BIS JOURNAL

ИНФОРМАЦИОННАЯ БЕЗОПАСНОСТЬ БИЗНЕСА

№3 (58) 2025

Артём ЗУБКОВ
Медиа Группа «Авангард»
Технологии доверенного ИИ
→ 52

Дмитрий ПОНОМАРЁВ
ООО НТЦ «Фобос-НТ»
Многоликий динамический анализ → 76

Осенний марафон-2025

Дмитрий ЕВДОКИМОВ
Luntry
Всё для SOC в контейнерных средах и Kubernetes → 9

Георгий КОРАБЛЕВ
АМИКОН
От узкого специалиста до массового потребителя → 27

Юрий ШАБАЛИН
AppSec.Sting
Цифровой рубль в телефоне → 94

МНОГОЛИКИЙ ДИНАМИЧЕСКИЙ АНАЛИЗ

Дмитрий ПОНОМАРЕВ
заместитель генерального директора – директор департамента внедрения и развития практик РБПО ООО НТЦ «Фобос-НТ», сотрудник ИСП РАН, преподаватель МГТУ им. Н. Э. Баумана, модератор испытаний статических анализаторов исходного кода

Второй блок статей из серии «Три кита РБПО» посвящен технологиям и инструментам динамического анализа программного обеспечения. На первый взгляд интуитивно понятное определение «динамический анализ» на практике оказывается одним из самых «крепких орешков» при выборе подходов и инструментов динамических исследований программного обеспечения. Определение их применимости и достаточности, в условиях ограниченности ресурсов (и кадров), с учетом необходимости выполнения регуляторных требований, является типовой задачей, регулярно решаемой как разработчиками ПО, так и специалистами в области ИБ и РБПО, в том числе сотрудниками испытательных лабораторий. В статье приведен обобщенный взгляд на существующие технологии динамического анализа (далее – ДА), требования и особенности их применения. Цель статьи – подготовить читателя к последующему знакомству с конкретными инструментами динамического анализа, помочь в определении их полезности и применимости.

Раздумывая несколько месяцев над наполнением статьи, я в очередной раз столкнулся с обозначенной в аннотации проблемой. Казалось бы тривиальный в понимании термин «динамический анализ», или, в соответствии с ГОСТ Р 56939–2024[1] «динамический анализ кода программы», определяемый как *«вид работ по инструментальному исследованию программы, основанный на анализе кода программы в режиме непосредственного исполнения (функционирования) кода»*, на самом деле не дает ответа на большую часть практических вопросов, возникающих у исследователя, которому поставлена задача по проведению ДА. Пытаться ответить на них все единолично в рамках одной небольшой статьи, ещё и в формате стройной, формально-выверенной картины – задача заведомо нерешаемая. Поэтому решено действовать другим – уже проверенным в предшествующих материалах[2] – способом, а именно перечислить типовые вопросы, с которыми уважаемые разработчики периодически «влетают» в эксперта испытательной лаборатории, а также некоторые апробированные временем и практикой тезисы ответов.

1. ДИНАМИЧЕСКИЙ АНАЛИЗ ЭТО ЖЕ DAST?

Мой «любимый» тезис, регулярно всплывающий при общении с представителями разработчиков и исследователей безопасности кода. Сильно упрощая, для большого числа инженеров-безопасников ДА примерно тождественен «походить по API с помощью условного OWASP Zap»[3]. Действительно, являясь одним из полезных и распространенных видов ДА, комплекс подходов и инструментов анализа, в обиходе известный как DAST, сконцентрирован в первую и основную очередь на анализе API (в первую очередь в отношении web-фреймворков). Данный комплекс, как правило, ориентирован на поиск известных уязвимостей – как составляющих API компонентов, так и безопасности их конфигураций – либо, на выявление «нехороших» API (Shadow, Orphan и Zombie[4]) и их параметров. Практики и инструменты DAST в наибольшей степени применимы при работе команд пентестеров (RedTeam), а также при создании и постоянном контроле защищенности микросервисных (зачастую разрабатываемых в парадигме API-First) приложений, по своей природе предполагающих частые внесения изменений. Тем не менее вышеуказанные практики состав-

[1] https://clck.ru/3Ncr6J

[2] РБПО-2024. Итоги и перспективы // BIS Journal. – 2025. – № 1 (56). – С. 23–25.

[3] Видеозапись моего выступления на Kaspersky Certification Day: https://certificationday.kaspersky.com/ (08:08:00)

[4] https://habr.com/ru/companies/webmonitorx/articles/804489/

ляют только подмножество существующих и требуемых в ряде случаев практик ДА.

2. ЧТО ТРЕБУЕТ И ЧТО НЕ ТРЕБУЕТ ФСТЭК РОССИИ, КОГДА УПОМИНАЕТ ДИНАМИЧЕСКИЙ АНАЛИЗ?

Дискутируя с коллегами, мы, как правило, делим РБПО-практики на «вширь» и «вглубь»[5]. «Вширь» определяется ГОСТ Р 56939–2024, в котором вводятся 25 процессов, составляющих комплексный процесс РБПО, при этом сами описания процессов достаточно верхнеуровые и зачастую оставляют на усмотрение разработчика выбор инструментов и детали (в т.ч. численные критерии) реализации процессов. «Вглубь» — а именно более конкретные требования к технологическим возможностям инструментов и численные критерии — определяется Методикой выявления уязвимостей и недекларированных возможностей в ПО ФСТЭК России[6], доступной для использования в работе лицензиатам ФСТЭК России, а также частично приказом № 76 ФСТЭК России «Требования, устанавливающие уровни доверия...».

В область требуемых практик ДА с точки зрения Методики (для испытаний по 4 — самому типовому — уровню контроля) попадают как минимум:

- функциональное тестирование (иначе: «пользовательское» тестирование функций безопасности);
- модульное тестирование (основным доп. требованием которого является запуск тестов с комплектами датчиков срабатывания ошибок в тех случаях, когда это применимо);
- генетическое фаззинг-тестирование (о нем ниже);
- комплексный сбор информации об объекте в динамике оценки, совмещенный с тестированием на проникновение (на пальцах этот процесс можно охарактеризовать как совмещение практик DAST и анализа в песочнице);
- а также ряд специальных практик, таких как динамический анализ потоков распространения чувствительных данных (в т.ч. полносистемный).

В область требуемых практик ДА с точки зрения ГОСТ явно попадают динамический анализ (процесс № 11) и функциональное тестирование (процесс № 18). При этом в процессе № 11 в явном виде упоминается необходимость фаззинг-тестирования, пусть и без указания требования «генетического» свойства, однако с введением комплексного примечания, в настоящей редакции ГОСТ пока что рекомендующего добровольно подходить к решению задачи ДА с должным энтузиазмом:

«Используемые методы динамического анализа могут позволять осуществлять динамический анализ кода программы путем подачи заведомо некорректных входных данных, динамического профилирования, путем отладки программы, путем поиска защищаемой информации (в оперативной памяти, других местах среды исполнения кода), путем исследования поведения программы с использованием встраиваемых инструментальных датчиков срабатывания ошибок (санитайзеров) или инструментированных с использованием средств динамического двоичного инструментирования, другими применимыми методами, в том числе определенными соответствующими национальными стандартами»

Также процессы определения поверхности атаки (процесс № 7) и нефункционального тестирования (процесс № 19) не требуют, но подразумевают использование различных, характерных для ДА, подходов.

Начиная с апреля 2025 ФСТЭК России также публикует методические рекомендации по проведению тех или иных видов испытаний в Телеграм-канале @sdl_inform, входящем в группу информационных ресурсов РБПО-сообщества ФСТЭК России и ИСП РАН[7]. Вопросы применимости и требований методик и инструментов динамического анализа рассмотрены представителем ФСТЭК России И. С. Гефнер в рамках большого эфира[8], посвященного тематике РБПО.

3. ФАЗЗИНГ, ЧТО ЭТО ЗА ЗВЕРЬ?

Самое простое определение фаззинга — это подача на вход тестируемому объекту случайных данных (в надежде обнаружить какое-либо неожиданное — вплоть до аварийного завершения — поведение тестируемого объекта или даже системы в целом). Простейшие инструменты фаззинга — мутаторы значений параметров запросов, как правило, включены в комплект поставки упоминавшихся выше DAST-инструментов.

Однако, в силу принципиального отсутствия в мире технологических и вычислительных возможностей доказать формальную корректность какого-либо типового кода (даже для нескольких сотен тысяч исполняемых инструкций, за исключением частных случаев тривиального кода), вероятность найти все случаи неожиданного поведения методом подачи полностью случайных данных практически нулевая. Такой метод, разумеется, позволяет находить некоторые конкретные нештатные ситуации, однако существенно отстает от возможностей исследователя, вооруженного более продвинутыми инструментами, такими как генетические фаззеры. Инструменты данного типа могут автоматически оценивать эффективность собственной работы, выражающуюся в открытии новых состояний тестируемой программы, и до некоторой степени автоматически корректировать стратегию мутаций.

Как правило, в тех случаях, когда регуляторикой требуется фаззинг, подразумевается именно генетический фаззинг. Более подробный рассказ о данной технологии будет представлен в статье ИСП РАН в данном

---

[5] Выступление Елены Быхановой (НТЦ Фобос-НТ): https://arppsoft.ru/ie/19556/. Видео доступно по ссылке: https://t.me/sdl_community/8690

[6] Ссылка на презентацию: https://www.tbforum.ru/2025/program/information-security, ссылка на просмотр: https://clck.ru/3NcrPB

[7] https://t.me/sdl_community/7859

[8] https://t.me/sdl_community/8407

блоке, также различные материалы публикуются в Телеграм-канале @sdl_dynamic[9]. При этом важность (и сложность) технологии в целом подчеркивает в том числе тот факт, что именно упоминанию вопросов генетического фаззинга посвящено наибольшее число методических рекомендаций ФСТЭК России, публикуемых в Телеграм-канале @sdl_inform.

4. МОЖЕТ БЫТЬ ВСЕ-ТАКИ СДЕЛАТЬ ЕДИНЫЙ СТАНДАРТ ПО ДИНАМИЧЕСКОМУ АНАЛИЗУ?

Отличное предложение! В настоящий момент помимо «большого» ГОСТ Р 56939–2024 вступили в силу и действуют «инструментальные» ГОСТы: статический анализ исходного кода[10]; безопасный компилятор языков Си/Си++[11]. Также на финальной стадии подготовки находится стандарт по композиционному анализу программного обеспечения[12]. В планах ТК362 на 2025 год среди прочих значится разработка стандарта «Динамический анализ программного обеспечения»[13], автором стандарта является ИСП РАН. Работа над первой редакцией стандарта уже ведётся, на текущий момент с высокой степенью вероятности можно утверждать о попадании в редакцию стандарта как минимум следующих областей динамического анализа: модульное (и возможно комплексное функциональное) тестирование, фаззинг-тестирование, анализ потоков помеченных данных (как для уточнения поверхности атаки, так и для выявления утечек чувствительных данных).

5. И ЭТО ВСЁ? А КАК ЖЕ X, Y И ИИ?

Разумеется, нет! Список типов инструментов, выполняющих виды динамического анализа, полезные как на этапе создания ПО, так и на этапе его эксплуатации, можно продолжать достаточно долго — упомянем для примера следующие из них:

9 https://t.me/sdl_dynamic/9103

10 https://docs.cntd.ru/document/1304734159

11 https://docs.cntd.ru/document/1304734158

12 https://t.me/sdl_community/8716

13 https://fstec.ru/tk-362/deyatelnost-tk362/plany-rabot/plan-raboty-na-2025-god

ЗАМЕТКИ О ХОДЕ ИСПЫТАНИЙ СТАТИЧЕСКИХ АНАЛИЗАТОРОВ ИСХОДНОГО КОДА

5 августа 2025 г. на площадке кафедры ИУ10 МГТУ им. Н. Э. Баумана стартовал основной этап испытаний открытых и коммерческих статических анализаторов исходных кодов компилируемых и динамических языков программирования под патронажем ФСТЭК России.

Подробнее о целях и задачах испытаний, а также о функциональных возможностях участвующих в испытаниях инструментов статического анализа, вы можете узнать, ознакомившись с материалом «Испытания статических анализаторов»[1]. В материале «Итоги этапа «Домашнее задание» испытаний статических анализаторов под патронажем ФСТЭК России»[2] приведена краткая характеристика инструментов, сформированная по итогам выполнения первого из двух этапов испытаний, а также обозначены планы организаторов испытаний по развитию публичного репозитория комплектов тестов и стандартизации варианта формата SARIF, допускающего обмен результатами испытаний инструментов в рамках работ Центра исследований безопасности системного ПО ФСТЭК России[3].

Представление результатов испытаний ожидается в декабре 2025 года на полях Открытой конференции ИСП РАН в инновационном кластере «Ломоносов»[4] — главного ежегодного отечественного события в области безопасной и качественной разработки и традиционной площадки встреч участников РБПО-сообщества ФСТЭК России и ИСП РАН.

1 Испытания статических анализаторов // BIS Journal. — 2025. — № 2 (57). — С. 72–74.

2 https://ib-bank.ru/bisjournal/post/2508

3 https://portal.linuxtesting.ru/

4 https://www.isprasopen.ru/

- инструменты динамического определения поверхности атаки по коду (например, Natch от ИСП РАН);
- платформы контроля и безопасности контейнерной инфраструктуры, использующие движок eBPF (например, Luntry от Лантри);
- инструменты комплексного динамического исследования REST API (например, ПроAPI от Вебмониторэкс);
- полноценные песочницы, совмещенные со средствами антивирусной защиты (например, Sandbox от Лаборатории Касперского).

Отдельным и весьма актуальным является создаваемый прямо на наших глазах тип инструментов ДА, предназначенный для различных динамических исследований работы моделей систем искусственного интеллекта. Составить верхнеуровневое представление о созданных и развивающихся методах и инструментах анализа можно на основе доклада Вартана Падаряна (ИСП РАН) «Разработка безопасных технологий искусственного интеллекта»[14].

* * *

Далеко не все перечисленные инструменты в настоящий момент явно требуются во всех видах испытаний, либо подходят для анализа и защиты всех видах программных продуктов и систем.

Однако темпы развития общей инженерной культуры, равно как и соответствующие темпы развития регуляторных требований, не оставляют сомнения в том, что все полезные и применимые технологии анализа и защиты скорее рано, чем поздно будут введены в обязательный набор требований. И, разумеется, никто не запрещает ответственным разработчикам и эксплуатантам, заинтересованным в качестве, безопасности и надежности кода, ПО и систем, а также в повышении общей эффективности процессов их создания и эксплуатации, применять указанные средства анализа уже сейчас!

[14] https://www.itsecexpo.ru/2025/program/ai-tools-for-coding

Состоявшийся 5–6 августа запуск основного этапа испытаний в целом признан успешным. Несмотря на потребовавшийся от участников испытаний, членов жюри, площадки и организаторов испытаний существенный объем ресурсных вложений — в первую очередь в разработку комплектов тестов и репозитория их хранения, а также в создание портала централизованной разметки и адаптацию инструментов для взаимодействия с ним — основная часть коллективов приняла активнейшее участие в процессе подготовки. К указанным категориям участников испытаний добавилась новая категория — эксперты, в зоне ответственности которых первичная разметка и кросс-верификация срабатываний инструментов. В данную категорию включены представители Ассоциации ФинТех, АО «ИнфоТеКС», ООО «Постгрес Профессиональный», ООО «Свордфиш Секьюрити», АО Центр «Атомзащитаинформ».

На запуске основного этапа присутствовали представители ФСТЭК России и всех участвующих в испытаниях компаний и организаций. Почетная обязанность определить порядок начала испытаний инструментов была предоставлена Михаилу Павловичу Сычеву — основателю и бессменному лидеру кафедры ИУ10 «Защита информации». Основной задачей, успешно решенной всеми участниками испытаний 5–6 августа, являлось проведение статического анализа в отношении комплекта компонентов с открытым исходным кодом, сформированного организаторами испытаний на основе предложений жюри, и загрузка данных результатов на портал централизованной разметки — работы под кодовым названием «Блок Б». Начиная с 11 августа и до 1 ноября 2025 будет выполняться разметка и кросс-верификация разметки, результаты которой станут основой для подготовки итоговых отчетных материалов.

В середине сентября участники и жюри вновь соберутся в МГТУ для выполнения комплекта тестов под кодовым названием «Блок А». В состав данного комплекта входят тесты, позволяющие проверить реализацию инструментами различных свойств методов статического анализа, наличие у инструментов технологических возможностей по перехвату сборочного процесса и созданию собственных спецификаций на источники помеченных данных, а также возможность инструментов успешно анализировать крупные проекты в установленные ГОСТ Р 71207–2024[5] предельные временные рамки — нагрузочное тестирование.

М. П. Сычев

[5] https://docs.cntd.ru/document/1304734159

ДИНАМИЧЕСКИЙ АНАЛИЗ

КАК ВАЖНЫЙ ЭТАП РАЗРАБОТКИ БЕЗОПАСНОГО ПО

Вартан ПАДАРЯН
заведующий лабораторией ИСП РАН, руководитель органа по сертификации процессов РБПО

ИСП РАН уже более пятнадцати лет проводит фундаментальные и прикладные исследования, связанные с разработкой и применением технологий анализа программ для обеспечения кибербезопасности. Результаты исследований реализованы в виде комплекса инструментов, обеспечивающих разработку безопасного ПО. Наряду с широко известными инструментами статического анализа программ, такими как Svace[1] и Svacer[2], в ИСП РАН разработан ряд решений по динамическому анализу (ДА) ПО.

Одним из передовых способов ДА на текущий момент является гибридный фаззинг. Это сочетание классического фаззинг-тестирования с методами динамического символьного исполнения (ДСИ). ДСИ выполняет анализ программы, опираясь на ее логику и внутреннюю структуру. Для этого реальные входные данные заменяются на символьные переменные, с использованием которых происходит интерпретация бинарного кода и построение математической модели исполнения программы. На основе данной модели генерируются новые входные данные, которые приводят к увеличению тестового покрытия или воспроизведению ошибок.

Гибридный фаззинг зарекомендовал себя как наиболее эффективный метод тестирования. Он объединяет в себе скорость классического фаззинга и способность к открытию труднодостижимых участков кода ДСИ. Тем самым быстрее достигает большего покрытия и обнаруживает больше ошибок, чем оба метода по отдельности.

Комплекс Sydr-Fuzz[3], разрабатываемый в ИСП РАН, позволяет организовывать гибридный фаззинг бинарных программ с помощью открытых инструментов фаззинга (libFuzzer, AFL++, Honggfuzz) и инструмента ДСИ Sydr. Sydr-Fuzz одновременно запускает заданное число процессов фаззинга и символьного исполнения и по ходу работы синхронизирует корпуса данных между ними: отсеивает лучшие пути для анализа методом ДСИ и отбирает полезные входные данные для фаззера. Sydr-Fuzz также контролирует работу инструментов, отслеживает статистику их работы и условия для завершения фаззинга.

Инструмент ДСИ Sydr, для увеличения покрытия генерирует новые входные данные для анализируемой программы путем инвертирования условных переходов, зависящих от входных данных. Также Sydr осуществляет целенаправленный поиск ошибок с помощью символьных предикатов безопасности. Sydr определяет потенциально опасные участки в бинарном коде и пытается подобрать входные данные для воспроизведения ошибки. В Sydr поддерживаются предикаты безопасности для обнаружения разыменования нулевого указателя, деления на ноль, переполнения буфера, целочисленного переполнения и усечения, инъекции форматной и командной строки.

Sydr-Fuzz поддерживает гибридный фаззинг для компилируемых программ на языках C/C++/Rust/Go и C# (при использовании компиляции Native AOT) на ОС семейства Linux. Также, в нем реализована поддержка анализа на архитектурах x86/x86–64, ARM64 (в т.ч. Байкал) и RISC–V.

Sydr-Fuzz поддерживает пайплайн ДА для компилируемого кода, включая следующие этапы: анализ, минимизация итогового корпуса входных данных, сбор покрытия и анализ аварийных завершений. Также реализована поддержка программ на Python/Java/JavaScript/Lua. Вместо гибридного фаззинга на этапе анализа для таких проектов выполняется обычное фаззинг-тестирование с использованием открытых инструментов Atheris, Jazzer, Jazzer.js, Sharpfuzz, luzer. Sydr-Fuzz реализует процедуры запуска фаззинга, минимизации корпуса, сбора покрытия и анализа аварийных завершений для каждого тулчейна, что сильно облегчает процесс автоматизации ДА.

Для автоматического анализа аварийных завершений Sydr-Fuzz использует инструмент CASR[4]. Он позволяет генерировать отчеты о каждой обнаруженной в процессе анализа ошибке, удалять дубликаты и кластеризовать оставшиеся ошибки по степени похожести стека вызовов. Отчеты об ошибках генерируются на основе информации от санитайзеров или gdb и содержат всю необходимую информацию для аналитика, в т.ч. стек вызовов и участок исходного кода с ошибкой. CASR являет-

[1] https://www.ispras.ru/technologies/svace/
[2] https://www.ispras.ru/technologies/svacer/
[3] https://www.ispras.ru/technologies/sydr/
[4] https://www.ispras.ru/technologies/casr/

ся мощным инструментом, позволяющий существенно облегчить разбор результатов фаззинга, где зачастую могут находиться сотни различных аварийных завершений.

Все инструменты ДА программы позволяют анализировать только тот код, который был реально выполнен. Поэтому важно выбрать такой набор тестов или сценариев работы, который покрывал бы все важные функции.

Для сбора сведений о выполнении программы, ее прерывают, обеспечив работу инструментального кода. Для этого анализатор заведомо замедляет целевую программу. Инструментальный код встраивается в код программы на этапе компиляции или предобработки бинарного файла перед запуском (статическое инструментирование). Другой подход — это добавление инструментального кода непосредственно во время выполнения фрагментов целевой программы (динамическое инструментирование).

В ИСП РАН разработан инструмент Natch[5], предназначенный для определения поверхности атаки (ПА) приложений. Он построен на основе виртуальной машины QEMU и использует динамическое инструментирование.

ПА ПО — это те интерфейсы или программные компоненты (функции, модули, программы, библиотеки), которые получают данные из недоверенных источников. Так как при отправке модифицированных данных на интерфейсы или в программные компоненты возможно нарушить работу программы.

Разработчики определяют поверхность атаки своего ПО для тестирования (в том числе с помощью фаззинга) этих функций. Также поиск поверхности атаки необходим для того, чтобы найти зависимости от лишних или устаревших библиотек и пр.

Для определения ПА можно применить статический анализ (построение зависимостей функций, просмотр кода через IDE) или ДА (поиск покрытия кода набором тестов). При этом, статический анализ не способен в принципе находить некоторые типы зависимостей, к примеру, из-за динамического связывания, с чем может помочь Natch. А ДА (при достаточно хорошем наборе тестов) находит именно тот код, который выполнялся. Но сведения о выполненных функциях (и строках кода) недостаточно точно отражают характер этих функций. Такой метод не создает ясности зависят ли функции от входных данных. Например, код инициализации выполняется всегда, но злоумышленник редко может на него повлиять. Поэтому этот код не должен быть включён в ПА. Для определения более точной ПА необходимо отследить те программные модули, которые взаимодействовали со входными данными из недоверенных источников, в чем полезен Natch.

Гибридный фаззинг зарекомендовал себя как наиболее эффективный метод тестирования. Он объединяет скорость классического фаззинга и способность к открытию труднодостижимых участков кода ДСИ

Чтобы выделить из множества выполненных функций только те, что относятся к ПА (то есть обрабатывающие небезопасные данные), Natch помечает входные данные из недоверенных источников, специфицированных пользователем, а затем средствами эмулятора отслеживает их распространение по памяти виртуальной машины. Каждая процессорная инструкция, копирующая помеченные данные, копирует также и пометку. Что позволяет узнать, куда передавались и где обрабатывались входные данные и получить поверхность атаки, интересующую аналитика.

Natch анализирует не отдельные приложения, а виртуальную машину целиком, что позволяет проследить работу с данными, перетекающими между сервисами. При этом необходимо: помещать исследуемую программу в контролируемое окружение (виртуальную машину), а также разделять выполняемый в виртуальной машине код между высокоуровневыми сущностями (процессами, модулями) для представления пользователю.

В базовой версии QEMU включен только эмулятор gdb-сервера для подключения отладчика и логирование выполняющихся инструкций. Для того, какие именно приложения выполняются в гостевой системе, этого недостаточно. Поэтому Natch загружает в QEMU плагины, которые инструментируют выполняемый код. С помощью этого в гостевой системе определяются обращения к файлам, переключение между процессами. Восстанавливается стек вызовов всех приложений, собирается информация о покрытии кода. Также Natch умеет распознавать выполнение функций из программ на Python и JavaScript и даже скомпилированного байт-кода Java.

Natch может анализировать виртуальные машины под управлением Linux, Windows или FreeBSD для архитектур x86_64 и AArch64. Генерация ПА и стека вызовов работает для компилируемых языков типа C/C++, а также Go, Python, Java, C# и JavaScript.

ИСП РАН продолжает исследования безопасности ПО и развитие технологий ДА, в том числе ИСП Crush[6], Блесна[7] и пр. О новых подходах и развитии технологий исследований безопасности ПО можно узнать на официальном сайте ИСП РАН[8].

[5] Описание: https://www.ispras.ru/technologies/natch/. Запись доклада на ТБ Форум 25: https://clck.ru/3Necyo. Слайды доклада на ТБ Форум 25: https://clck.ru/3Neczn

[6] https://www.ispras.ru/technologies/crusher/

[7] https://www.ispras.ru/technologies/blesna/

[8] https://www.ispras.ru/news/

ПРОБЛЕМЫ ДИНАМИЧЕСКОГО АНАЛИЗА ВЕБ-ПРИЛОЖЕНИЙ И API

КАК МЫ ИХ РЕШАЕМ

Валерий КУВАЕВ
руководитель направления защищенной разработки группы компаний SolidLab

Чтобы использовать DAST на практике, заказчик должен понимать, как работает этот класс решений, для каких целей его нужно выбирать, какие приложения и в каких сценариях необходимо анализировать и, наконец, как работают собственные сканируемые приложения. В этой статье мы разберем некоторые аспекты динамического анализа веб-приложений с упором на актуальный опыт применения и пилотирования нашего решения по динамическому анализу SolidPoint DAST.

РАЗБИРАЕМ ТЕРМИНЫ

Следует разграничить термины «DAST», «динамический анализ», «сканер уязвимостей» и «сканер безопасности». Путаницу в терминах я встречал как в формальных взаимодействиях с заказчиками, так и в личных беседах, конечно, она осложняет взаимопонимание, и может приводить к ложным ожиданиям.

Если говорить о терминах проще, то:

- «Динамический анализ» — общее название методов анализа ПО во время его выполнения. Сюда входит не только безопасность, но и проверка производительности и функциональности, анализ поведения ПО, сетевая активность, работа с памятью и другие категории.
- «Сканеры уязвимостей» — это подкласс «сканеров безопасности». Они используют разные методы, включая динамические, для поиска уязвимостей в широком стеке инфраструктуры и ПО. Чаще всего сканеры отрабатывают уязвимости из классификатора CVE. При этом «сканеры безопасности» могут идентифицировать более широкую номенклатуру рисков. Оба термина применяются достаточно вольно для обозначения инструментов, представленных на рынке.
- «DAST» — наиболее узкий термин. Он указывает на класс сканеров, применяющих динамический анализ для идентификации недостатков безопасности веб-приложений и API без доступа к коду, методом «черного ящика». Результатом работы DAST, как правило, является список выявленных недостатков ПО, категорированных по CWE.

Таким образом, под DAST мы понимаем специфический класс инструментов для анализа безопасности веб-приложений и API. Хотелось бы, чтобы на рынке вся терминология становилась более унифицированной. Разница в понимании ведёт к потере времени при переговорах, что критично в условиях срочных запросов и интенсивных коммуникаций. Сейчас мы рекомендуем согласовать все термины ещё на старте.

Нередко мы встречаем конкурсные процедуры, где указаны широкие требования для сканеров уязвимостей, например, по поддержке протоколов, не имеющих отношения к вебу. При этом конкурс заявлен на выбор инструмента DAST. Добавлю, что есть некоторая надежда на то, что ожидаемый новый ГОСТ по динамическому анализу зафиксирует терминологию и взаимодействие станет проще.

ЧТО ЗНАЧИТ ПРАВИЛЬНАЯ ПОДГОТОВКА К СКАНИРОВАНИЮ

Основное, если вы не знаете, как устроено ваше приложение, которое вы хотите сканировать, а также инфраструктура его доставки, то эффективности от DAST'а ожидать не приходиться. Опыт пилотных внедрений показал, что успешное сканирование требует тщательной подготовки. Были случаи, когда мы сталкивались со сложностями при сборе информации о приложении и при проведении первичного сканирования: недостоверная устаревшая или неполная информация, систематические ошибки в полученных «на руки» данных, низкий показатель стабильности и доступности стендов.

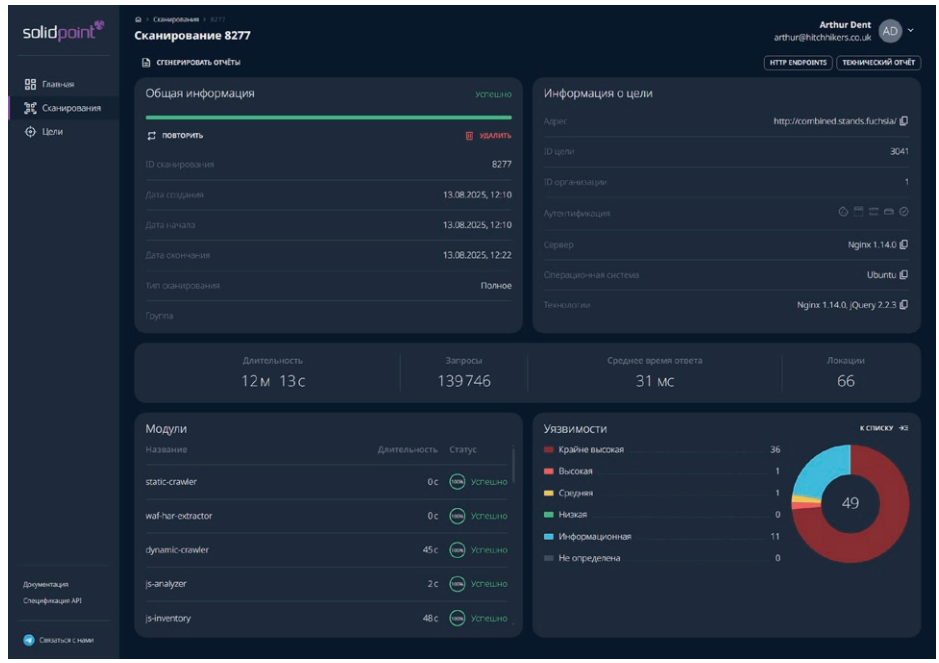
По сути, инструменты DAST умеют сканировать приложения без ка-

кой-либо подготовки, даже без данных для аутентификации. Но обычно всё интересное закрыто от доступа, поэтому первое, с чем необходимо определиться, — это какую инсталляцию приложения мы сканируем, какие данные по аутентификации нам нужны, достаточно ли сканирования под одной ролью или нужно под разными, как работает механизм аутентификации, ограничено ли время сессии, и если да, то как конкретно? Зачастую типы аутентификации и применяемые методы могут смешиваться, тогда инструмент должен уметь их применить «по месту».

С помощью SolidPoint DAST мы поддерживаем различные механизмы: аутентификация по кукам, токенам, HTTP Basic, по клиентским сертификатам, через локальное хранилище браузера (localStorage). Наконец, существуют более сложные сценарии — запись браузерного скрипта с данными аутентификации и метод аутентификации по HTTP-запросу, например, для работы с JWT-токенами и для обновления короткоживущих сессий. При этом для сканируемого приложения можно задать целую группу данных аутентификации для основного приложения и для входящих в приложение API, находящихся на соседних URL, — сканер сам применяет эти данные.

Для подготовки сканирования следом идут дополнительные вопросы, есть ли CAPTCHA, OTP, SMS, можно ли как-то зафиксировать их значения в тестовых средах и что делать с ними в продуктивной среде, поскольку динамически подгрузить их в запущенное сканирование может оказаться проблематичным. Следующий вопрос — как ограничена нагрузка в продуктивной или в тестовой среде? Зачастую тестовые среды сами по себе менее мощные, а в продуктивной не хочется влиять на производственный процесс. Важно понять предел нагрузки, сколько условных RPS можно разрешить сканеру, этот же параметр, кстати, напрямую повлияет на время сканирования.

Есть ли WAF или иные системы, блокирующие трафик? Если да, необходимо внести сканер в белый список.



Также, нужно выявить, какие конечные точки приложения нельзя затрагивать сканированием. В первую очередь, это касается конечных точек, приводящих к выходу из сессии — «разлогину». Еще в продуктивных приложениях это могут быть конечные точки, связанные с вызовом внешних интеграций, например, SMSGateway — мы же не хотим исчерпать весь бюджет платного сервиса, отправив тысячи SMS в никуда? Также могут быть конечные точки, связанные с критическим бизнес-процессом, например, с кнопкой «оплатить». Обычно на сайте это последний этап воронки продаж, и он должен работать идеально.

Сканирование — это отправка множества запросов, не всегда корректных функционально, для проверки того, как приложение отреагирует. Приложение может начать регистрировать поток ошибок. В ситуации, когда сканер внесён в белый список и средства защиты пропускают его трафик, это может неожиданно для всех привести к реальным операционным рискам, например, вызвав DoS. Но чаще всего регистрируемые ошибки могут замаскировать сопутствующую реальную проблему, и служба мониторинга имеет все шансы за штормом событий её пропустить, нарушив SLA и, как следствие, не выполнив KPI.

Также возникает вопрос о длительности сканирования. Как правило, время зависит от размера приложения, сложности конечных точек приложения — количества параметров в каждой, а также от скорости сети и выбранного RPS. При сканировании составляется карта приложения и различными методами проверяется каждый выявленный объект: конечная точка или ресурс на сайте, атаки посылаются по каждому выявленному параметру. Все эти факторы в совокупности влияют непредсказуемо сложно, и чтобы ответить точно, потребуется провести первое полноценное сканирование.

Вытекающий вопрос из предыдущего — а сколько узлов сканирования понадобиться для реальной промышленной инсталляции? До первого полноценного сканирования каждого приложения и API ответить сложно, т.е. это становится понятным в середине внедрения. По этой причине в сканере SolidPoint DAST поддерживается горизонтальное масштабирование и количество узлов не ограничено лицензией.

Таким образом польза от DAST прямо пропорциональна степени подготовки к анализу и коректности настройки сканирования, которые невозможны без глубокого понимания сканируемого приложения, его

механизмов и инфраструктуры доставки. Свой вклад в полезность несут и инструменты, и на мой взгляд недооцененной является сложность составления карты поверхности атаки — списка конечных точек и ресурсов с помощью которых возможно взаимодействие с сервером.

ТЕХНИЧЕСКИЕ ПРОБЛЕМЫ АНАЛИЗА ПОВЕРХНОСТИ АТАКИ

Несмотря на то, что DAST как класс решений работает с приложениями как с «черным ящиком» (т.е. через его внешние интерфейсы), влияние технологий разработки сканируемого приложения все же сказывается на результатах сканирования. Так было при появлении SPA приложений, после чего сканеры на рынке дружно заявили о всяческой поддержке SPA — почти сразу, как ее реализовали.

Возникает вопрос, насколько качественно средства динамического анализа определяют поверхность атаки? Ответ не очевиден, ведь недостаточное покрытие не приводит к ложным срабатываниям, на которые можно указать, а приводит к ложно негативным результатам, оставляя реальный скоуп приложения и уязвимости без внимания. Ведь чтобы просканировать функцию на стороне сервера, надо сначала найти, как ее вызвать — узнать формат запроса, параметров, а дальше уже проверять на уязвимости; если инструмент не умеет с хорошей полнотой идентифицировать функции на стороне сервера — т.е. поверхность атаки — до этапа тестирования на уязвимости этой функции инструмент даже не доберется, т.е. фактически приложение останется просто непроанализированным. Эти проблемы характерны для современных приложений с изощренной сложностью интерфейсов и глубокой зависимостью от разнообразных JS-фреймворков.

В исследовании “The Security Tools Gap”[1], проведенном компанией Axeinos в марте 2025 года, прямо сказано, что при определении поверхности атаки BlackBox сканеры очень быстро сдаются, оставляя значительную часть приложения неисследованной, причем чем выше сложность приложения, тем больше эта проблема. Получается, что изначально, сканеры ищут уязвимости и ошибки не в приложении, а в его малой части, при этом существующие технологии динамического краулинга не могут компенсировать это упущение, нужно что-то иное.

Значительно ранее, проводя многочисленные пентесты для наших заказчиков, мы обнаружили, что периодически встречаются запросы, хранящиеся в коде JS, не имеющие соответствующей части в UI. Стало понятно, что классические методы динамического краулинга, не способны их найти. Кроме того, у них есть дополнительные ограничения: страницы приложения могут создаваться динамически, что удлиняет и иногда зацикливает анализ, не всегда элементы управления и связанные с ними функции получается идентифицировать, не все введенные краулером данные оказываются корректными, чтобы приложение перешло на следующий экран. По этой причине при разработке своего сканера мы уделили особое внимание этой проблеме, разработав специальный модуль статико-динамического анализа клиентского кода — анализатор JavaScript-кода[2]. Мы рассказывали об этой технологии на Standoff в 2022 году[3]. Удобными особенностями JS-анализатора являются относительная быстрота анализа, игнорирование ветвлений, т.е. весь загруженный код может быть проанализирован без ограничений логики работы приложения, и в нем могут быть найдены необходимые вызовы, включая необходимый формат и параметры. Также, если приложение отдает сразу весь JS-код, появляется возможность без авторизации получить полную карту конечных точек, включая связанные с административными функциями. Также этот анализ интересен пентестерам, с его помощью можно быстро получить информацию о поверхности атаки приложения заказчика.

Таким образом, в SolidPoint DAST проблема обнаружения возможной поверхности атаки и как следствие, проблема сужения проверяемого скоупа приложения эффективно решаются спектром инструментов: статическим и динамическим краулерами, дирбастингом, и уникальным JS-анализатором. Для получения дополнительной информации о конечных точках приложения сканер может интегрироваться с SolidWall WAF, а также способен импортировать спецификацию OpenAPI/Swagger, SOAP, GraphQL сервиса.

О РЕШЕНИИ SOLIDPOINT DAST

SolidPoint DAST входит в реестр отечественного ПО. Разработчиком кода является компания «СолидСофт», которая активно развивает продукт с оглядкой на потребности российских заказчиков.

Мы предлагаем внедрять инструмент SolidPoint DAST как централизованную систему автоматического динамического анализа веб-приложений и API, которая включает возможности масштабирования задач сканирования с единым центром управления и поддержкой горизонтального масштабирования мощностей сканирования. Централизованное управление и разделение доступов позволяют организовать работу множества подразделений одновременно, оптимизируя потребляемые ресурсы. SolidPoint DAST из коробки поддерживает интеграции в различные автоматизированные процессы, как для анализа продуктивных сред, так и для интеграции в среды разработки, с помощью REST API или CLI, взаимодействующего с REST API. Также можно автоматизировать выгрузку результатов сканирования во внешние системы ASOC или интегрироваться с системами управления задачами, например Jira, Redmine и т.п. Таким образом, SolidPoint DAST может решать большой спектр задач автоматизированного сканирования веб-приложений и API на практике.

[1] http://web.archive.org/web/20250424002323/https://axeinos.co/report/The%20Security%20Tools%20Gap.pdf

[2] http://journals.tsu.ru/engine/download.php?id=223281&area=files

[3] https://www.youtube.com/watch?v=vG0EzOr81pE

kaspersky активируй будущее

Защита организации от сложных и целевых атак

Kaspersky Anti Targeted Attack

с модулем NDR

ДиалогНаука

СИСТЕМНАЯ ИНТЕГРАЦИЯ В ОБЛАСТИ ИНФОРМАЦИОННОЙ БЕЗОПАСНОСТИ

dialognauka.ru

С ФОКУСОМ НА БЕЗОПАСНОСТЬ ПРИЛОЖЕНИЙ

ВКЛЮЧАТЬ DAST В ПРОЦЕССЫ СТОИТ РАЗУМНО И ПОСТЕПЕННО

Анна МЕЛЕХОВА
Senior Security Architect, Kaspersky OS

Если вы интересуетесь темой безопасной разработки, то, наверняка читали прошлый выпуск РБПО, посвящённый статическим анализаторам. Сегодня мы поговорим о следующей практике безопасной разработки – динамическом анализе кода (DAST). В статье сосредоточусь на санитайзерах (sanitizers), подклассе инструментов динамического анализа в инструментарии C/C++ разработчиков.

СУТЬ ДЕЛА

Так что же такое динамический анализ и чем он отличается от статического? В первую очередь это процесс тестирования уже скомпилированного кода (либо его частей) в динамике (во время исполнения). Как это происходит? Создаём специальную «санитизированную» сборку с помощью всего лишь дополнительных опций компилятора (-fsanitize=). Такая сборка будет содержать дополнительную инструментацию — некие универсальные проверки, которые будут отрабатывать в ходе выполнения приложения. Тестирование подготовленной сборки и есть процесс динамического анализа.

ПРОВЕРКИ

Сразу становится интересным: какие могут быть проверки, которые не зависят от задач и позволяют санитайзерам быть полезными и для hello_world, и для условного SAP. Это проверки некоторых низкоуровневых инвариантов, нарушение которых приводит к серьёзным ошибкам. И, поскольку мы говорим о безопасном программировании, то речь об ошибках, которые могут использовать атакующие. Например, санитайзер встроит проверку, что к освобождённой памяти не будет обращения (защищая от UAF — use-after-free). Или предотвратит обращение за границу массива на стеке (OOB — out-of-

bound). Также можно проверить на утечки, работу с памятью, надёжность многопоточной работы, отсутствие неопределённого поведения и другое. С помощью опций компилятора можно настроить не только объём проверок, но и поведение программы в случае срабатывания. В зависимости от способа использования динамического анализа в вашей компании разные варианты реакций могут быть полезны — от автоматического оповещения SOC до тихого подавления или рестарта приложения.

О СКРЫТЫХ ПРОБЛЕМАХ

Итак, есть универсальные полезные проверки. И чтобы включить их, нужно лишь добавить флаги компиляции. Значит, достаточно небольшого изменения в пайплайне сборки, чтобы динамический анализ работал и повышал безопасность продукта? И неужели нет скрытых проблем? Конечно, есть.

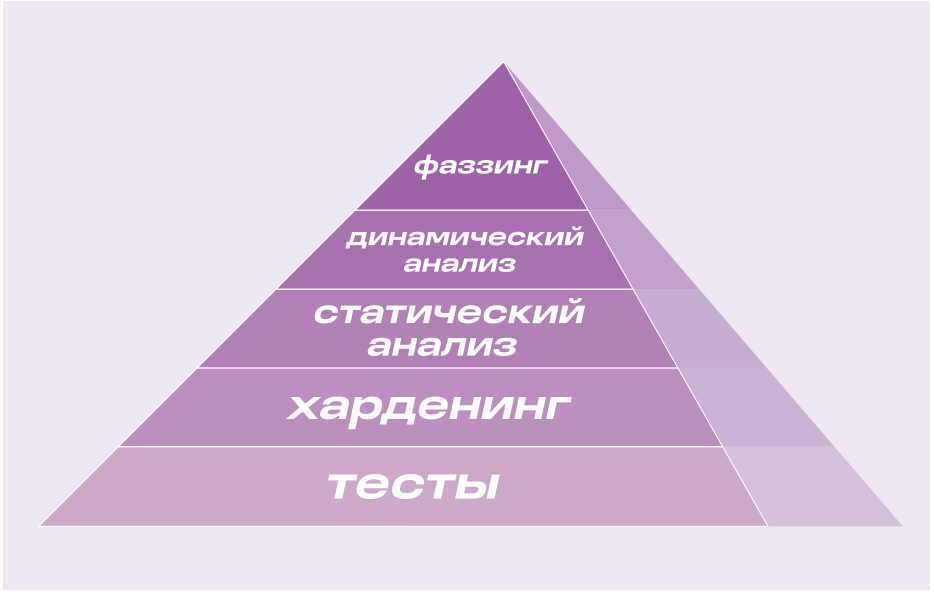
Во-первых, полезность санитайзеров прямо пропорциональна качеству ваших тестов, обычных и фаззинговых. Если санитайзеры включены для теста с покрытием 1%, вероятность найти утечку памяти или UAF существенно снижается, в отличие от статического анализа. Если до юнит-тестирования вы пока не добрались, браться за DAST нецелесообразно.

Во-вторых, санитайзеры существенно замедляют работу: от 1,5 до 5–6 раз (!) в зависимости от объёма инструментации и характера нагрузки. А значит, многие тесты будут завершаться аварийно по time-out и разработчик будет злиться на краснеющий пайплайн. Когда пайплайны существенно замедлятся, разработчик ради стабильности и надёжности отключит динамический анализ для части тестов. От этого естественным образом снизится и полезность DAST, смотри пункт первый. Ну и при таких замедлениях и речи не идёт о том, чтобы включать динамический анализ в «релизе».

ДВЕ НОВОСТИ

Выходит, что полагаться только на динамический анализ нерационально? Да, но есть и хорошая новость. Некоторые проверки можно выполнить другими средствами. Например, чтобы обнаружить переполнение целочисленного знакового, можно использовать как динамический (-fsanitize=signed-integer-overflow), так и статический анализ. Статический анализ способен найти доступ к неинициализированной переменной. Однако статический анализ тоже раздражает разработчиков: для некоторых правил он выдаёт до 30% ложноположительных срабатываний. Случаются и ложноотрицательные, когда реальная проблема упускается.

Также некоторые проверки из арсенала динамического анализа можно выполнять через харденинги — другой вид инструментирования приложения. При скромном замедлении на 3–10% мы защитим приложение не только от ошибок разработчика, но и от намеренных повреждений при распространении атаки. Например, опция компилятора FORTIFY_SOURCE заменяет обращение к «небезопасным» функциям стандартной библиотеки дополнительными безопасными вставками и так позволяет избегать доступа вне диапазона (OOB). Аппаратный механизм MTE (memory tagging extension) с некоторой вероятностью защищает и от OOB и ошибок вида type confusion. Однако, этот механизм доступен не на всех аппаратных платформах.



УВЫ, НУЖЕН!

Так динамический анализ не нужен? Увы, нужен. Многие проверки не сделаешь иначе как санитайзерами. Это большая часть случаев undefined behavior, утечки памяти и нарушения memory safety разных сортов на «старых» аппаратных платформах или в виртуализованных окружениях. Фаззинговые тесты вообще бесполезны, если не настроены проверки инвариантов, и самая простая из них — инструментирование от санитайзеров, тот самый динамический анализ. Но включать DAST в процессы стоит разумно и постепенно. Я предлагаю идти по пирамиде: тесты, харденинг, статический анализ, динамический анализ, фаззинг (см. рис.). Так вы балансируете уровни пользы и затрат, настраивая проверки под себя для оптимального результата.

* * *

А если искать баланс между разными инструментами и их настройками слишком сложно, можно ориентироваться на систему, созданную с фокусом на безопасность приложений. Мы, разрабатывая KasperskyOS, выстроили всю систему в парадигме конструктивной безопасности и адаптировали инструменты разработки, чтобы сформировать целостный подход к безопасности приложений.

СОВРЕМЕННЫЕ МЕТОДЫ ДИНАМИЧЕСКОГО АНАЛИЗА КОДА

ФАЗЗИНГ-ТЕСТИРОВАНИЕ И ЕГО КЛАССИФИКАЦИЯ

Илья АНТИПОВ
руководитель группы Центра оценки соответствия и тестирования АО «НПО «Эшелон»

Роберт АРУСТАМЯН
заместитель директора Центра оценки соответствия и тестирования АО «НПО «Эшелон»

Нелли МАГАКЕЛОВА
руководитель группы Центра оценки соответствия и тестирования АО «НПО «Эшелон»

ВВЕДЕНИЕ

Динамический анализ безопасности приложений (Dynamic Application Security Testing, DAST) прочно закрепился в арсенале средств обеспечения безопасности информации как метод оценки работающего приложения «извне», с позиции злоумышленника. Традиционные DAST-инструменты эффективно выявляют широкий спектр известных уязвимостей (таких как инъекции, уязвимости конфигурации, небезопасные десериализации), используя предопределённые наборы тестовых векторов и сигнатур, основанных на знаниях о распространённых атаках и уязвимых шаблонах. Тем не менее, несмотря на свою эффективность, классический DAST обладает существенным недостатком: он часто полагается на базы данных уже известных уязвимостей. Это делает его менее эффективным в обнаружении принципиально новых (zero-day) уязвимостей, сложных логических ошибок или уязвимостей, возникающих при обработке специфических, нестандартных или некорректно сформированных входных данных.

Фаззинг-тестирование, в свою очередь, представляет собой методику обеспечения безопасности, основанную на автоматизированной подаче в тестируемую систему большого объёма полуслучайных, искажённых или мутированных входных данных («корпусов») с целью спровоцировать сбои, аварийные завершения или иные неожиданные поведения программы. Именно способность фаззинга исследовать «пограничные случаи» и «неизведанные пути» выполнения кода делает его незаменимым для обнаружения уязвимостей, которые остаются невидимыми для традиционных, сигнатурно-ориентированных методов.

Целью данной статьи является детальное рассмотрение фаззинг-тестирования как активного метода поиска неожиданного поведения программы. В рамках статьи будут рассмотрены методологические аспекты фаззинг-тестирования, его виды, а также преимущества и недостатки.

Способность фаззинга исследовать «пограничные случаи» и «неизведанные пути» выполнения кода делает его незаменимым для обнаружения уязвимостей, которые остаются невидимыми для традиционных методов

ФАЗЗИНГ-ТЕСТИРОВАНИЕ ПРИЛОЖЕНИЙ

Фаззинг-тестирование представляет собой метод динамического анализа безопасности и устойчивости программного обеспечения путём подачи на его вход неверных, неожиданных или случайным образом сгенерированных данных. Его основная цель — выявление ошибок реализации, приводящих к сбоям (крашам), уязвимостям безопасности (таким как переполнение буфера), утечкам памяти или некорректному поведению программы. В соответствии с современными требованиями к надёжности ПО фаззинг-тестирование рекомендуется применять как к собственному коду, так и к интегрируемым сторонним компонентам (биб-

лиотекам, исполняемым файлам), доступным для тестирования в скомпилированном виде.

Первые подходы, схожие с фаззинг-тестированием, заключались в ручном или полуавтоматическом вводе некорректных данных для проверки реакции программы. Это могли быть попытки подачи на вход очень длинных строк, бинарного «мусора», невалидных параметров командной строки или файлов с повреждённой структурой. Подобные методы были трудоёмкими, несистемными и охватывали лишь малую часть возможных аномальных сценариев. По мере развития методологии появились специализированные инструменты, способные автоматически генерировать большие объёмы случайных или структурированных аномальных входных данных («корпусов») и массово подавать их на тестируемую программу, отслеживая её состояние на предмет сбоев. Изначально фокус этих инструментов смещался в сторону улучшения общего качества кода, но со временем акцент сменился на поиск критических уязвимостей безопасности. Это стало возможным благодаря чёткой постановке целей.

Эволюция фаззинг-тестирования от простых методов к сложным инструментам привела к появлению различных методик, классифицируемых по уровню инсайдерской информации, доступной тестеру. Этот уровень доступа напрямую определяет глубину анализа и эффективность поиска уязвимостей. Одним из ключевых критериев классификации фаззинг-тестирования является уровень доступа к внутренней информации тестируемой программы и степень её инструментирования. В зависимости от этого критерия выделяют следующие основные виды.

Фаззинг-тестирование методом чёрного ящика (black-box). Включает взаимодействие с программой исключительно через её стандартные интерфейсы (ввод/вывод, файлы, сеть) без какого-либо доступа к внутренней структуре. Ошибки выявляются путём наблюдения за внешними реакциями системы (падения, зависания, ошибки), моделируя атаку внешнего злоумышленника.

Фаззинг-тестирование методом серого ящика (grey-box). Включает инструментирование программы для сбора ограниченной внутренней информации, прежде всего о покрытии кода (какие базовые блоки или переходы были выполнены). Эта информация используется для управления генерацией/мутацией данных, отдавая приоритет входам, которые приводят к выполнению новых путей кода, что значительно повышает эффективность поиска глубоких уязвимостей.

Фаззинг-тестирование методом белого ящика (white-box). Включает полный доступ к исходному коду и состоянию программы. Использует методы динамического анализа (включая символьное выполнение) для построения ограничений на путях выполнения и целенаправленной генерации входных данных, удовлетворяющих этим ограничениям, теоретически позволяя находить самые сложные уязвимости, но требует детального изучения исходных текстов программы.

Выбор между black-box, grey-box и white-box фаззингом задаёт рамки возможностей, однако внутри каждой из этих методик критическую роль играет конкретная стратегия формирования входных данных, подаваемых на программу. Именно алгоритм генерации или модификации этих данных во многом определяет, какие участки кода будут протестированы и насколько глубоко. Ключевым аспектом фаззинг-тестирования является стратегия генерации входных данных. По этому критерию выделяют два основных подхода.

Выбор между black-box, grey-box и white-box фаззингом задаёт рамки возможностей, однако внутри каждой из этих методик критическую роль играет конкретная стратегия формирования входных данных, подаваемых на программу

Генерационный фаззинг. Включает использование формальной модели или спецификации ожидаемого формата входных данных (грамматика файла, протокола, API). Тестовые данные создаются «с нуля» на основе этой модели, целенаправленно формируя синтаксически корректные, но семантически аномальные входы для проверки глубоких ветвей кода и сложных состояний программы.

Мутационный фаззинг. Включает использование набора валидных входных данных (начальный корпус — seed corpus). Тестовые данные генерируются путём применения случайных искажений (битовые перевороты, замена/удаление/вставка байтов, изменение длины) к этим образцам, позволяя быстро находить ошибки обработки на поверхностном уровне. Для мутации (искажения) используются данные, полученные с помощью обратной связи от приложения. Примерами таких данных могут служить: информация об увеличении покрытия, сведения о новых путях исполнения кода или время выполнения программы.

Эффективность любой стратегии генерации данных — будь то генерационный или мутационный подход — необходимо измерять. Наиболее объективной метрикой здесь выступает покрытие кода, показывающее, какая часть программы была фактически протестирована фаззером. Разные способы расчёта покрытия дают существенно отличающуюся картину тестирования. С точки зрения получения информации

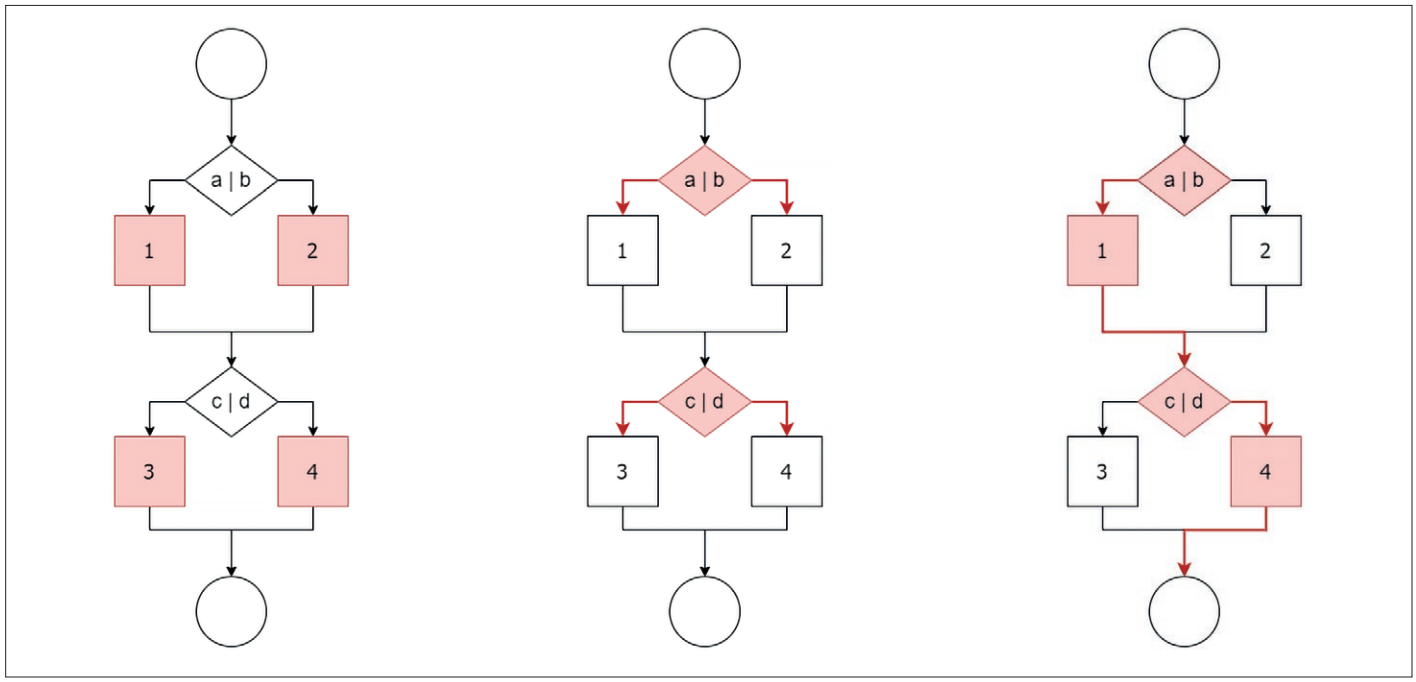


Рисунок 1. (а) — тестовое покрытие по базовым блокам, (б) — тестовое покрытие по ветвям условных переходов, (в) — тестовое покрытие по путям исполнения в программе

по покрытию, которое получается в результате проведения фаззинг-тестирования, выделяются три подхода (рис. 1).

При этом в зависимости от способа сбора тестового покрытия процент, которого достиг фаззер, будет сильно меняться. Если входное значение прошло по блокам 1 и 4, то по базовым блокам мы достигли 50% покрытия (блоки 1 и 4), по условным ветвям мы достигли 50% покрытия (условные переходы a и d), по путям исполнения мы достигли только 25% покрытия (путь a-1-d-4).

Показатели покрытия — важный, но не единственный критерий оценки фаззинга. Для понимания реальной ценности этого метода для тестирования безопасности необходимо взвесить все его сильные и слабые стороны в сравнении с другими подходами. Говоря об эффективности фаззинг-тестирования, стоит учитывать и недостатки. Далее представлена таблица с преимуществами и недостатками фаззинг-тестирования по сравнению с иными подходами к обнаружению ошибок в программах.

Преимущества	Недостатки
Нахождение проблем безопасности, которые невозможно обнаружить при других анализах	Сложность настройки и анализа
Высокая степень автоматизации	Сильная зависимость от инструментации
Эффективность для сложных форматов данных	Низкая эффективность при наличии в программе большого количества входных данных
Высокая масштабируемость	Возможность пропуска редких путей выполнения
Работа на конкретном наборе входных данных	Трудоёмкость и сильные временные затраты
Возможность проведения без наличия исходных текстов программы	Ресурсоёмкость и высокие вычислительные мощности

Таблица. Преимущества и недостатки фаззинг-тестирования

ВЫВОД

Инструменты динамического анализа, и особенно фаззинг-тестирование, играют критически важную роль в обеспечении безопасности и надёжности программного обеспечения, выявляя уязвимости, проявляющиеся исключительно во время выполнения. В отличие от статического анализа, который исследует код, не подразумевая его исполнения, динамический анализ проверяет поведение работающей программы в реальных или близких к реальным условиях. Его ключевая особенность — способность находить сложные, эксплуатируемые баги в обработчиках недоверенных данных (парсерах, протоколах, API), которые часто ускользают от других методов тестирования. Несмотря на свою ресурсоёмкость и необходимость настройки, фаззинг-тестирование является незаменимым элементом современного подхода к разработке безопасного программного обеспечения, дополняя статический анализ и обеспечивая выявление критических уязвимостей на этапе тестирования.

ФАЗЗИНГ-ТЕСТИРОВАНИЕ

ОТ ПРОСТОГО К СЛОЖНОМУ

Степан ХАРИТОНОВ
руководитель направления
безопасной разработки ООО «НПЦ «КСБ»

С августа 2024 года проектом OSS-Fuzz зарегистрировано не менее 750 подтверждённых уязвимостей в open-source-компонентах. Более 30 исправлений ошибок и уязвимостей в коде за этот же период внесено Центром исследования безопасности системного ПО, сформированным при участии отечественных экспертов. Эти цифры объединяет одно: данные уязвимости выявлены видом динамического анализа кода под названием «фаззинг». Тестировщики, которые впервые с ним знакомятся, ассоциируют его с анализом граничных значений, оценкой на основе эквивалентных классов, тестированием property-based. Действительно, эти виды тестирования отчасти покрываются в ходе фаззинга, но всплывают и нюансы, которые делают процедуру не такой уж и простой.

ФАЗЗИНГ: ЧТО ЭТО?

Фаззинг — подход к анализу кода во время его исполнения, основанный на оценке работы программы в ответ на подачу на вход множества невалидных значений. Гибкость в его проведении достигается широкими возможностями:

- в зависимости от оценки выполнения кода фаззинг проводится по принципам «белого», «серого» или «чёрного» ящиков;
- подход к формированию входных значений (семплов) может быть генерационным — не связанным с процессом фаззинга — или мутационным — основанным на получаемом покрытии кода;
- целью исследования может быть как конечный исполняемый файл, так и конкретная библиотечная функция.

Большой ассортимент существующих инструментов, фаззеров, «даёт разгуляться». Однако такая вариативность создаёт определённую сложность в реализации фаззинг-тестирования.

С ЧЕГО НАЧАТЬ?

Простой фаззинг можно запустить «на коленке», достаточно определить цель. В таком случае вы с легкостью сможете выполнить генерационный фаззинг методом «чёрного» ящика: входные семплы генерируются случайно, информации о покрытии кода нет, целью выступает конечный исполняемый процесс, который можно запустить и вручную. Такой подход зачастую применим при фаззинге веб-приложений: программа-клиент отправляет на сервер запросы, в их содержимое подставляются случайные данные. Результат ожидаемо «сырой»: искать «вредные» семплы необходимо самостоятельно.

К ЧЕМУ СТРЕМИТЬСЯ?

Освоение мутационного фаззинга методом «серого» ящика удобно начать с формирования на базе юнит-тестов «фаззинг-обёрток» — программ, вызывающих исследуемый код и подающих на вход данные от фаззера. Интересными целями выступают библиотечные функции: строкочувствительные, утилитарные, с высокой цикломатической сложностью, парсеры, десериализаторы и т.п. Их тест завязан на трёх элементах:

- проинструментированная библиотека;
- фаззер;
- скомпилированная обёртка с входным набором данных, обеспечивающих стартовое покрытие.

Дополнить процесс можно:

- мутаторами, увеличивающими входной набор данных;
- словарями, насыщающими семплы значениями, влияющими на ход выполнения кода.

Работа фаззера сводится к увеличению числа уникальных путей исполнения кода. Но следует помнить, что важен не только конечный уровень покрытия, но и общее число запусков цели. Также для повышения эффективности поиска дефектов при фаззинге, предлагается использовать средства отлова ошибок — санитайзеры, способные фиксировать отклонения от штатного выполнения программы.

ИГРА СТОИТ СВЕЧ

Фаззинг давно доказал свою эффективность, как технология, способствующая повышению качества и безопасности продуктов. Современные тенденции превращают внедрение этой разновидности динамического анализа кода из опционального метода тестирования в устойчивую практику, демонстрирующую практический эффект в поиске ошибок и уязвимостей, которая уже сейчас применяется отечественными разработчиками.

АНАЛИЗАТОР КОДА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ АК-ВС 3

ИНСТРУМЕНТ, КОТОРЫЙ СОВМЕЩАЕТ В СЕБЕ ТРИ ВИДА ДИНАМИЧЕСКОГО АНАЛИЗА

Виталий ВАРЕНИЦА
заместитель генерального директора АО «НПО «Эшелон»

ВВЕДЕНИЕ

«АК-ВС 3» представляет собой инструмент, предназначенный для автоматизации процесса проведения испытаний при сертификации в соответствии с требованиями различных регуляторов, а также для выполнения периодических проверок в рамках разработки безопасного программного обеспечения. Инструмент объединяет три ключевых метода динамического анализа, которые позволяют детально исследовать поведение программы в условиях реального исполнения, обнаруживать как типовые, так и трудноуловимые ошибки, и обеспечивать соответствие современным требованиям к безопасности программного обеспечения.

ДИНАМИЧЕСКИЙ АНАЛИЗ БЕЗОПАСНОСТИ ПРИЛОЖЕНИЙ (DAST)

Динамический анализ безопасности приложений представляет собой процесс выявления уязвимостей в режиме выполнения (run-time) программного обеспечения. Основные методы динамического анализа включают общий сбор путей выполнения (трассировку выполнения), полносистемный анализ и фаззинг-тестирование.

Общий сбор путей выполнения включает проведение сборки исходных текстов, в которые будут интегрированы специальные датчики. Эти датчики отслеживают проходящие через них вызовы функций и осуществляют запись логов в файл, на основе которых впоследствии осуществляется анализ и формируется трасса выполнения. Это позволяет определить, какие блоки кода были вызваны, а какие остались неиспользованными, а также проследить за прохождением по веткам выполнения.

Таким образом, можно выделить следующие этапы трассировки выполнения:

1. Сбор логов по динамическому выполнению проекта.
2. Обнаружение путей ветвления и исполнения проекта на основе реального выполнения.
3. Определение модулей и функций, через которые проходило выполнение.
4. Выявление мест возникновения ошибок и сбоев.

Данный подход в АК-ВС 3 позволяет выявить наиболее используемые участки кода, а при наличии всех возможных сценариев работы программы с помощью данного метода возможно найти «мёртвый код». В результате данный анализ позволяет из всей кодовой базы выделить именно те участки кода, которые требуют тщательного анализа безопасности кода.

ПОЛНОСИСТЕМНЫЙ ДИНАМИЧЕСКИЙ АНАЛИЗ (КОД2)

АК-ВС 3 обеспечивает эффективное выполнение полносистемного динамического анализа благодаря способности записывать и воспроизводить сценарии работы системы в контролируемой виртуальной среде. Это осуществляется за счёт использования специализированной платформы QEMU и возможностей виртуализации, предоставляемых инструментом. Виртуализация позволяет фиксировать состояние системы на любом этапе, что, в свою очередь, упрощает диагностику сложных и скрытых проблем.

Одной из ключевых особенностей АК-ВС 3 является автоматизированное отслеживание и анализ «помеченных данных» и функций, изменяющих их, что позволяет оперативно выявлять источники и пути распространения «чувствительной» информации и проверять корректность обращения с секретами путём поиска их следов в артефактах работы программы — от переменных окружения до оперативной памяти.

АК-ВС 3 также обеспечивает визуальное представление результатов анализа. Например, с помощью таких инструментов, как граф процессов и flame-граф, пользователи могут легко интерпретировать результаты, видеть конкретные места возникновения проблем безопасности и оперативно устранять их. Это значительно сокращает время анализа и повышает эффективность работы специалистов.

Важным преимуществом инструмента является высокая точность и глубина анализа, достигаемая за счёт сочетания динамического подхода с возможностью детального просмо-

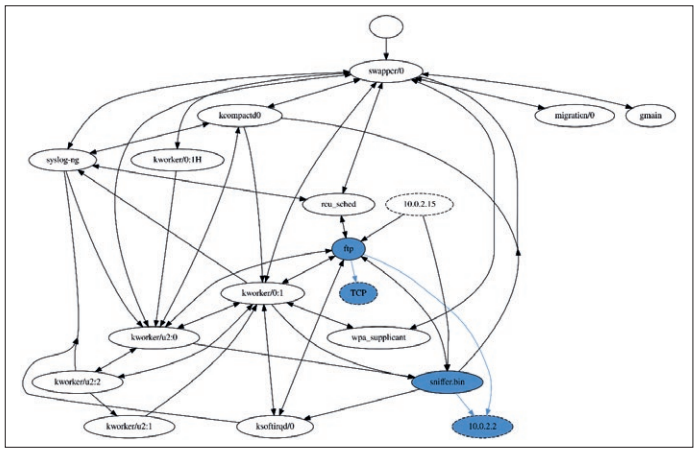


Рисунок 1. Граф, отражающий последовательность вызова функций/программ. Синим отмечены «заражённые» объекты, в которых модифицируются помеченные данные

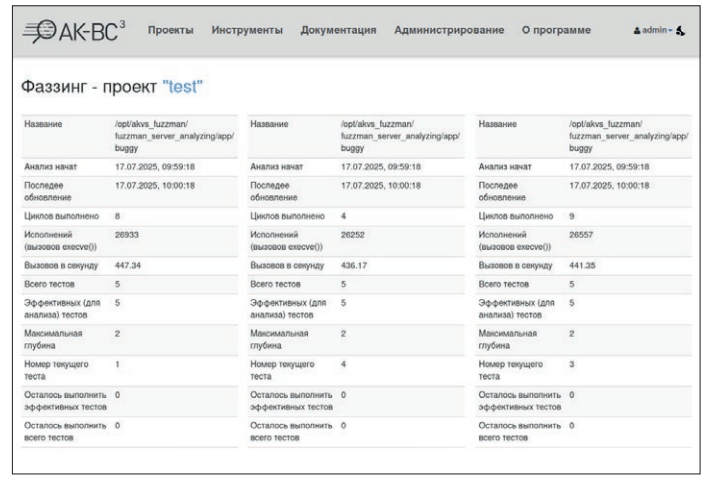


Рисунок 2. Статистика о выполнении фаззинг-тестирования в АК-ВС 3

тра состояния памяти и сети на момент выполнения сценария.

Этапы полносистемного анализа включают:

- полносистемный анализ трассы выполнения кода;
- запись данных всей системы с отслеживанием каждого элемента;
- поиск утечек данных и критической информации.

ФАЗЗИНГ-ТЕСТИРОВАНИЕ

Фаззинг-тестирование представляет собой метод исследования, при котором на вход программы подаются случайные данные. В АК-ВС 3 реализован современный подход фаззинг-тестирования, основанный на использовании обратной связи (feedback-based fuzzing). Данный подход значительно увеличивает качество проводимого тестирования, так как в данном случае инструмент получает информацию о том, как случайные данные влияют на систему. Стоит отметить, что для получения наиболее точной информации о внутреннем состоянии программы данный подход требует доступа к исходному коду программы. Однако также предусмотрена возможность фаззинг-тестирования без статического инструментирования, если доступ к исходным текстам ограничен. Входные данные генерируются с использованием как мутационных, так и генерационных методов. Для повышения качества генерируемых данных возможно использование динамического символьного выполнения — метода, при котором внутренние и внешние данные программы оцениваются как абстрактные символьные переменные, а не как конкретные значения.

АК-ВС 3 содержит в себе все необходимые модули для качественного проведения фаззинг-тестирования:

- различные виды компиляторов для инструментации исходных текстов (при необходимости инструментации);
- модуль фаззера;
- специальные датчики, которые находят ошибки в поведении программы, которые не может выявить оригинальный AFL++;
- инструмент динамического символьного выполнения.

Также необходимо отметить удобство использования веб-интерфейса АК-ВС 3 — одна из тех деталей, которые превращают фаззинг в прозрачный рабочий процесс. Достаточно открыть браузер, выбрать проект, и вы сразу видите живую картину: как меняется интенсивность прогона по времени, сколько найдено уникальных сбоев и другую сводку результатов фаззинг-тестирования (рис. 2).

ЗАКЛЮЧЕНИЕ

АК-ВС 3 — комплексное решение для обеспечения безопасности программного обеспечения, объединяющее современные методы анализа и механизмы соблюдения нормативных требований. Благодаря гибкой архитектуре и интеграции с конвейерами CI/CD продукт легко встраивается в процессы разработки и выступает единой мощной платформой тестирования. На одной площадке можно проводить разные виды проверок и выполнять регулярный анализ безопасности на протяжении всего жизненного цикла ПО. Клиент-серверная архитектура обеспечивает быструю интеграцию, масштабируемость и удобное сопровождение, позволяя централизованно управлять проектами и распределять вычислительную нагрузку между несколькими узлами в инфраструктуре разработчика.

* * *

Для отслеживания последних новостей в сфере информационной безопасности подписывайтесь на наш телеграм-канал Echelon Eyes (*t.me/EchelonEyes*):

ЦИФРОВОЙ РУБЛЬ В ТЕЛЕФОНЕ

ЧТО ГОВОРИТ СТАНДАРТ ЦР ПРО МОБИЛЬНЫЕ ПРИЛОЖЕНИЯ И ПРИ ЧЁМ ТУТ ТРАССА ВЫЗОВОВ?

Юрий ШАБАЛИН
директор продукта AppSec.Sting

Платформа цифрового рубля постепенно превращается из концепции в реальность. Переводы между физическими лицами, оплата в торговых точках, работа кассы мерчанта — все эти сценарии должны быть интуитивно доступны со смартфона в ближайшее время. В этой парадигме мобильное приложение, в которое участники платформы встраивают Программный модуль Банка России (ПМ БР), становится не просто интерфейсом, а ключевым звеном в этой цепочке. Именно поэтому мобильное приложение с интегрированным ПМ БР становится центральным элементом всей архитектуры платформы. Однако эта центральная роль делает мобильное приложение и самым уязвимым звеном. Здесь пересекаются требования пользовательского опыта, криптографической стойкости и регуляторного соответствия. Любое изменение в коде приложения потенциально может повлиять на работу ПМ БР, что автоматически требует дополнительных проверок со стороны надзорных органов.

ЧТО В СТАНДАРТЕ?

Опубликованный «Стандарт Платформы цифрового рубля» от 20 ноября 2024 года (далее — Стандарт) чётко определяет статус мобильного приложения с интегрированным ПМ БР: он относится к средствам криптографической защиты информации. Это означает, что каждое обновление приложения формально требует прохождения процедуры оценки влияния изменений в аккредитованной лаборатории.

Но понимая, что мобильные приложения обновляются значительно чаще традиционного программного обеспечения и изменения в приложениях могут не затрагивать функционал, связанный с цифровым рублём, регулятор предусмотрел компромиссное решение: если в новой версии МП сохранились «неизменность трасс вызовов и порядка вызовов ПМ БР», то повторная оценка влияния не нужна, достаточно предоставить материалы, подтверждающие эту неизменность.

Если посмотреть детальнее, то Стандарт предъявляет к мобильным приложениям два основных требования:

1. Фиксировать и сравнивать трассы вызовов встроенного ПМ БР для каждой новой сборки.
2. Доказывать, что публичная сборка, размещённая в магазине приложений, идентична той, что была проверена в аккредитованной лаборатории.

Стандарт констатирует, что нужно получить (граф вызовов, контрольные суммы), но умалчивает, как этого достичь в реалиях современной мобильной разработки. В этой статье мы последовательно разберём эти вопросы и представим проектные методики, которые закрывают технические и организационные меры, позволяя автоматизировать получение необходимой информации для соблюдения регуляторных норм.

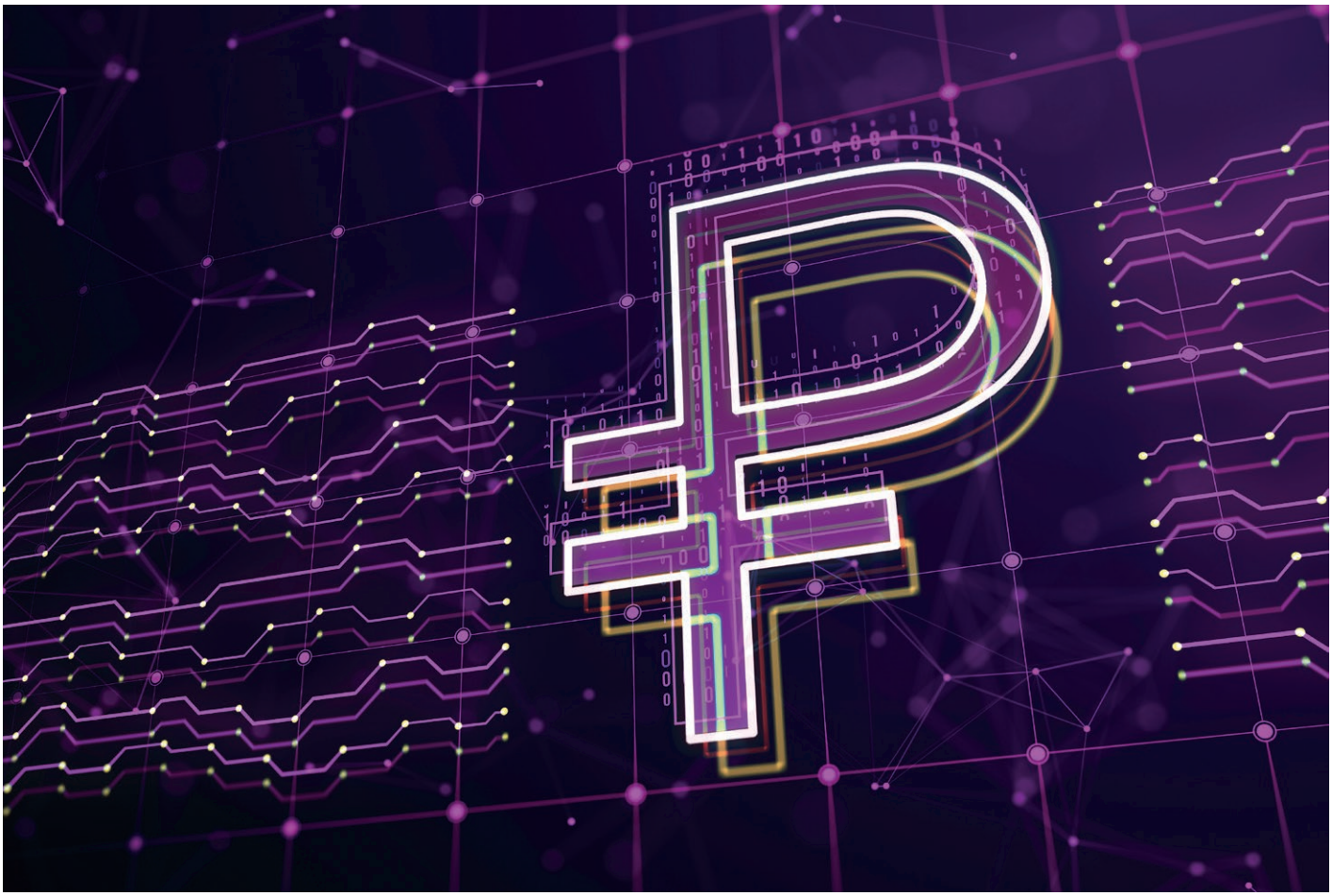
ТРАССЫ ВЫЗОВОВ, ЧТО ЭТО И КАК ПОЛУЧИТЬ?

Итак, для того, чтобы упростить себе жизнь, нам необходимо построить трассу вызовов и при каждом обновлении приложения доказывать её неизменность. Вроде бы ничего сложного, но давайте разберёмся.

Для начала определим, что же такое «Трасса вызовов». По сути, это последовательность инструкций и функций, которые выполняются при обращении к Программному модулю Банка России.

Как же построить эту трассу? Это один из главных вопросов, на который ответа пока что нет, Стандарт прямо требует, чтобы «неизменность трасс вызовов и порядка вызовов ПМ БР в составе МП» подтверждалась при каждом релизе; при этом сами трассы «разрабатываются (согласовываются) совместно с испытательной лабораторией». Формально метод оставлен на усмотрение разработчика и лаборатории, поэтому практика только складывается.

На первый взгляд задача кажется простой: запусти любой популярный сканер, получи граф вызовов и сравни с эталоном. Но основная сложность построения трасс вызовов заключается в том, что современные мобильные приложения представляют собой сложные многослойные системы. Android-приложения могут содержать код одновременно на Kotlin, Java, использовать динамическую загрузку модулей. iOS-приложения сочетают Swift, Objective-C, также могут включать нативные компоненты.



Ну а множество современных фреймворков с «синтаксическим сахаром» делают разработку удобнее, но анализ такого кода сильно затрудняется[1]. Классические инструменты статического анализа очень плохо справляются с задачей построения полного графа вызовов в таких сложных и разнообразных системах. Они могут проанализировать код на одном языке, но теряют связи на границах между языками, не понимают динамически создаваемые вызовы, не всегда поддерживают всё многообразие фреймворков и путаются при использовании в коде «сахара».

Что делать в том случае, когда статические анализаторы не справляются? На мой взгляд, тут на помощь приходит анализ уже скомпилированного кода. После компиляции весь «синтаксический сахар» разворачивается компилятором в стандартные конструкции, Java и Kotlin унифицируется в один формат для корректного восприятия виртуальной машиной Android, все трансграничные вещи между Objective-C и Swift имеют точную и явную связь и отображение в бинарном файле. То есть вместо запутанного, сложного и непредсказуемого кода можно использовать унифицированный и единый формат представления после компиляции.

Таким образом, практическое решение задачи построения трасс вызовов требует комплексного подхода, объединяющего несколько технологий и методик. Процесс методически можно разбить на четыре ключевых этапа, каждый из которых решает определённую техническую задачу.

ЭТАП 1: ПОДГОТОВКА «ЧИСТОЙ» СБОРКИ

Первый шаг — получение приложения в форме, пригодной для анализа. Для Android это означает APK или AAB без обфускации. Для iOS требуется IPA-файл с сохранёнными dSYM-символами, содержащими отладочную информацию. Эти артефакты должны быть подписаны тем же ключом, который будет использоваться для финальной сборки, чтобы исключить возможность подмены.

ЭТАП 2: ПОСТРОЕНИЕ ОРИЕНТИРОВАННОГО ГРАФА

Второй этап заключается в построении полного ориентированного графа вызовов, где каждая вершина представляет метод или функцию, а каждое направленное ребро — возможный вызов. Граф сохраняется в стандартизированном формате (GraphML, Mermaid и других), что обеспечивает совместимость с раз-

[1] Синтаксический сахар в программировании — упрощённые формы записи конструкций языка, которые не меняют логику работы программы, но делают код более читаемым и удобным для написания. Это своего рода «дополнительные возможности» в синтаксисе, которые позволяют выражать те же идеи более лаконично и понятно для разработчика.

личными инструментами анализа и визуализации.

ЭТАП 3: ФИЛЬТРАЦИЯ И ВЫДЕЛЕНИЕ ТРАСС ПМ БР

Критически важным является третий этап — фильтрация полного графа с целью выделения только тех путей, которые связаны с ПМ БР. Из полного графа, содержащего десятки тысяч вершин, необходимо выделить подграфы, описывающие все возможные пути обращения к методам ПМ БР. Алгоритм фильтрации начинает с известных точек входа в ПМ БР и рекурсивно обходит граф в обратном направлении, находя все методы, которые прямо или косвенно могут привести к вызову модуля.

ЭТАП 4: СРАВНЕНИЕ С ЭТАЛОНОМ

Финальный этап — сравнение полученных трасс с эталонными, зафиксированными при первичной экспертизе. Алгоритм сравнения ищет структурные различия: добавленные или удалённые вершины, изменившиеся связи, новые пути вызовов. Любые изменения, затрагивающие цепочки, ведущие к ПМ БР, классифицируются как «критичные» и требуют дополнительного анализа.

В результате мы получаем гарантированно повторяемый и стабильный результат по составлению и контролю трассы вызовов, который подсвечивает любые изменения или новые трассы до методов ПМ БР, что полностью соответствует описанным требованиям в Стандарте.

ДОКАЗАТЕЛЬСТВО ИДЕНТИЧНОСТИ СБОРОК

Но остаётся ещё один вопрос: как доказать, что приложение, которое передаётся на анализ в испытательную лабораторию, соответствует тому, что выкладывается в магазины приложений?

Казалось бы, сравни два файла, посчитай хеш-суммы и убедись в идентичности. Но как оказывается, это совсем не тривиальная задача, так как при сборке даже из одного и того же кода приложение будет иметь разные хеш-суммы по разным причинам. Это и различные создаваемые во время сборки файлы, и уникальные для каждой сборки идентификаторы (рекламные, аналитические и другие). Более того, магазины приложений сами вносят изменения в приложения. Например, Google Play с 2021 года требует загрузки приложений в формате AAB (Android App Bundle), который затем автоматически разделяется на device-специфичные пакеты и переподписывается ключами, сгенерированными на стороне Google. Apple же применяет к приложениям FairPlay-шифрование, которое кардинально изменяет структуру исполняемого файла. В результате контрольная сумма приложения в магазине никогда не совпадает с контрольной суммой исходной сборки. А если разработчик приложения применяет усиленные меры шифрования для своих сборок — это отдельная история.

Дополнительную сложность создаёт обфускация кода. Для анализа в лаборатории требуется «чистая» сборка без обфускации, но финальная версия для публикации в магазинах приложений обязательно проходит процедуры минификации и запутывания кода. И все эти, и другие неупомянутые нюансы делают проверку соответствия между версиями совершенно нетривиальной задачей.

Решение этой проблемы лежит в плоскости организационно-технических мер, сочетающих технические маркеры с процедурными гарантиями. Ключевая идея заключается в том, чтобы вместо контроля каждого байта исполняемого файла фиксировать «отпечаток происхождения» — набор метаданных, который остаётся неизменным на всём пути от исходного кода до финального приложения. Такой подход предполагает автоматическое внедрение в каждую сборку уникальных идентификаторов.

Организационная составляющая решения требует строгого контроля доступа к релизным ключам и переменным сборочного конвейера. Все операции должны выполняться автоматически и только в CI/CD-системе, исключая возможность ручного вмешательства. Перечень лиц, имеющих доступ к критически важным компонентам, должен быть документально зафиксирован и ограничен техническими средствами. Такое решение позволяет владельцу ПО скачать приложение из магазина, извлечь из него метаданные происхождения и сопоставить их с данными из лабораторного протокола. Организационные меры обеспечивают, что эти метаданные не могли быть подделаны благодаря защищённому процессу сборки и ограниченному доступу к ключам подписи.

ЗАКЛЮЧЕНИЕ

Стандарт цифрового рубля ставит перед рынком правильные цели — обеспечить безопасность и контроль над критически важным компонентом, ПМ БР. На данный момент два ключевых требования, контроль неизменности трасс вызовов ПМ БР и доказательство идентичности сборок, сформулированы на концептуальном уровне, далее необходимы конкретные методические рекомендации и технические спецификации.

Индустрия мобильной разработки уже располагает технологиями, методиками и экспертизой, необходимыми для создания эффективных решений данной задачи. Предложенный в статье комплексный подход демонстрирует, как современные DevSecOps-практики могут быть адаптированы для соответствия требованиям финансового регулятора без ущерба для скорости и качества разработки.

Ключевые принципы предложенного решения, автоматизация рутинных операций, стандартизация форматов данных, прозрачность процессов и разделение ответственности между техническими и организационными мерами, превращают регуляторные требования из потенциального увеличения Time2Market в катализатор технологического развития отрасли.

Дальнейшее развитие предложенного подхода требует тесного взаимодействия между тремя ключевыми группами участников: технологическими командами, создающими инструменты и методики, регуляторными органами, формализующими требования и процедуры, и бизнес-сообществом, обеспечивающим реальное внедрение решений в коммерческую практику.

МОБИЛЬНЫЕ ПРИЛОЖЕНИЯ ВЫХОДЯТ НА НОВЫЙ УРОВЕНЬ. НО ХАКЕРЫ ОПЕРЕЖАЮТ

Юрий ШАБАЛИН
директор продукта AppSec.Sting

95% клиентов банков пользуются финтех-услугами посредством мобильных приложений — это статистика крупнейших банков России. При этом 90% мобильных приложений для ФинТех содержат серьезные уязвимости. Такой результат показало исследование мобильных приложений, проведенное платформой AppSec.Sting компании AppSec Solutions, опубликованное в 2025 году. Уязвимостей высокого и критического уровня становится больше, несмотря на активное развитие безопасной разработки. Директор по продукту AppSec.Sting Юрий Шабалин рассказал об основных трендах в безопасности мобильных приложений в 2025 году.

— Звучит как парадокс: мобильные приложения — самый быстроразвивающийся сервис, но и самый уязвимый, почему?

— Мобильные приложения — самый молодой клиентский продукт, в отличие от веба, которому уже больше 40 лет, приложения появились лет 15 назад. Раньше они были просто витринами данных: отображали информацию, поступающую с бэкэнда, и на этом всё — никакой особой логики внутри не было. Сейчас же мобильные приложения представляют собой самостоятельные, сложные системы со своими фреймворками, архитектурой, межпроцессным взаимодействием. Внутри одного приложения может быть очень сложный архитектурный код, со своими проблемами в бизнес-логике — и всё это на стороне клиента, на самом устройстве пользователя. При этом очень немногие понимают, как на самом деле устроен процесс дистрибуции мобильных приложений, то есть каким образом они попадают к пользователю. Немало возможностей хакерам предоставляет использование сторонних сервисов. Например, отправка push-уведомлений, установка точек банкоматов на карте и многое другое.

— Возьмем, например, финтех. В чем основное отличие в области безопасности?

— Если уязвимость обнаружена на веб-сайте, срабатывает традиционная система защиты данных: ИБ-инженеры применяют наложенные средства защиты, Web Application Firewall, чтобы не дать злоумышленникам использовать уязвимость. Это своего рода «пластырь на ране». Затем нужно «обработать рану»: устранением проблемы компания занимается во внутреннем контуре. Инженеры полностью контролируют среду выполнения: это их серверы, они знают, где они находятся, кто к ним имеет доступ, как их обновлять. А в случае с приложением мы не контролируем среду, и даже если удастся выявить и оперативно вылечить уязвимость, придется ждать модерации исправленной версии и выхода обновления приложения.

— Достаточно ли в распоряжении компаний средств для этого?

— В распоряжении коммерческого сектора не так много продуктов, которые позволяют выполнять проверку приложений комплексно. На сегодняшний день, единственная платформа автоматизированного анализа мобильных приложений, которая сочетает несколько видов анализа — это AppSec.Sting.

Ручной анализ мобильного приложения обычно занимает около двух дней, а платформа справляется с этим за два часа. Это автоматизированный и повторяемый процесс, который можно встроить в цикл разработки. После каждой сборки вы получаете отчёт о безопасности и понимаете, какие уязвимости есть в приложении, что именно выходит в релиз. Это значительно ускоряет анализ и сокращает затраты на обеспечение безопасности.

Кроме того, благодаря AppSec.Sting специалист по тестированию на проникновение сможет сосредоточиться на тех проблемах, которые невозможно обнаружить автоматически.

Наш продукт обеспечивает максимальное покрытие за счёт различных практик анализа. Мы смотрим на приложение со всех сторон: анализируем его в статике — исходный код, декомпилированный или восстановленный, — запускаем на устройстве, наблюдаем за его поведением, отслеживаем сетевые запросы, выясняем, куда и какие данные уходят, и как приложение взаимодействует с внешними сервисами, так называемыми third-party.

Платформа проводит и компонентный анализ: определяем, какие библиотеки используются в приложении, и оцениваем их уязвимость. То есть AppSec.Sting позволяет охватить практически все аспекты безопасности.

ИНЖЕНЕРНЫЙ ПОДХОД К БЕЗОПАСНОСТИ В IDECO

АРХИТЕКТУРА, ПРОЦЕССЫ, ИНТЕРФЕЙС

Владимир ИВЧЕНКО
системный архитектор Ideco

Илья СОБОЛЕВ
менеджер по продукту Ideco NGFW

Марк КОРЕНБЕРГ
CTO Ideco

Создание корпоративного NGFW требует не только абстрактного подхода, но и конкретных инженерных решений, учитывающих реалии эксплуатации. В Ideco архитектура проекта строится на кейсах заказчиков, внутренних ценностей и опыте эксплуатации в сложных инфраструктурах. Ниже — обзор принципов, которыми мы руководствуемся в разработке.

ИСХОДНАЯ ТОЧКА — ПОЛЬЗОВАТЕЛЬСКИЕ СЦЕНАРИИ

Технические решения не принимаются вслепую. Помимо собственных компетенций и изучения конкурентных решений, мы регулярно проводим интервью с заказчиками: собираем обратную связь с пилотов, из поддержки, от пресейлов. Анализируем не только «что нужно», но и «как это используется». Это позволяет понять ограничения, с которыми живёт инфраструктура заказчика, и то, как наша архитектура должна адаптироваться к потребностям. Интерфейс, модули, структура, способ реализации — всё строится от задачи, а не от абстрактного идеала. Иногда это означает усложнение архитектуры ради простоты в эксплуатации — но это разумный обмен.

ИЗОЛЯЦИЯ CONTROL PLANE И DATA PLANE — НЕ ОПЦИЯ, А ТРЕБОВАНИЕ

В корпоративной архитектуре сети управления и сети передачи данных жёстко разделены — они имеют разную адресацию и разные политики доступа. Управляющий трафик считается доверенным, а пользовательский — потенциально враждебным. Поэтому изоляция Control Plane и Data Plane — это не вопрос удобства, а требование безопасности.

В Ideco NGFW это реализовано через изолированные контейнеры VCE: управляющая и трафиковая части работают раздельно, с разными ресурсами и изоляцией на уровне сети и процессов. В других решениях это достигается через отдельные ВМ или даже разные платы (как у Check Point).

Control Plane и Data Plane имеют разные архитектурные требования: к языкам, отказоустойчивости, производительности и логике взаимодействия. Такое разделение позволяет сохранить управляемость NGFW даже при атаке на Data Plane — и наоборот. Это критично при восстановлении, отладке или экстренной смене конфигурации.

КОНТЕЙНЕРНАЯ МОДЕЛЬ С VCE: ЭТО ПРО МАСШТАБИРУЕМОСТЬ

Каждый VCE (Virtual Context Engine) — это логически сетевая изолированная среда, фактически это копия NGFW (отдельно запущенные копии всех программ для обработки сети со своей копией сетевого стека). Она используется для разделения политик доступа, зон безопасности. Это не просто удобство, а механизм управления сложной инфраструктурой. VCE позволяет оптимизировать потребление ресурсов: у каждого контекста своя адресация, маршруты, сессии и журналирование. Ранее отчётность хранилась внутри каждого VCE, но это привело к росту потребления памяти. С сентября 2025 года мы централизуем хранение данных в корневом контексте, чтобы уменьшить нагрузку и упростить поддержку. Этот переход стал логичным продолжением пути оптимизации, и мы не исключаем, что в будущем подобная консолидация затронет и другие тяжёлые сервисы.

АТОМАРНОСТЬ КОНФИГУРАЦИЙ КАК СРЕДСТВО ОТКАЗОУСТОЙЧИВОСТИ

Состояние системы не должно зависеть от того, прервалась ли сессия администратора во время внесения изменений, отключилось ли питание. В Ideco NGFW изменение конфигурации либо полностью применяется, либо откатывается. Это принцип архитектуры, а не «удобная фича». Цель — исключить рассогласованность между подсистемами. Атомарность конфигураций позволяет выполнять обновление в автоматическом режиме в течение 5–10 минут. В перспективе рассматриваем переход к транзакционному применению настроек — по аналогии с подходами западных вендоров, когда изменения группируются и применяются только по подтверждению администратора.

OPEN SOURCE — С АДАПТАЦИЕЙ И КОНТРОЛЕМ

Использование внешнего кода — нормальная практика. Но мы не переносим чужие проекты в чистом виде. Библиотеки анализируются, интегрируются как модули, подвергаются фаззингу, покрываются тестами. Некоторые подсистемы (например, Suricata) доработаны вручную. Всё, что попадает в сборку, проходит через наш CI, и мы исключаем неконтролируемые зависимости. Также в разработке применяются внутренние политики: мы не используем.tar.gz из интернета, не доверяем сторонним бинарным репозиториям, не допускаем автоматических зависимостей без ручной ревизии. Такой подход может показаться избыточным, но он позволяет избежать классических проблем supply chain-инцидентов, которые в последние годы затронули десятки вендоров.

Мы не делаем «коробочное решение с галочками». Мы делаем систему, которая не теряет управляемость тогда, когда это критично. И именно поэтому в наших приоритетах — не только функциональность, но и прозрачность, отказоустойчивость и операционная предсказуемость

CI/CD-ПРОЦЕСС: СТРОГАЯ РЕГЛАМЕНТАЦИЯ ВМЕСТО «БЫСТРОГО РЕЛИЗА»

Релизы каждый месяц возможны только при стабильной инфраструктуре. Каждая фича живёт в своей ветке до полной готовности. DroneCI, шаблоны jsonnet, автоматические сборки, автотесты в VK и Yandex Cloud, ручные проверки, статус-контроль версий — всё это выстроено в пайплайн. Мы не делаем «релиз по флагу», а стабилизируем каждый образ перед публикацией. Важной особенностью является то, что вся сборочная инфраструктура — собственная. Мы не полагаемся на SaaS-инструменты и разворачиваем весь процесс внутри облаков, которыми управляем самостоятельно. Это позволяет не только быстрее реагировать на ошибки, но и гарантировать предсказуемость среды сборки.

ИНТЕРФЕЙС КАК ИНСТРУМЕНТ УМЕНЬШЕНИЯ ОШИБОК

Понятный интерфейс снижает порог ошибок. Дашборды по пользователям, контекстное создание объектов, фильтры, drag & drop — это не декоративные элементы. Это способ быстрее диагностировать, кто генерирует аномальный трафик и как на него реагировать. Мы постепенно выносим в UI те настройки, которые раньше были доступны только через терминал. Это не упрощение, а повышение управляемости. Интерфейс становится частью инженерной дисциплины: он не заменяет документацию, но позволяет ускорить принятие решений в условиях инцидента.

ЗАКЛЮЧЕНИЕ

Ideco NGFW — это набор архитектурных решений, принятых для того, чтобы продукт был предсказуемым в эксплуатации, масштабируемым в инфраструктуре и устойчивым к сбоям. Мы не делаем «коробочное решение с галочками». Мы делаем систему, которая не теряет управляемость тогда, когда это критично. И именно поэтому в наших приоритетах — не только функциональность, но и прозрачность, отказоустойчивость и операционная предсказуемость. Это не лозунги, а результат конкретных инженерных решений, о которых мы честно рассказываем.