направленных на повышение эффективности фаззинга.

При проведении тестирования операционной системы Astra Linux [2] (далее – ОС Astra Linux), разрабатываемой в ГК «Астра» и сертифицированной по наивысшему, первому уровню доверия, необходимо применять наиболее передовые методы и программные средства фаззинга. При этом важно выбирать технологии и средства анализа в соответствии с исследуемым ПО, опираясь на его функциональные особенности и специфику для достижения наибольшей эффективности фаззинг-тестирования.

Исходя из этого, объектами исследования при проведении авторами фаззинг-тестирования в первую очередь являются собственные средства защиты информации (далее – СЗИ), интерфейсы которых являются важнейшей составляющей поверхности атаки ОС Astra Linux. При этом эти СЗИ функционируют как в пользовательском пространстве, так и на уровне ядра ОС, где имеется подсистема безопасности PARSEC, реализованная в модулях ядра ОС linux-astra-modules на основе мандатной сущностно-ролевой ДП-модели управления доступом и информационными потоками (МРОСЛ ДП-модели) [3]. Помимо тестирования СЗИ в ГК «Астра» также проводится фаззинг-тестирование ПО с открытым исходным кодом,

которое играет важную роль в реализации функций защиты информации. Среди такого ПО, например, подключаемые модули аутентификации PAM, представляющие собой одну из частей стандартного механизма аутентификации UNIX-подобных систем, которые также дополнены собственными модулями принятия решений. Помимо PAM-модулей также проводится тестирование ядра ОС и его модулей безопасности, в том числе с использованием различных санитайзеров (KASAN, KCSAN и др.), что в сочетании с разработанными авторами подходами к фаззинг-тестированию позволяет обнаруживать уникальные ошибки, не отраженные в результатах тестирования ядер ОС семейства Linux в рамках других проектов – общедоступной базы syzbot от компании Google [4] и Технологического центра исследования безопасности ядра Linux [5]. С целью сравнения обнаруженных на собственных стендах ошибок с ошибками, обнаруженными в рамках syzbot используется разработанное в ГК «Астра» инструментальное средство syzStats, позволяющее с помощью графического интерфейса отсеивать уникальные «падения» ядра ОС, ранее не отраженные в syzbot.

В ходе фаззинг-тестирования авторами также используются различные технологии расширения полученного покрытия исходного кода, позволяющие одновременно сократить время выполнения фаззинг-тестирования интересующих модулей. Также апробируются технологии по корреляции результатов различных инструментальных средств, предназначенных для анализа программного кода, например, сопоставление результатов динамического и статического анализа с использованием корпусов и наборов системных вызовов, сгенерированных с помощью syzkaller [6], что в результате позволяет с помощью направленного фаззинг-тестирования достоверно подтверждать ошибки, обнаруженные статическими анализаторами.

В данной работе изложены применяемые в ГК «Астра» подходы по повышению эффективности фаззинг-тестирования с использованием как собственных инструментов, так и инструментов с открытым исходным кодом. При этом важно отметить применяемые авторами критерии, которые позволяют охарактеризовать процедуру фаззинг-тестирования как эффективную:

- количество обнаруженных ошибок на стенде фаззинг-тестирования с учетом времени работы стенда, а также количество подтвержденных срабатываний, полученных в ходе статического анализа;
- количество обнаруженных уникальных ошибок в ядре Linux, не выявленных ранее другими исследователями;
- процент покрытия кода, полученный в результате работы стенда или консолидированное покрытие на нескольких стендах, а также время его достижения;
- степень автоматизации создания, запуска и обработки результатов, полученных на стендах фаззинг-тестирования.

Статья организована следующим образом. В разд. 2 содержится обзор применяемых в ходе исследований безопасности кода инструментальных средств, как для ядерного, так и для пользовательского пространства, а также используемых при этом подходов, позволяющих не только увеличить уровень покрытия кода, но и сократить время выполнения фаззинг-тестирования интересующих частей кода. В разд. 3 излагаются подходы к фаззингу ядра ОС Astra Linux и ее подсистемы безопасности, реализованной в модулях linux-astra-modules. В разд. 4 описан новый подход к корреляции результатов статического и динамического анализа, реализованный авторами в инструменте syzCore. Разд. 5 посвящен применяемым подходам в рамках фаззинг-тестирования ПО, входящего в пространство пользователя, в том числе с использованием алгоритмов машинного обучения. Заключение завершает статью, в нем приводятся возможные направления дальнейших исследований.

2. Используемые инструментальные средства

Для реализации фаззинг-тестирования компонентов ядра ОС, как правило, могут использоваться общедоступные средства динамического анализа, среди которых наибольшее распространение получило специально предназначенное для ядер ОС программное средство syzkaller [6] – проект с открытым исходным кодом, с применением которого было выявлено множество уязвимостей в различных версиях ядра Linux. Он является наиболее перспективным и динамично развивающимся средством фаззинг-тестирования ядра ОС, драйверов и модулей. Данный инструмент имеет встроенную поддержку ядра Linux и является наиболее функциональным средством для тестирования ядра ОС, драйверов и модулей, в том числе и выгружаемых модулей linux-astra-modules, специально включаемых в состав ядра при фаззинг-тестировании, и в рамках которых функционируют механизмы защиты подсистемы безопасности PARSEC.

С целью тестирования отдельных драйверов, библиотек и приложений пользовательского пространства, в том числе входящих в подсистему безопасности PARSEC и являющихся важной составляющей поверхности атаки, в ГК «Астра» применяются положительно зарекомендовавшие себя средства динамического анализа, среди которых различные модификации фаззера American Fuzzy Lop (AFL) [7], libFuzzer [8] – инструмент, предназначенный для фаззинга библиотек, комплекс динамического анализа программ Crusher [9], разработанный ИСП РАН. С применением данных инструментальных средств авторами также создана единая автоматизированная система фаззинга приложений пользовательского пространства, которая непрерывно улучшается и адаптируется под новые условия, инструменты и подходы. Это делается с целью ускорения развертывания стендов, автоматизации фаззинг-тестирования и сбора покрытия в рамках цикла непрерывной разработки и регрессионного тестирования.

Рис. 1. Общая схема работы системы для сбора покрытия
Fig. 1. General scheme of coverage collecting system

Для сбора и анализа покрытия в ходе работы фаззеров применяется разработанная авторами система (рис.1), основанная на свободно распространяемой утилите для построения покрытия исходного кода Gcov [10] и ее расширении lcov [11]. Эта система минимизирует исходный корпус с целью снижения времени, требуемого на воспроизведение всех тестовых примеров, создаваемых инструментами фаззинг-тестирования, а после использует их для генерации результатов покрытия кода. Эти результаты сохраняются для каждого файла отдельно и в дальнейшем сливаются в один файл, демонстрирующий итоговое покрытие тестами кода всей анализируемой библиотеки или подсистемы. Покрытие кода интерпретируется от одного тестового значения к другому, чтобы определить, какие новые ветви, функции и строки покрываются фаззером с новым тестовым примером. Получение полных и исчерпывающих данных о покрытии кода в сочетании с ручным анализом помогает максимизировать эффективность фаззинга.

Таким образом, применяются различные инструменты и их комбинации, совместно с другими апробированными технологиями безопасной разработки и обеспечения доверия к ОС Astra Linux, а также проводится сбор и анализ выявленных ошибок и результаты их устранения. Весь перечисленный комплекс инструментальных средств, вместе с инструментами, предназначенными для других видов анализа и верификации объединяется в единый комплекс – стенд доверия (рис. 2), развернутый в рамках практики непрерывной интеграции в инструментальной среде GitLab [12] с целью автоматизации процессов сборки, тестирования и верификации программного кода в соответствии с требованиями нормативных документов [1, 12].

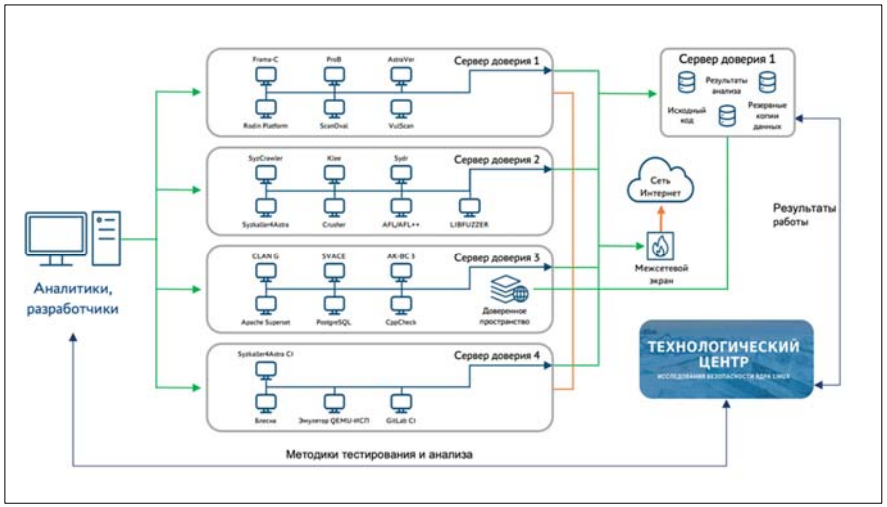

Рис.2. Масштабируемая структура стенда доверия
Fig. 2. Scalable structure of the Trust Bench

3. Фаззинг ядра ОС

Инструмент syzkaller обладает достаточно широкими возможностями для динамического анализа ядра ОС, драйверов и модулей и реализует эффективные алгоритмы фаззинг-тестирования: генерационные, для составления начальных тестовых данных на основе предоставленных ему описаний системных вызовов, мутационные для генерации новых тестовых значений с целью достижения большего покрытия, генетические для отбора наиболее подходящих экземпляров набора сгенерированных корпусов входных данных – расширяющих покрытие или чаще приводящих к падениям.

С учетом специфики ОС на основе фаззера syzkaller в ГК «Астра» разработан комплекс непрерывного фаззинг-тестирования syzkaller4astra, обеспечивающий встраивание в тестируемое ядро необходимых модулей подсистемы безопасности PARSEC, и их настройку, а также включающий в себя все созданные в компании инструментальные средства, направленные на повышение эффективности фаззинг-тестирования. На рис. 3 изображена обобщенная схема функционирования стендов динамического анализа кода syzkaller4astra.

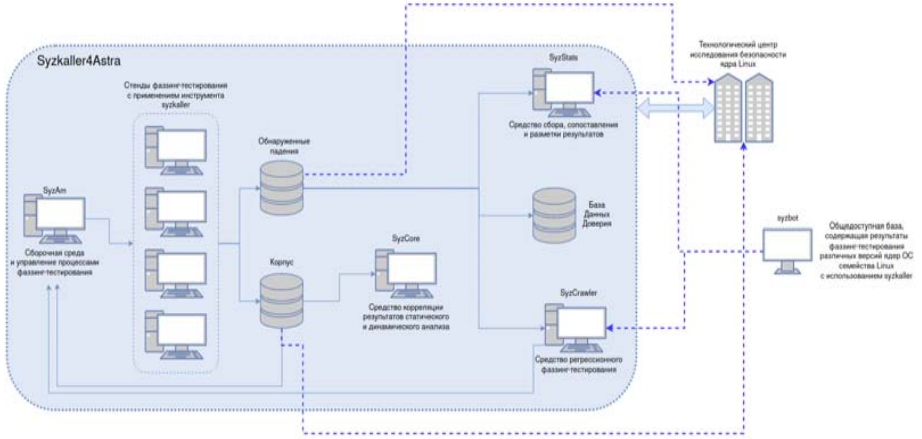

Рис.3. Динамический анализ кода ОС Astra Linux с использованием стенда syzkaller4astra
Fig. 3. Dynamic analysis of the OS Astra Linux code using the syzkaller4astra bench

При использовании данного инструмента авторами применяются следующие подходы, позволяющие увеличить уровень покрытия кода, одновременно сократив время выполнения фаззинг-тестирования в первую очередь для подсистемы безопасности PARSEC:

- ограничение тестируемых системных вызовов ядра ОС с целью ускорения процесса фаззинг-тестирования и увеличения покрытия программного кода linux-astra-modules, для чего было разработано специальное автоматизированное средство сбора и анализа статистики базового покрытия в общем и по каждому системному вызову в частности, а также для отслеживания динамики изменения покрытия по каждому системному вызову с течением времени;
- уточнение существующих и описание новых прототипов системных вызовов, что позволяет увеличить покрытие исходного кода при фаззинг-тестировании и выполнить направленный фаззинг для сокращения времени тестирования модулей подсистемы безопасности PARSEC, интерфейсы которой являются важнейшей составляющей поверхности атаки к ОС Astra Linux;
- с целью использования результатов фаззинг-тестирования ядер ОС семейства Linux, полученных сторонними организациями, авторами был написан инструмент syzcrawler, который осуществляет сбор и обработку данных из общедоступной базы syzbot [4], содержащей результаты фаззинг-тестирования различных версий ядер ОС семейства Linux с применением инструментального средства syzkaller, и данных, полученных в ходе взаимодействия с Технологическим центром исследования безопасности ядра Linux [5], после чего эти данные тестируются на ядре ОС Astra Linux, а корпусы, содержащие последовательность системных вызовов и позволяющие увеличить покрыте кода, добавляются к собственным корпусам и используются на стендах syzkaller4astra;
- регрессионное фаззинг-тестирование, также реализованное в рамках проекта syzcrawler, позволяющее обеспечить автоматизированное тестирование обнаруженных ранее падений на стендах фаззинга на доработанных версиях ядра ОС с целью подтверждения их устранения;
- обнаружение аналогичных уже найденным ошибок по шаблонам, что позволяет находить типовые ошибки, не дожидаясь их выявления с помощью инструментов динамического

анализа;

- динамический анализ модулей ядра ОС при имитации исключительных ситуаций, например, нехватки системных ресурсов, с помощью инструментального средства KEDR, разработанного ИСП РАН [14];
- корреляция результатов статического и динамического анализа с помощью разработанной авторами автоматизированной системы syzcore, что позволяет подтверждать наличие ошибки, обнаруженной статическим анализатором;
- сбор покрытия модулей ядра одновременно с фаззинг-тестированием модулей защиты, входящих в пространство пользователя, что позволяет получать более полную информацию о покрытии подсистемы безопасности PARSEC в пространстве ядра во время выполнения программ в пространстве пользователя, и, как следствие, повышать эффективность фаззинг-тестирования.

С целью упрощения анализа обнаруженных на стендах syzkaller4astra ошибок авторами разработан инструмент syzStats, позволяющий автоматически сопоставлять найденные на стендах syzkaller4astra ошибки с ошибками, содержащимися в общедоступной базе syzbot, что дает возможность быстрее обнаруживать уникальные ошибки, ранее не выявленные другими исследователями.

Для применения инструментального средства syzkaller в рамках цикла непрерывной разработки авторами развивается автоматизированная система для фаззинг-тестирования ядер ОС – syzAm, которая позволяет упростить и ускорить развертывание стендов, предоставляет балансировку нагрузки серверов, консолидированную информацию о производительности стендов и покрытии кода тестами, автоматически генерирует файлы-конфигураций. Помимо этого, в данную автоматизированную систему включены инструменты тестирования syzcrawler и сопоставления ошибок syzStats.

Для администрирования данных процессов разработана панель управления стендами фаззинга ядра ОС, которая позволяет не только централизовано хранить и просматривать информацию со всех стендов тестирования, отслеживать покрытие кода, анализировать обнаруженные ошибки и их уникальность относительно общедоступной базы данных syzbot, но и контролировать их работу, запуская регрессионное тестирование, а также создавая дополнительные и управляя уже существующими стендами фаззинга.

Рис. 4. Обобщенная статистика стендов фаззинга, построенная с помощью Grafana и Prometheus
Fig. 4. General fuzzing benches stats built using Grafana and Prometheus

Рис. 5. Статистика стендов фаззинга, построенная с помощью Grafana и Prometheus
Fig. 5. Fuzzing benches stats built using Grafana and Prometheus

Для сбора статистических данных по всем стендам фаззинг-тестирования в этой автоматизированной системе используется набор инструментов для сбора метрик Prometheus [15] с открытым исходным кодом, позволяющий собирать и хранить метрики в виде временных рядов. На рис. 4 и 5 приведены примеры построения статистики с помощью

Prometheus, интегрированной с Grafana [16]. Применение монитора состояния фаззинг-тестирования позволяет своевременно выполнять балансировку нагрузки, задавать и отслеживать множественные критерии определения эффективности работы стендов, осуществлять своевременную выгрузку и реагирование на полученные результаты тестирования.

В результате применения описанных подходов удалось повысить эффективность фаззинг-тестирования. В частности, покрытие по базовым блокам для ядерных модулей подсистемы безопасности PARSEC превышает 60% (более 95% для монитора управления доступом), а скорость его достижения в результате применения описанных подходов сократилась вдвое, что, несомненно, влечет за собой значительное сокращение накладных расходов ресурсов серверов стенда доверия. При этом также важны и качественные показатели: в ходе применения предложенных подходов обнаруживаются и устраняются ошибки в PARSEC, а также уникальные ошибки для других компонентов ядра ОС Linux, которые не были отражены в syzbot, а благодаря применению автоматизированной системы регрессионного тестирования и системе обнаружения аналогичных уже существующим падениям по шаблонам удалось также сократить время на устранение и подтверждение устранения обнаруженных ранее ошибок.

4. Корреляция результатов статического и динамического анализа

При наличии значительных объемов кодовой базы ручная разметка кода и ручной анализ потенциальных ошибок, обнаруженных статическим анализатором, является затруднительным в первую очередь из-за большого количества срабатываний (предупреждений), в том числе имеющих высокий уровень критичности. На отмеченных объемах программного кода применяемые инструментальные средства выдают тысячи, а иногда, десятки тысяч предупреждений о возможных ошибках, большая часть из которых, как правило, оказываются ложными или не учитывающими специфику анализируемого кода, что наиболее актуально для модулей или драйверов ядра ОС [17]. В связи с этим, одним из важнейших направлений научно-исследовательской деятельности в ГК «Астра» является решение задачи разработки инструментальных средств, автоматизирующих обработку этих предупреждений. В рамках этих исследований авторами была выдвинута идея, что в качестве метода подтверждения обнаруженных статическим анализатором ошибок возможно использовать направленное фаззинг-тестирование.

Эта идея основывается на следующем предположении: если в ходе работы syzkaller было получено покрытие потенциально опасной функции (т.е. функции, в которой с помощью статического анализа были обнаружены ошибки высокого уровня критичности), то возможно ограничить тестируемые системные вызовы до тех, что позволили получить покрытие этой функции, а корпус до тех тестовых примеров, которые позволили достигнуть интересующих участков кода. При этом участки кода, являющиеся в данном случае «интересными», могут быть вручную выбраны и расширены исследователем. В случае, если syzkaller не удалось получить покрытие на тех участках, где статический анализатор обнаружил ошибку, подбор направленных системных вызовов остается на усмотрение исследователя.

При этом важно отметить, что предложенный подход не позволяет достоверно подтвердить отсутствие ошибки, обнаруженной статическим анализатором. Несмотря на это, среди большого количества срабатываний статического анализатора для тех ошибок, для которых был организован направленный фаззинг, но которые не были подтверждены на стендах фаззинг-тестирования, снижается уровень приоритета. Это означает, что при ручной разметке аналитик в первую очередь обратит внимание на те предупреждения, для которых либо еще не было предпринято попытки подтверждения с помощью направленного фаззинг-тестирования, либо на те, которые уже были обнаружены (подтверждены) на стенде фаззинг-

тестирования. Такой подход позволяет эффективно распределять ресурсы для анализа и устранения обнаруженных ошибок.

Для внедрения описанного подхода в масштабируемую автоматизированную систему верификации и анализа кода, поддерживаемую в ГК «Астра» на базе стенда доверия, авторами было написано инструментальное средство syzcore, общая схема функционирования которого отражена на рис. 6.

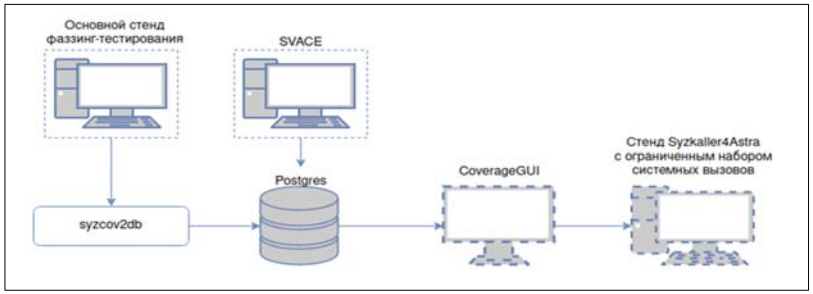

Рис. 6. Общая схема функционирования инструмента syzcore
Fig. 6. General scheme of the sycore tool

На текущем этапе этим инструментальным средством осуществляется сопоставление покрытия, полученного в результате фаззинг-тестирования ядра ОС с помощью syzkaller, и результатов работы статического анализатора SVACE [18], разработанного ИСП РАН.

Рис. 7. Пример срабатывания SVACE, обнаруженного в рамках работы Технологического центра исследования безопасности ядра Linux
Fig. 7. An example of error, detected using SVACE as a part of the work of The Linux Kernel Security Technology Research Center

Для сбора покрытия ядра syzkaller использует kcov, а само покрытие представляется с помощью добавления на этапе инструментации вызова __sanitizer_cov_trace_pc в каждый базовый блок, сгенерированный во время процесса сборки. В ходе апробации различных подходов к выгрузке покрытия и информации со стенда syzkaller4astra был выбран способ получения «сырых» адресов, так как в условиях большого объема кода такой способ является наиболее эффективным. Далее с помощью addr2line и объектного файла ядра vmlinux полученные данные возможно преобразовать в строки исходных файлов. SVACE, в свою

очередь, предоставляет информацию о принадлежности ошибки к конкретной функции в файле.

С помощью графической визуализации (рис. 7), можно проанализировать совместное покрытие двух инструментов, а также тестовые примеры и системные вызовы, зашедшие в ходе фаззинг-тестирования в то или иное ветвление. В результате, после экспертного анализа пересечений обнаруженных SVACE ошибок и полученных данных о покрытии syzkaller создаются новые стенды фаззинг-тестирования с усеченными корпусами и системными вызовами.

Предупреждение, сформированное статическим анализатором SVACE и отраженное на рис. 7 (строка 319), было подтверждено с помощью направленного фаззинга инструментом syzkaller (рис. 8). С целью подтверждения системные вызовы были ограничены только вызовами, вошедшими в функцию с предупреждением, также был собран усеченный корпус, состоящий только из релевантных тестовых примеров.

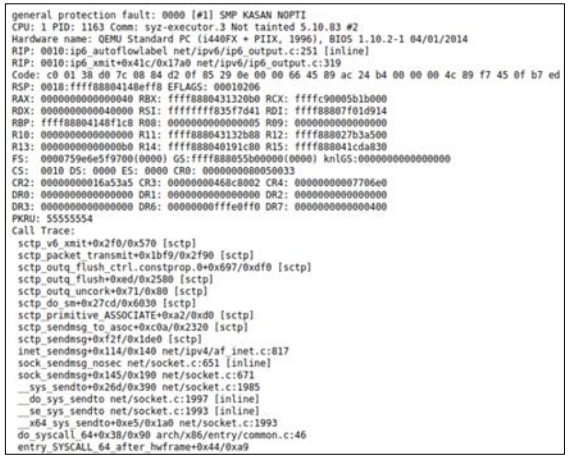

Рис. 8. Подтверждение ошибки, обнаруженной SVACE, с помощью syzkaller
Fig. 8. Confirming an error detected by SVACE with sykaller

Таким образом, на текущем этапе разработан инструмент, позволяющий сопоставлять результаты, полученные в ходе статического анализа, с покрытием, полученным со стендов фаззинг тестирования и системными вызовами, позволяющими попасть в тот или иной участок кода. При этом благодаря автоматизированным технологиям по корреляции результатов статического и динамического анализа успешно подтверждаются срабатывания статических анализаторов, и на очередной итерации исследования кода модулей безопасности ядра удалось подтвердить 5 новых срабатываний. При этом благодаря применяемым подходам направленного фаззинг-тестирования, количество воспроизводимых ошибок неуклонно увеличивается.

5. Фаззинг программного обеспечения пользовательского пространства

В ходе динамического анализа приложений пользовательского пространства, в том числе утилит и библиотек, входящих в подсистему безопасности PARSEC, являющуюся важнейшей составляющей поверхности атаки из пространства пользователя, для них были авторами написаны специальные «программы-обертки», которые получают на вход данные, сгенерированные фаззером, и передают их в целевую программу или конкретную функцию. При этом важным фактором для обертки является возможность принимать и обрабатывать

любые искаженные значения, создавать и изменять используемые в ходе работы программ файлы. Также для достижения эффективности фаззинг-тестирования важно, чтобы обертка имела узкую цель, по этой причине одной библиотеке часто соответствует несколько оберток, каждая из которых отвечает за анализ своего модуля. Помимо индивидуальных оберток для каждой программы с помощью системы автоматической генерации словарей создаются свои словари, при этом в словарь включаются флаги программ, длинные текстовые и числовые значения, и другие константы, сравнение с которыми содержится в исходном коде программы и подбор которых в ходе фаззинг-тестирования займет длительное время. При этом для сложных программ, работающих, например, с анализом содержимого объемных файлов, составленные словари авторами просматриваются вручную. Это связано с большим количеством сравнений в исходном коде программы, так как включение в словарь задействованных при этом данных в конечном итоге снизят эффективность от использования словаря.

С целью повышения эффективности фаззинг-тестирования приложений пользовательского пространства применяется подход, основанный на имитации исключительных ситуаций, направленный в первую очередь на то, чтобы покрыть те пути исполнения, которые возникают в таких ситуациях, например, при нехватке системных ресурсов, когда запрошенная программой память не выделилась из-за ее недостатка или какой-либо ошибки. Данный подход позволяет не только обнаружить ошибки, которые могут возникнуть в исключительной ситуации, но и расширить покрытие по ветвям, обрабатывающим данную исключительную ситуацию. Также такой подход используется для фаззинг-тестирования утилит и библиотек, входящих в подсистему безопасности PARSEC, работающих с системными файлами, и позволяет обработать случаи, когда файл был намеренно изменен или удален. В результате, применение данного подхода позволило увеличить покрытие утилит и библиотек PARSEC, входящих в пространство пользователя, в среднем на 10%.

Существует ряд готовых наборов инструментов, созданных с целью отслеживания и автоматизации процессов сборки и анализа кода с применением фаззинг-тестирования, среди них, например, набор утилит для аудита безопасности приложений BugBane [19]. Однако с учетом особенностей сборки и функционирования собственных СЗИ, а также необходимости создания и внедрения систем отслеживания состояния серверов, функционирующих в рамках стенда доверия, балансировки их нагрузки и потребности в удаленном администрировании всех стендов фаззинг-тестирования, возникает необходимость создания собственной автоматизированной системы, удовлетворяющей всем требованиям, предъявляемым к подобной системе в организации. Таким образом, для обеспечения применимости перечисленных инструментов для фаззинг-тестирования приложений пользовательского пространства в рамках цикла непрерывной разработки безопасного программного обеспечения в ГК «Астра» реализован проект по автоматизации процессов фаззинг-тестирования приложений пользовательского пространства, что обеспечивает целевую сборку анализируемого ПО сразу после внесения изменений разработчиками, генерацию словарей, распределение стендов тестирования по серверам. Аналогично с автоматизированной системой фаззинга ядра ОС, для фаззинга приложений пользовательского пространства разработана отдельная панель для управления стендами фаззинга, позволяющая отслеживать нагрузку на сервера, хранить и просматривать информацию со всех стендов, контролировать их работу, создавать новые стенды фаззинга и управлять уже существующими.

Процент покрытия, полученный в результате применения описанных подходов составляет свыше 85% по строкам для утилит и библиотек пользовательского пространства, входящих в подсистему безопасности PARSEC. Полученные результаты позволяют охарактеризовать процедуру фаззинг-тестирования как эффективную.

Кроме того, существует ряд исследований [20-22], подтверждающих эффективность применения алгоритмов машинного обучения в рамках фаззинг-тестирования, а также представляют ряд задач фаззинг-тестирования, которые могут решаться с использованием алгоритмов машинного обучения среди них [23]: генерация исходного тестового примера, создание тестового набора и его фильтрация, выбор эффективных мутаций и др. В конечном итоге решение этих задач непосредственно повлияет на покрытие кода и позволит в целом повысить результативность фаззинг-тестирования. Качество исходного тестового примера непосредственно влияет на результаты фаззинг-тестирования, а использование алгоритмов машинного обучения позволяет исследовать общие закономерности в тестируемых файлах и выделить те, которые приводят к большему покрытию исходного кода и увеличению количества обнаруженных ошибок [24]. Правильная генерация тестового корпуса напрямую влияет на результативность фаззинга и иногда только от него зависит, будет ли обнаружена та или иная ошибка. Фильтрация, в свою очередь, позволяет выделить среди всего (иногда достаточно объемного) корпуса минимальный набор тестовых примеров, позволяющий с наибольшей вероятностью обнаружить новые пути исполнения программы, что в конечном итоге не только расширит покрытие, но и может повысить вероятность обнаружение ранее не обнаруженной ошибки в коде исследуемой программы [25, 26]. Выбор эффективных мутаций в фаззинге также является важной задачей, так как влияние мутаторов в AFL не одинаково и их распределение напрямую зависит от тестируемой программы [27].

В рамках исследований, проводимых в ГК «Астра», проходят апробацию подходы по применению машинного обучения для создания и фильтрации тестового набора, а также для генерации исходного тестового примера с целью повышения результативности фаззинг-тестирования.

6. Заключение

Совокупность описанных подходов и реализующих их инструментальных средств, применяемых авторами как к ядру ОС и встроенным в него модулям подсистемы безопасности PARSEC, так и к приложениям пользовательского пространства, позволяет минимизировать участие аналитика в настройке и запуске стендов фаззинг-тестирования, а также сократить время, затрачиваемое на анализ обнаруженных срабатываний анализаторов. Предварительные результаты апробации говорят о корреляции между собой не менее 30% результатов, полученных в результате применения двух методов анализа кода (статического и динамического). Разработанное в ГК «Астра» инструментальное средство корреляции позволяет не только эффективно расставлять приоритеты при ручной разметке обнаруженных срабатываний, но и в конечном итоге повысить результативность проведения тестирования компонентов защищенной ОС Astra Linux. Помимо этого, важным критерием эффективности также является время выявления ошибок, которое в результате применения всех описанных подходов сократилось втрое по сравнению с классическим методом фаззинг-тестирования, как для пользовательского пространства, по имеющимся ранее оберткам, так и для модулей ядра ОС, благодаря направленному фаззинг-тестированию, позволяющему реализовывать целевое воспроизведение ошибок. В итоге за счет применения перечисленных подходов общее покрытие возросло на 15%, а время его достижение сократилось вдвое. А, значит, совокупность всех применяемых подходов обеспечивает повышение эффективности тестирования и большую уверенность в качестве и достоверности полученных результатов. Таким образом, предложенные подходы к фаззинг-тестированию позволяют обеспечивать минимизацию ошибок в коде СЗИ на протяжении всего жизненного цикла разработки, устраняя большинство программных ошибок на его ранних этапах, что создает условия для разработки безопасного системного ПО на базе ядра Linux и обеспечивает высокий уровень доверия непосредственно к ОС Astra Linux.

В качестве направлений дальнейших исследований наиболее приоритетными являются:

- оценка результативности подхода корреляции результатов статических и динамических анализаторов для анализа ПО, входящего в пользовательское пространство, в том числе СЗИ, реализуемых подсистемой безопасности PARSEC;
- создание инструмента для автоматизированного уточнения системных вызовов, что позволит быстрее расширять покрытие с целью достижения интересующих участков кода;
- построение графа вызовов ядерных функций, позволяющих достигать интересующих частей кода и сопоставлять их с системными вызовами.

Список литературы / References

[1] Информационное сообщение ФСТЭК России от 10.02.2021 № 240/24/ / Informational message of FSTEK Russia of 10th February 2021 #240/24/647. Available at: https://fstec.ru/normotvorcheskaya/informatsionnye-i-analiticheskie-materialy/2171-informatsionnoe-soobshchenie-fstek-rossii-ot-10-fevralya-2021-g-n-240-24-647, accessed 12.09.2022 (in Russian).

[2] Девянин П.Н., Тележников В.Ю., Третьяков С.В. Основы безопасности операционной системы Astra Linux Special Edition. Управление доступом. Учебное пособие. М., Горячая линия – Телеком, 2022 г., 148 стр. / Devyanin P.N., Telezhnikov V.Y., Tret'yakov S.V. Astra Linux Special Edition security basics. Access control. Hotline-Telecom, 2022, 148 p. (in Russian).

[3] Девянин П.Н. Модели безопасности компьютерных систем. Управление доступом и информационными потоками. Учебное пособие для вузов. М., Горячая линия – Телеком, 2020 г., 352 стр. / Devyanin P.N. Security models of computer systems. Control for access and information flows. Hotline-Telecom, 2020, 352 p. (in Russian).

[4] Проект syzbot / Syzbot project. Available at: https://syzkaller.appspot.com, accessed 12.09.2022.

[5] Технологический центр исследования безопасности ядра Linux / The Linux Kernel Security Technology Research Center. Available at: https://portal.linuxtesting.ru, accessed 12.09.2022 (in Russian).

[6] Syzkaller / Syzkaller. Available at: https://github.com/google/syzkaller, accessed 12.09.2022.

[7] Инструментальное средство фаззинг-тестирования American Fuzzy Lop / American Fuzzy Lop. Available at: https://github.com/google/AFL, accessed 12.09.2022.

[8] Инструментальное средство фаззинг-тестирования libFuzzer / libFuzzer. Available at: https://llvm.org/docs/LibFuzzer.html, accessed 12.09.2022.

[9] Комплекс динамического анализа программ Crusher / ISP Crusher: a dynamic analysis toolset. Available at: https://www.ispras.ru/en/technologies/crusher/, accessed 12.09.2022 (in Russian).

[10] Инструментальное средство сбора и подсчета покрытия Gcov / Gcov: source code coverage analysis and statement-by-statement profiling tool. Available at: https://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Gcov.html, accessed 12.09.2022.

[11] Инструментальное средство визуализации покрытия Lcov / Lcov: graphical front-end for Gcov. Available at: https://github.com/linux-test-project/lcov, accessed 12.09.2022.

[12] GitLab. Веб-инструмент жизненного цикла DevOps / GitLab. A full DevOps tool. Available at: http://gitlab.com, accessed 12.09.2022.

[13] ГОСТ Р 56939-2016 «Защита информации. Разработка безопасного программного обеспечения. Общие требования» / GOST R 56939-2016 «Information protection. Secure Software Development. General requirements». Federal Agency for Technical Regulation and Metrology, 2016 (in Russian).

[14] KEDR. Расширяемая система для динамического анализа модулей ядра Linux / KEDR. An extensible system for dynamic analysis and verification Linux kernel modules. Available at: https://github.com/euspectre/kedr, accessed 12.09.2022.

[15] Набор инструментов для мониторинга Prometheus / Prometheus. An open source monitoring system. Available at: https://prometheus.io, accessed 12.09.2022.

[16] Платформа для визуализации данных Grafana / Grafana. Multi-platform open-source analytics and interactive visualization web application. Available at https://grafana.com, accessed 12.09.2022.

[17] Девянин П.Н., Хорошилов А.В., Тележников В.Ю. Формирование методологии разработки безопасного системного программного обеспечения на примере операционных систем. Труды ИСП РАН, том 33, вып. 5, 2021 г., стр. 25-40 / Devyanin P.N., Telezhnikov V.Y., Khoroshilov V.V. Building a methodology for secure system software development on the example of operating systems. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 5, 2021, pp. 25-40 (in Russian). DOI: 10.15514/ISPRAS–2021–33(5)–2.

[18] Статический анализатор Svace / SVACE static analyzer. Available at: http://www.ispras.ru/technologies/svace, accessed 12.09.2022 (in Russian).

[19] Набор утилит для аудита безопасности приложений BugBane / BugBane. A set of utilites for auditing application security. Available at: https://github.com/gardatech/bugbane, accessed 12.09.2022 (in Russian).

[20] Cheng L., Zhang Y. et al. Optimizing seed inputs in fuzzing with machine learning. In Proc. of the 41st International Conference on Software Engineering: Companion Proceedings, 2019, pp. 244-245.

[21] Cummins C., Petoumenos P. et al. Compiler fuzzing through deep learning. In Proc. of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2018, pp. 95-105.

[22] Godefroid P., Peleg H., Singh R.. Learn&Fuzz: Machine learning for input fuzzing. In Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, pp. 50-59.

[23] Wang Y., Jia P. et al. A systematic review of fuzzing based on machine learning techniques. PLoS ONE, vol. 15, issue 8, 2020, article no. e0237749, 37 p.

[24] Lyu C., Ji S. et al. SmartSeed: Smart Seed Generation for Efficient Fuzzing. arXiv preprint arXiv:1807.02606, 2018, 17 p.

[25] Gong W, Zhang G, Zhou X. Learn to Accelerate Identifying New Test Cases in Fuzzing. Lecture Notes in Computer Science, vol. 10656, 2017, pp. 298-307.

[26] Rajpal M, Blum W, Singh R. Not all bytes are equal: Neural byte sieve for fuzzing. arXiv preprint arXiv: 1711.04596, 2017, pp. 1-10.

[27] Lyu C., Ji S. et al. MOPT: Optimized Mutation Scheduling for Fuzzers. In Proc. of the 28th USENIX Security Symposium (USENIX Security 19), 2019, pp. 1949–1966.

Информация об авторах / Information about authors

Виктория Вячеславовна ЕГОРОВА – старший научный сотрудник. Область интересов: анализ программ, динамический анализ, фаззинг.

Victoriia Vyacheslavovna EGOROVA – senior researcher. Field of Interest: program analysis, dynamic analysis, fuzzing.

Алексей Сергеевич ПАНОВ – научный сотрудник. Область интересов: поиск уязвимостей в ПО, анализ защищенности ИС.

Alexey Sergeevich PANOV – researcher. Field of Interest: Vulnerability research, penetration testing.

Владимир Юрьевич ТЕЛЕЖНИКОВ – кандидат технических наук, начальник отдела научных исследований. Область интересов: формальные модели безопасности компьютерных систем, методы статического и динамического анализа программного кода, операционная система Linux.

Vladimir Yurevich TELEZHNIKOV, Ph.D in Technical Science, Head of Research Department. Field of Interest: formal security models of computer systems, methods of static and dynamic analysis of program code, Linux operating system.

Петр Николаевич ДЕВЯНИН – член-корреспондент Академии криптографии России, доктор технических наук, профессор, научный руководитель ООО «РусБИТех-Астра» (ГК «Астра»). Область интересов: теория информационной безопасности, формальные модели безопасности компьютерных систем.

Petr Nikolaevich DEVYANIN – Doctor of Technical Sciences, Corresponding Member of Russian Academy of Cryptography, Professor, Scientific Director in RusBITech-Astra (Astra Linux). Field of Interest: information security theory, formal security models of computer systems.