BIS JOURNAL

ИНФОРМАЦИОННАЯ БЕЗОПАСНОСТЬ БИЗНЕСА

№2 (61) 2026

Герман КЛИМЕНКО
Фонд развития цифровой экономики
«Цифра – она как кукушонок!» → 39

Юрий ШАБАЛИН
Swordfish Security
Как защитить ИИ-решения? → 38

ГосСОПКА — всё под контролем!
→ 5

Защита данных. В эпоху Zero Trust → 46

РБПО. Триаж секретов → 78

Алексей МАРТЫНЦЕВ
Норильский никель
«След атаки ведет в ловушку» → 5

Сергей ЛЕБЕДЬ
Сбер
«Сбер плюс ГосСОПКА – удар по фишингу!» → 6

Роман ШАПИРО
АО «Почта России»
«Не ждите проверок – инициируйте диалог» → 8

КАК ОРГАНИЗОВАТЬ ЭФФЕКТИВНЫЙ ПРОЦЕСС БЕЗОПАСНОЙ РАЗРАБОТКИ С ПОДРЯДЧИКОМ

Александр ГАДАЙ
руководитель службы консалтинга Swordfish Security

IT-отрасль стремительно развивается, и, чтобы успеть за её темпами, частные компании и госсектор часто прибегают к услугам внешних подрядчиков для разработки программного обеспечения. Однако эта практика приводит к росту числа атак на инфраструктуру организаций через доступ, предоставляемый исполнителям, и к появлению уязвимостей в ПО. Вы можете заключить с контрагентом строгий NDA (соглашение о неразглашении информации) и понадеяться на его благонадёжность, но это не спасёт вас от рисков компрометации его системы или рабочих станций. С таким доступом злоумышленники способны проникнуть внутрь сетевого периметра вашей компании или подменить объект поставки: добавить закладку в исходный код или зависимый компонент ПО.

В этой статье мы рассмотрим вариант архитектуры процессов безопасной разработки ПО, который позволяет сохранять высокую скорость создания приложений без доступа подрядчика к инфраструктуре.

В первую очередь необходимо выделить несколько сетевых контуров в порядке повышения их значимости:

1. **Контур подрядчика** – внешний сегмент разработки в инфраструктуре компании-исполнителя.
2. **Контур вноса** – демилитаризованная зона (DMZ), в которую загружаются результаты работы подрядчика.
3. **Контур конвейера безопасной разработки** – изолированный контур, в котором находятся средства проверки защищённости цифрового продукта.
4. **Продуктивный контур** – изолированная зона, в которой размещаются релизные версии ПО.

Чтобы свести к минимуму риски компрометации, необходимо ограничить сетевой доступ между контурами на уровне правил межсетевого экранирования. Должна быть запрещена инициация сетевых соединений из менее безопасных контуров к более безопасным, то есть доставка ПО должна осуществляться в итерационной последовательности, не «перескакивая» через уровни (рис. 1):

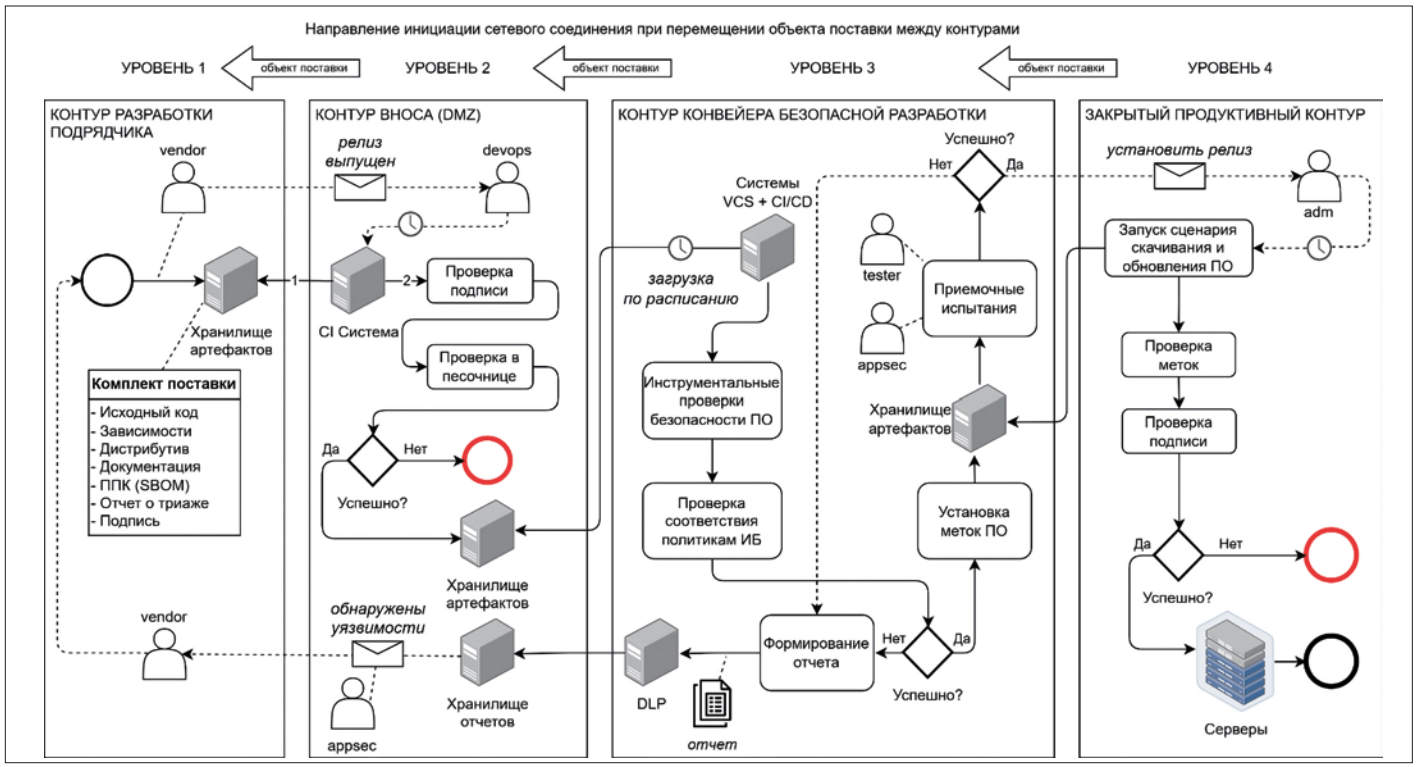


Рисунок 1. Должна быть запрещена инициация сетевых соединений из менее безопасных контуров к более безопасным, то есть доставка ПО должна осуществляться в итерационной последовательности

◆ на второй уровень через обращение к первому;

◆ на третий уровень через обращение ко второму;

◆ на четвертый уровень через обращение к третьему.

Рассмотрим ключевые этапы процесса доставки, начиная с момента готовности подрядчика провести отгрузку ПО. Представим, что объект поставки — это ZIP-архив, содержащий документацию, исходный код, зависимые компоненты приложения и собранный дистрибутив. С точки зрения процесса безопасной разработки, в него важно включить:

◆ ППК (перечень программных компонент) — специальный файл с описанием элементов поставляемого ПО, который позволяет на ранних этапах оценить безопасность зависимых компонент;

◆ Отчёт о триаже — размеченные результаты тестирования SAST, SCA, DAST и других ИБ-проверок;

◆ Цифровая подпись — файл подписи архива, который обеспечивает защиту от подмены при передаче.

Отдельно следует отметить практику предоставления отчёта о триаже в составе поставки. Наличие ложноположительных срабатываний в инструментах неизбежно, поэтому лучше сообщить о результатах вашего анализа уязвимостей до того, как они вернутся в виде замечаний от принимающей стороны. Это позволит значительно сократить время приёмки программного обеспечения и продемонстрирует ответственный подход подрядчика к безопасности собственного ПО.

Подрядчик уведомляет заказчика о готовности релиза, после чего специалист со стороны клиента **запускает процесс вноса дистрибутива** в контуре DMZ. В этот момент выполняется проверка подписи архива поставки и его отправка на сканирование решением класса «песочница» (sandbox). Если подпись верна и поведенческий анализ подтверждает чистоту архива, он перемещается в специальное файловое хранилище. В ином случае процесс завершается.

Рассмотрим контур конвейера безопасной разработки. Он включает следующие ключевые элементы:

◆ хранилище артефактов (например, Nexus Repository);

◆ система хранения кода, сборки и доставки ПО (например, GitLab);

◆ инструменты DevSecOps: анализаторы SAST, SCA, DAST и другие (подробнее об этих решениях можно прочитать в статье «Построение контура разработки безопасного ПО» в выпуске BIS Journal № 1 2025).

Процесс работы в контуре безопасной разработки начинается с получения объекта поставки из файлового хранилища в контуре DMZ. Это реализуется за счёт автоматической проверки наличия новых архивов в заданном каталоге, выполняемой по расписанию.

После этого запускаются проверки инструментами DevSecOps с целью подтверждения отсутствия недекларированных возможностей (НДВ) и вредоносных зависимостей. На данном этапе AppSec-инженер рассматривает отчёты о триаже подрядчика и переносит разметку о ложноположительных срабатываниях в проведённые сканирования. Если анализаторы подрядчика и заказчика совпадают, этот процесс можно автоматизировать, однако участие специалиста всё равно является обязательным для предотвращения случаев некорректной разметки.

По каждой из применяемых практик DevSecOps проводится оценка прохождения политик ИБ. Если они не пройдены, отчёт с обнаруженными уязвимостями возвращается в файловое хранилище контура вноса. Для предотвращения утечки конфиденциальной информации рекомендуется встроить в этот процесс DLP-систему. После исправления найденных уязвимостей цикл повторяется.

Когда проверки безопасности пройдены, артефакт подписывается и публикуется в хранилище, а также к нему добавляются специальные метки, например, SAST_SUCCESS. Доверие к этим меткам обеспечивается за счёт разграничения ролей в хранилище артефактов: установить метку может только специально выделенная техническая учётная запись, работающая в рамках пайплайна.

После того как мы убедились, что объект поставки проходит политики ИБ, проводятся приёмочные испытания. Они включают как функциональное, так и ручное тестирование безопасности с целью выявления, например, уязвимостей бизнес-логики ПО. В случае успеха направляется уведомление администратору по эксплуатации продуктивного контура о необходимости установки релиза.

Продуктивный контур содержит самые ценные активы: данные и серверные мощности, на которых исполняется ваше приложение. Процесс получения новых релизов должен инициироваться со стороны продуктивного контура следующим образом:

1. Администратор запускает сценарий обновления на новую версию ПО.
2. Скрипт обращается к хранилищу артефактов ПО, расположенному в контуре конвейера, для получения дистрибутива.
3. Перед загрузкой дистрибутива производится проверка скачиваемого артефакта на наличие меток об успешных ИБ-проверках. Если меток нет, процесс завершается.
4. После загрузки дистрибутива выполняется проверка его подписи. Если проверка не проходит, процедура прерывается.

Таким образом, в продуктивный контур попадают только те приложения, которые успешно прошли ИБ-сканирования и имеют корректную подпись.

* * *

Мы описали общий подход, который позволяет обеспечивать эффективное выполнение процессов безопасной разработки с внешним подрядчиком. Однако опыт реализованных нами проектов показывает, что в каждом случае нужно подстраиваться под имеющуюся инфраструктуру и организационные процессы компании, поэтому данный материал следует воспринимать как ориентир при проектировании архитектуры и процессов.

СОВРЕМЕННАЯ ЗАЩИТА ЦЕПОЧКИ ПОСТАВОК

И ПРИ ЧЁМ ЗДЕСЬ ХРАНИЛИЩЕ АРТЕФАКТОВ РАЗРАБОТКИ?

Дмитрий КЛЮЧНИКОВ
руководитель разработки продукта CodeScoring. Save

Когда команда, которая занимается безопасностью кода, начинает строить собственное хранилище артефактов, предсказуем вопрос: а зачем еще одно? Ведь на рынке уже есть Sonatype Nexus, JFrog Artifactory, Harbor, встроенные хранилища платформ разработки, а может быть и еще кто-то. На первый взгляд — выбор богатый. Но если копнуть глубже, становится понятно: выбор между хранилищами — это не выбор между наборами функций. Это выбор философии. И принципы безопасности занимают здесь не последнее место. Мы как разработчик хранилища артефактов хотим поделиться своим видением современного мира хранения компонентов (артефактов) программного обеспечения.

Современная разработка — это не только код. Это контейнерные образы, пакеты библиотек, Helm-чарты, ML и AI-модели, скомпилированные бинарные файлы. Всё, что возникает в результате сборки, тестирования и подготовки к развертыванию, принято называть артефактами. И у каждого из них должно быть место, где он хранится, версионируется, раздаётся потребителям и, к сожалению, пока только в идеале, проверяется на безопасность. Это место — **хранилище артефактов**, или менеджер репозиториев.

Долгое время выбор хранилища был вопросом сугубо инженерным: скорость работы, поддерживаемые форматы, удобство API, настройка проксирования внешних репозиториев. Но чем глубже артефакты проникают в цепочки поставки ПО, тем острее становится другой вопрос — вопрос безопасности. Масштаб хорошо иллюстрируют цифры: в 2025 году нашей командой было зарегистрировано 457 тыс. вредоносных библиотек, в 11 раз больше, чем годом ранее. Зафиксировано также 14 тыс. новых уязвимостей в открытых компонентах. При этом старые проблемы никуда не уходят, 15 тыс. пакетов экосистемы Java продолжают зависеть от уязвимой версии пресловутого log4j (уязвимость — Log4Shell). Логичным выглядит исследование Cloudsmith Artifact Management Survey 2025, которое показывает, что 56% инженерных команд уже считает главной задачей хранилища артефактов именно защиту цепочки поставки, а не удобство хранения.

Можно ли быть уверенным в том, что артефакт, который прямо сейчас разворачивается в продуктивной среде, не несёт в себе уязвимость, вредоносный код или нелицензионный компонент? На этот вопрос помогает ответить SBOM (Software Bill of Materials) — перечень компонентов программного обеспечения, который позволяет понять, из чего именно собран конкретный артефакт и отследить его цепочку зависимостей и «происхождение». Практики формирования SBOM закрепляются по всему миру, от зарубежных стандартов до ГОСТ Р 56939–2024 «Защита информации. Безопасная разработка. Общие требования». И именно этот сдвиг от вопроса «что хранить» к вопросу «чему доверять» сделал тему хранения артефактов по-настоящему актуальной для всех, кто занимается безопасностью разработки.

Если разложить существующие решения, получится не каталог технологий, а история эволюции идей. Каждое поколение хранилищ отвечало на вызовы своего времени — и каждое несло в себе компромиссы, ставшие очевидными лишь спустя годы.

КЛАССИКА ХРАНЕНИЯ АРТЕФАКТОВ

Sonatype Nexus и JFrog Artifactory — ветераны и де-факто стандарты. Именно они сформировали само понятие «универсального менеджера репозиториев»: поддержка десятков экосистем, многозадачные комбинации хранения данных, развитые механизмы репликации и масштабирования. Заслуженная зрелость и большая инсталляционная база. Для организаций, которым важна максимальная широта поддержки форматов и годами выстроенная экосистема интеграций и плагинов, эти решения по-прежнему остаются обоснованным выбором.

Но зрелость имеет обратную сторону. Обе системы — тяжёлые монолиты с высокими требованиями к ресурсам. Критичные для промышлен-

ной эксплуатации возможности — HA (высокая доступность), расширенное хранение, security-функционал — закрыты за дорогими лицензиями. Кластеризация превращается в отдельный инфраструктурный проект. Обновления болезненны: меняется схема БД, формат метаданных, модель плагинов. А вот уязвимости находятся часто, что создаёт неприятную вилку: обновляться — больно, не обновляться — страшно. Архитектурные решения, принятые пятнадцать лет назад, становятся не только устаревшими (легаси), но и ограничением для тех, кто хочет строить современные процессы.

ПРОСТЫЕ ХРАНИЛИЩА АРТЕФАКТОВ

Следующая волна решений взяла на флаг идею «чем проще — тем лучше». Harbor, проект CNCF, ограничился OCI-совместимыми артефактами и предложил разумный баланс между простотой и функциональностью: kube-native архитектура, интегрированные open source сканеры образов контейнеров, механизм проверки подписей. Для cloud-native команды, живущей контейнерами — то, что нужно. Harbor — отличный пример того, как фокус на конкретном формате позволяет довести пользовательский опыт до уровня, недоступного универсальным решениям.

Но если в стеке есть Maven, npm, PyPI или что-то ещё — нужно или второе хранилище, или набор упражнений для перестройки в формат OCI всех поступающих артефактов. На больших объемах неизбежно приходится бороться со сборкой мусора (garbage collection), а мажорные обновления требуют почти ручной миграции без волшебной кнопки «перенести». Это был справедливый компромисс для своего времени, но всё же компромисс.

ХРАНИЛИЩА АРТЕФАКТОВ НА ПЛАТФОРМАХ РАЗРАБОТКИ

Параллельно развивался другой подход: артефакты как бонус к платформе разработки. Встроенные хранилища GitLab, GitHub, Azure DevOps и ряда других платформ предложили единый пользовательский интерфейс (UI), общие права доступа, нативную (по умолчанию) интеграцию с CI — всё в одном месте. Главная сила подобных решений — околонулевая стоимость входа, ведь команда уже живет в этой экосистеме. На российском рынке, где потребность в быстром импортозамещении в какой-то момент стала критической, хостовые экосистемные решения упали в благодатную почву: они закрывали сразу множество задач одним инструментом. Хорошим примером можно назвать решения от GitFlic или Сфера.

Но платформенная привязка — палка о двух концах. Интеграция с внешними инструментами ограничена. Поддержка пакетных менеджеров часто реализована на уровне «минимально достаточного», с огрублениями протоколов. Data layer (уровень данных) привязан к самой платформе, масштабирование — только вместе со всем сервером, обновление реджистри отдельно невозможно. По сути, вы не выбираете хранилище — вы принимаете то, что даёт платформа.

НЕКАНОНИЧНЫЕ РЕАЛИЗАЦИИ

Наконец, есть категория решений, предлагающих принципиально иную логику. Проекты вроде Dragonfly (CNCF) переосмысляют саму модель распространения артефактов: P2P-дистрибуция образов поверх основного хранилища по аналогии с торрент-сетями. Ценностное предложение для бизнеса убедительно: радикальное снижение трафика, разгрузка центрального хранилища, эффективная раздача больших образов на сотни нод. Но это меняет всю экономику и модель доверия: образ приходит не из реджистри, а от «соседа» по кластеру, отладка запросов, гуляющих по пирам, нетривиальна, а порог входа для команды высок. Существуют и другие нишевые решения — Pulp для сценариев зеркалирования в изолированных средах, Zot Registry как минималистичное OCI-хранилище с фокусом на безопасность. Каждое точно решает свою задачу, но ни одно не претендует на универсальность.

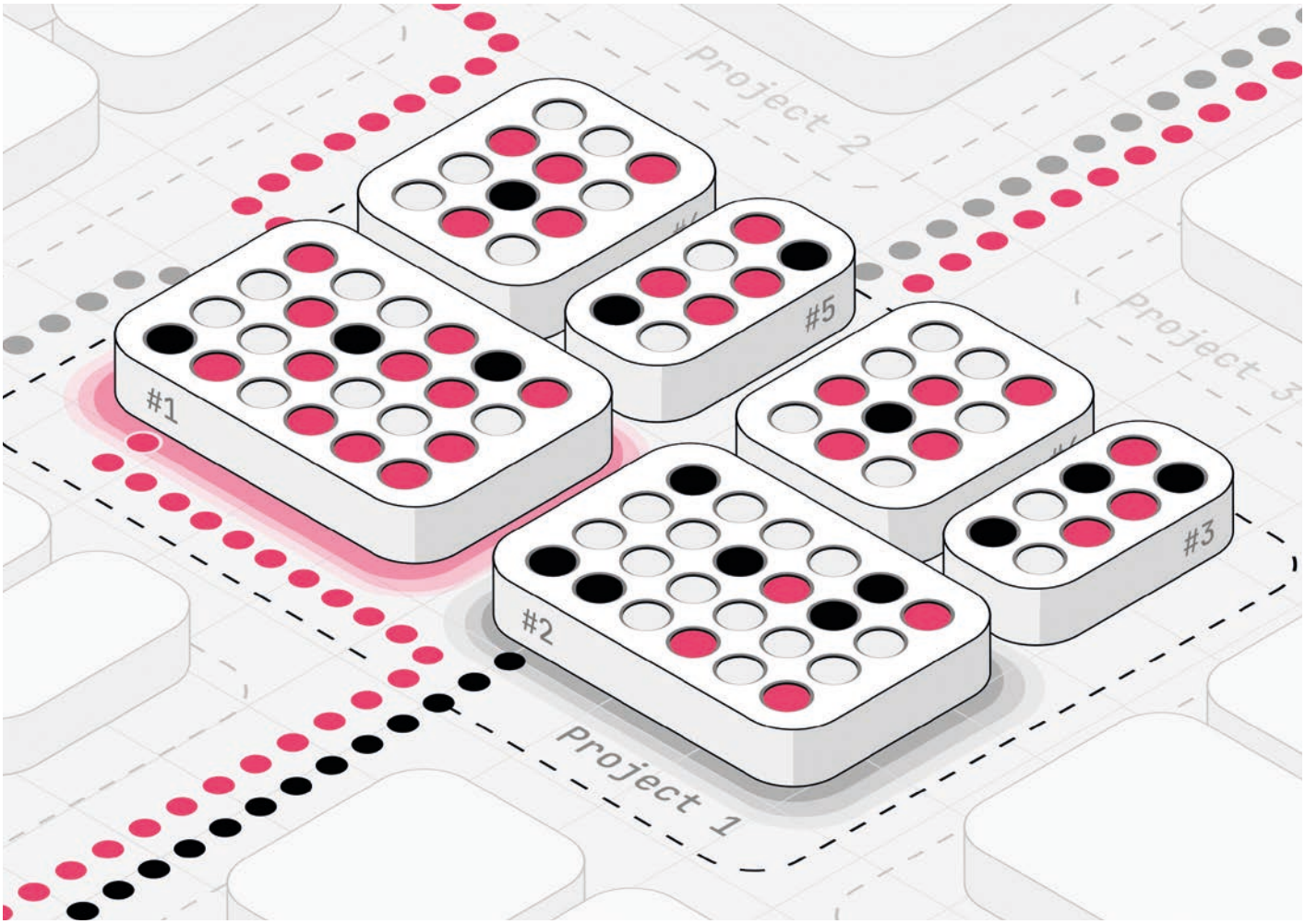
НАЗРЕВШАЯ ПОТРЕБНОСТЬ В ЭВОЛЮЦИИ ХРАНИЛИЩ АРТЕФАКТОВ

Мы выстроили эту классификацию не только из академического интереса. Она помогает увидеть не случайный набор конкурирующих инструментов, а эволюцию идей. Ключевой вывод не в том, что какое-то решение лучше или хуже: каждый класс хорош в своем контексте. Что-то для крупных инсталляций в устоявшихся компаниях с разношёрстным стеком, что-то для cloud-native команд, что-то для ценителей единой среды, а что-то для новаторов и задач, где, например, решение классической задачи транспортировки компонентов упрётся в потолок. Проблема не в инструментах и их выборе, а в том, что ландшафт изменился и появился контекст, для которого ни одно из имеющихся решений изначально не проектировалось.

Для компаний, которые занимаются безопасной разработкой, артефакт — это не просто бинарный файл, который нужно положить на полку и отдать по запросу. Это объект, у которого есть происхождение, зависимости, уязвимости, лицензионная история, SBOM. И хранилище должно это **учитывать нативно**, а не через внешние надстройки.

И вот мы подходим к тому, что стало для нас в CodeScoring отправной точкой. Главный вопрос к хранилищу артефактов сегодня — это давно не «можешь ли ты хранить контейнер или ML-модель?».

Главный вопрос звучит иначе: «можешь ли ты доказать, что этот артефакт безопасен?». Существующие решения отвечают на него по-разному. Nexus подключает отдельный IQ Server, JFrog — платформу XRay, Harbor — встроенные OSS-сканеры. Платформенные решения делегируют это внешним инструментам. Кто-то закрывает вопрос широко, кто-то точечно, но почти никто — оптимально. Безопасность в них «сбоку»: отдельный сервис, отдельная интеграция, отдельная лицензия. Хранилище и безопасность существуют параллельно, а не как единое целое. Говоря о современных реалиях, нельзя не упомянуть вектор угроз, напрямую



связанный с хранением артефактов. Исследование Техасского университета в Сан-Антонио (и нескольких участников из других американских университетов), в котором проводился анализ 576 000 фрагментов кода от 16 популярных LLM, показало, что в среднем около 20% рекомендуемых моделями пакетов не существует в реальных репозиториях. При этом 58% таких «выдуманных» пакетов стабильно воспроизводятся при повторных запросах. То есть это не случайный сбой, а устойчивый паттерн. Именно эта устойчивость открывает дорогу для slopsquatting-атак: злоумышленник заранее регистрирует пакет с именем, которое LLM будет рекомендовать снова и снова, превращая галлюцинацию в точку входа.

Именно этот контекст определил архитектурные решения, которые мы заложили в наше хранилище артефактов CodeScoring.Save. Не как попытку построить лучшую замену всему, а как набор требований сегодняшнего российского рынка.

Современный технологический стек и инженерные практики диктуют вполне конкретные ожидания к, в первую очередь, инфраструктурному ПО: контейнерная среда, горизонтальное масштабирование, разделение compute и storage, API-first подход к управлению. В современных хранилищах артефактов есть вещи, которые нужны на этапе проектирования, например: разработка на Go, kube-native архитектура с HA доступна сразу, модульная SOA-структура, отделяемые компоненты хранения, с реляционной базой данных и S3-хранилищем. Это не уникальные фичи, это отражение того, как строится инфраструктурное ПО сегодня, когда ему не нужно тащить на себе пятнадцатилетнее легаси. Поддержка популярных пакетных менеджеров, улучшенные возможности контроля и аудита — следствие того, что продукт пишется командой, для которой безопасная разработка — не надстройка, а исходная точка.

Отдельно стоит сказать о локальной специфике. Ни одно из существующих зарубежных решений не учитывает требования российских стандартов безопасной разработки — в частности, ГОСТ Р 56939–2024 и связанных с ним практик РБПО. Для команд, которым важно соответствие этим требованиям, возможности зарубежных хранилищ заканчиваются там, где начинается отечественная нормативная база.

Важно подчеркнуть: все предыдущие решения не «плохие». Каждое из них решало задачи своего времени, и многие продолжают успешно это делать по сей день. Но ландшафт изменился. Артефакт перестал быть просто файлом, а хранилище — просто полкой. И мы считаем, что пришло время для продукта, который проектировался именно для этой реальности, а не адаптировался к ней.

ТРИАЖ СЕКРЕТОВ

С ПРИМЕНЕНИЕМ БОЛЬШИХ ЯЗЫКОВЫХ МОДЕЛЕЙ В ПРОЦЕССАХ РБПО ПО ГОСТ Р 56939-2024

Александр ДЕМИДОВИЧ
ведущий эксперт Центра оценки соответствия и тестирования АО «НПО «Эшелон»

Утечки чувствительных данных остаются одной из наиболее опасных проблем современной разработки ПО, что подчёркивается включением отдельного процесса по выявлению и управлению секретами ГОСТ Р 56939–2024 (п. 5.15 «Обеспечение безопасности используемых секретов»). По мере роста числа интеграций, использования облачных сервисов и автоматизированных конвейеров секреты всё чаще появляются в исходном коде, конфигурационных файлах, сценариях сборки, логах и артефактах процессов непрерывной интеграции и непрерывного развёртывания (CI/CD). Основная проблема кроется в том, что секрет, единожды попав в систему управления версиями, навсегда там остается, в чем можно убедиться, просканировав историю изменений и обнаружив секреты даже из удаленных файлов. Именно поэтому такие утечки часто становятся входными точками для атак. Получив действующий токен или ключ, злоумышленник может обойти часть защитных механизмов, получить доступ к внутренним сервисам, облачной инфраструктуре, базам данных или сторонним API. В отличие от многих других уязвимостей, здесь не требуется сложная эксплуатация, поскольку сам факт раскрытия секрета уже создаёт условия для компрометации.

ПОИСК СЕКРЕТОВ И ОГРАНИЧЕНИЯ КЛАССИЧЕСКИХ СКАНЕРОВ

Для поиска секретов используются инструменты, основанные на поиске по шаблонам, регулярным выражениям. Подобный поиск присутствует и в статических анализаторах кода, которые могут выявлять жестко заданные значения. К числу распространённых решений по поиску секретов относятся такие инструменты как Gitleaks, TruffleHog, Detect-Secrets, GitGuardian, AWS git-secrets, Talisman, SpectralOps, а также встроенные механизмы GitHub Advanced Security.

Однако стандартные сканеры секретов почти не учитывают контекст использования найденной строки. Они хорошо выявляют подозрительные последовательности символов, но нередко помечают как секреты тестовые значения или случайные идентификаторы, не являющиеся секретами. Например, библиотека с открытым исходным кодом Gitleaks берёт за основу шаблоны и анализирует прежде всего внешний вид строки, а не её роль в программе или окружение, в котором она используется. В результате такой тривиальный поиск идентифицирует в качестве секретов случайный численно-буквенный набор символов или тестовые значения. Необходимость ручной разметки результатов, которые заведомо содержат большое количество ложноположительных срабатываний, значительно замедляет процессы разработки ПО.

ПРАКТИЧЕСКАЯ ОСОБЕННОСТЬ ТРИАЖА

Разметка и приоритизация результатов сканирования — это ключевой этап работы с секретами. Когда утилита находит десятки или сотни потенциальных секретов, необходимо понять, какие из них являются реальными секретами, а какие представляют собой ложные срабатывания. В контексте практик разработки безопасного ПО (РБПО) задача разметки результатов особенно важна, поскольку связанная с временными и ресурсными трудозатратами команды разработки. Поэтому после первичного поиска требуется дополнительная классификация результатов по степени риска, типу секрета, его актуальности и потенциальным последствиям компрометации. На практике при такой оценке учитываются не только формат найденного значения, но и его жизненный цикл, уровень привилегий, место обнаружения и связь с производственной средой. Например, истёкший тестовый ключ и активный привилегированный токен от продуктивного сервиса формально относятся к одному классу находок, но их реальная опасность принципиально различается.

ПРИМЕНЕНИЕ БОЛЬШИХ ЯЗЫКОВЫХ МОДЕЛЕЙ (LLM) ДЛЯ КОНТЕКСТНОЙ КЛАССИФИКАЦИИ СЕКРЕТОВ

На этом этапе особенно полезны большие языковые модели, способные анализировать не только саму строку, но и окружающий её контекст. В отличие от классических механизмов LLM позволяют перейти от вопроса «Похоже ли это на секрет?» к вопросу «Может ли найденное значение представлять угрозу, к какому типу секрета оно относится и как срочно необходимо отреагировать?» Такой подход даёт возможность использовать модель как слой интеллектуальной разметки поверх классических детекторов.

Алгоритм работы выглядит следующим образом: сканер первично формирует набор потенциальных секретов с сопутствующими метаданными, после чего результаты поступают в модуль LLM, где выполняется классификация. На практике в модель передаётся не только сама найденная строка, но и окружающий её контекст: фрагмент кода, имена пе-

Рисунок 1. Схема конвейера

ременных, путь к файлу и прочие метаданные. На этой основе система определяет, является ли найденное значение секретом, к какой категории оно может быть отнесено и какой уровень риска следует ему присвоить. В результате LLM используется не только как средство дополнительной проверки, но и как инструмент разметки, в которой решающую роль играют контекст, расположение файла, история изменений и признаки связи с продуктивной средой.

Архитектура системы разметки и приоритизации секретов на основе LLM представляет собой многоэтапный конвейер (рис. 1).

В модуле классификации каждая находка обрабатывается как отдельный объект анализа. Перед передачей данных в модель выполняется сокрытие содержимого секрета, что позволяет снизить риск утечки чувствительной информации при обращении к модели. После этого формируется фрагмент окружающего кода, имя файла, сведения о ветке, коммите и иные признаки контекста (рис. 2). На основе этих данных модели ставится задача определить, является ли найденное значение реальным секретом, к какому типу оно относится, каков уровень риска его компрометации и какие действия являются предпочтительными. Результат такой обработки может быть направлен либо в специализированный интерфейс анализа, либо непосредственно в систему управления инцидентами, средства мониторинга безопасности или корпоративную систему постановки задач. При этом аналитик получает не только итоговую категорию и уровень риска, но и краткое пояснение, позволяющее понять логику классификации.

```
{
  "secret_id": "finding-00125",
  "repository": "payments-service",
  "branch": "main",
  "commit": "a7f3c91",
  "file_path": "src/config/auth.js",
  "line_start": 42,
  "line_end": 42,
  "detector": "gitleaks",
  "secret_type_candidate": "aws_access_key",
  "masked_value": "AKIA*******7H2Q",
  "code_fragment": "const accessKey = \"AKIA*******7H2Q\";",
  "author": "ivan.petrov",
  "commit_date": "2026-03-20T10:14:00Z",
  "environment_candidate": "production",
  "is_public_repo": true,
  "days_exposed": 8,
  "context_tags": [
    "cloud",
    "credentials",
    "production-config"
  ]
}
```

Рисунок 2. Контекст найденного секрета

АВТОМАТИЗИРОВАННОЕ РЕАГИРОВАНИЕ И СВЯЗЬ С ПРОЦЕССАМИ РБПО

Следующим уровнем зрелости является автоматизация ответных действий. Если система обладает достаточной уверенностью в том, что обнаруженное значение представляет собой актуальный секрет, возможно автома-

тическое уведомление ответственных разработчиков ПО, запись инцидента в баг-трекер, открытие задачи на устранение или инициирование процедуры отзыва и замены ключа через систему управления секретами. Одновременно с этим должны сохраняться все необходимые артефакты, фиксирующие, какой найденный секрет был обработан, когда это произошло и каким компонентом системы было принято соответствующее решение. Практически эффективной может оказаться комбинированная схема, в которой один инструмент используется для быстрого локального контроля перед коммитом, а более глубокий анализ выполняется в централизованном режиме в процессе непрерывной интеграции с привлечением LLM-классификатора.

При этом не стоит забывать, что применение LLM в задачах обработки секретов требует соблюдения дополнительных мер защиты. Передача чувствительных данных во внешние модели без специальной защитной прослойки недопустима. Реальные ключи и токены должны удаляться из входных данных или заменяться нейтральными обозначениями ещё до обращения к модели. Во многих случаях политика безопасности организации может требовать, чтобы языковая модель разворачивалась исключительно во внутреннем контуре компании либо на изолированном вычислительном узле, а журналы запросов шифровались, ограничивались по сроку хранения и очищались после завершения анализа. Более безопасным считается подход, при котором модели передаются не сами секреты, а только их признаки и контекстные характеристики, например тип токена, уровень привилегий, место обнаружения и связь с производственной средой. При этом фактические значения секретов должны храниться и использоваться только в специализированном защищённом хранилище во время исполнения, не попадая в текстовые запросы к модели. Такой подход снижает вероятность как утечки данных, так и их непреднамеренного воспроизведения моделью.

ЗАКЛЮЧЕНИЕ

В итоге данный подход целесообразно рассматривать как часть системы контролируемых проверок и реагирования в жизненном цикле ПО. При корректной интеграции в процессы РБПО LLM-классификатор может повысить качество анализа результатов сканирования и сократить объём ручной работы. Однако важно отметить, что окончательное решение должен принимать ответственный специалист, поскольку часть результатов неизбежно будет относиться к неоднозначным и требующим дополнительной проверки. Именно поэтому на практике наибольшую устойчивость демонстрирует подход с участием человека в контуре принятия решений, при котором модель ускоряет обработку и снижает нагрузку на экспертов, но не исключает профессиональной верификации там, где цена ошибки особенно высока.

ПРИНЦИП GIGO

ЗАЧЕМ СОВРЕМЕННОЙ РАЗРАБОТКЕ ТОЧНЫЙ SAST

Антон МИХАЙЛОВ
владелец группы продуктов SASTAV

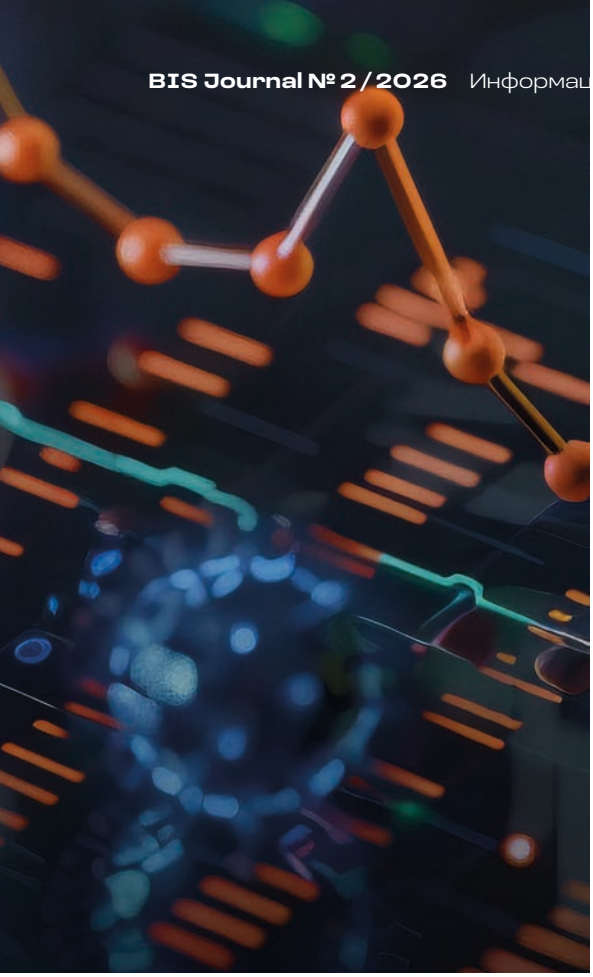
Дмитрий ПОЛИВАНОВ
архитектор отдела технических решений АО «ДиалогНаука»

Внедрение комплексных платформ управления безопасностью приложений (Application Security Posture Management, ASPM) набирает обороты: бизнесу нужна единая, управляемая картина рисков по всем продуктам и командам. Но любая платформа управления стоит ровно столько, сколько стоит качество данных на входе. Принцип Garbage In, Garbage Out (GIGO, «мусор на входе — мусор на выходе») здесь работает буквально: некорректные или «зашумлённые» результаты от инструментов анализа обесценивают даже сильную оркестрацию, корреляцию и отчётность. Поэтому статический анализ безопасности приложений (SAST) — не просто ещё один источник событий, а фундамент всей экосистемы AppSec.

СИСТЕМНЫЙ КРИЗИС ЛОЖНЫХ СРАБАТЫВАНИЙ: ПОЧЕМУ ТОЧНОСТЬ SAST ОПРЕДЕЛЯЕТ УСПЕХ DEVSECOPS

Исторически главный недостаток многих SAST-инструментов, особенно тех, что построены исключительно на сигнатурном анализе,— высокий процент ложных срабатываний (false positives). Это проблема не только операционная, но и экономическая: специалисты по безопасности и разработчики тратят существенную долю времени на верификацию предупреждений, что замедляет итерации и увеличивает стоимость владения процессом. В DevSecOps, где безопасность должна работать в такт быстрой поставке, такой объём ручной работы становится неприемлемым. Плюс постоянный шум приводит к «усталости от предупреждений» — и уже на этапе разбора результатов растёт риск пропустить настоящие угрозы из-за снижения внимания и приоритизации, а сам инструмент в лучшем случае начинают использовать формально, сводя на нет ценность всего пайплайна анализа.

В интегрированных цепочках, когда результаты SAST автоматически уходят в системы оркестрации и корреляции (ASOC) или в платформы стратегического управления (ASPM), эффект шума лавинообразно масштабируется. Если первичные данные низкого качества, то ломается приоритизация, искажаются метрики безопасности и в итоге ресурсы расходуются не на те задачи. По сути, неточные результаты SAST создают иллюзию контроля: платформы управления показывают красивые дашборды, но реальные риски остаются за кадром, маскируясь под шум или теряясь в потоке ложных срабатываний. Поэтому точность и надёжность SAST — это уже не внутренняя «техническая метрика», а параметр



управления рисками, который напрямую влияет на окупаемость инвестиций в безопасность.

SASTAV: АРХИТЕКТУРА, ОРИЕНТИРОВАННАЯ НА ДАННЫЕ ВЫСОКОГО КАЧЕСТВА

Решение SASTAV изначально проектировалось так, чтобы держать баланс между полнотой обнаружения (recall) и точностью (precision, доля истинных срабатываний). Архитектура и функциональность нацелены на то, чтобы данные, которые затем попадут в процессы первичной оценки инцидента (триаж) и в управленческие системы, были максимально надёжными и объяснимыми.

В основе подхода — комбинация оптимизированного статического анализатора и многоэтапной валидации результатов с применением ИИ. После первичного сканирования потенциальные уязвимости проходят каскад специализированных моделей. Ключевой момент — контекстный анализ без передачи в модель полного исходного кода, что важно для соблюдения требований конфиденциальности и внутренних политик работы с интеллектуальной собственностью. Вместо этого формируется набор запросов (промптов), включающий только релевантные фрагменты, информацию о потоках данных и метаданные. В результате система помогает отделить реальные проблемы от шума и даёт разработчику понятное объяснение. По результатам пилотных внедрений и контрольных выборок это позволяет сократить объём ложных срабатываний на десятки процентов, в отдельных сценариях — более чем на 80%, без ухудшения полноты на проверяемых наборах.

Дополнительно на качество результатов сильно влияет создание и отладка собственных правил, которые учитывают специфику кодовой базы: внутренние безопасные обёртки, соглашения, архитектурные паттерны. В SASTAV есть встроенный редактор правил с интерфейсом, близким к IDE: можно создавать, отлаживать и тестировать правила, опираясь на внутренние фреймворки и бизнес-логику, а не только на универсальные шаблоны. Поддержка широкого спектра языков (Java, C#, Python, Go, JavaScript, TypeScript, C++) и актуальных фреймворков помогает покрывать разнородный технологический стек.

Точность — не единственный параметр, по которому стоит оценивать SAST. Для встраивания в быстрые CI/CD-конвейеры критична скорость и предсказуемость анализа. Архитектура SASTAV поддерживает параллельное выполнение нескольких движков сканирования и рассчитана на контейнерные развёртывания (включая Kubernetes). Это даёт отказоустойчивость, горизонтальную масштабируемость и стабильное время анализа даже для крупных монолитов, снижая риск «узких мест» в поставке.

SASTAV поддерживает подход Shift Left Security («сдвиг влево»), интегрируясь с системами контроля версий (GitLab, GitHub, Bitbucket), CI/CD (Jenkins, GitLab CI, GitHub Actions, Azure DevOps), трекерами задач (Jira, YouTrack) и IDE. Развитый REST API позволяет встраивать проверки в существующие автоматизированные процессы и подключать результаты к платформам ASPM. Отдельно сделан акцент на миграционный сценарий: есть механизмы импорта и конвертации данных триажа из других SAST-решений, чтобы сохранить инвестиции в уже проделанную работу и снизить порог перехода.

Платформа поддерживает детальное разграничение прав доступа (RBAC) для больших команд, панели мониторинга и визуализацию потоков данных, чтобы разработчик видел контекст уязвимости. Практический эффект — можно быстрее разбирать результаты и устранять дефекты, а не спорить о том, «насколько это похоже на проблему».

SAST КАК СТРАТЕГИЧЕСКИЙ АКТИВ В АРХИТЕКТУРЕ APPSEC

Эволюция Application Security логично ведёт к централизации управления через платформы ASPM. Но их ценность определяется не количеством виджетов в интерфейсе, а достоверностью и воспроизводимостью данных, которые на них опираются. Инструменты вроде SASTAV, которые закрывают фундаментальную проблему качества первичных данных, становятся не просто сканерами, а частью управляемой архитектуры AppSec.

Их роль — обеспечить переход от реакции на шум к управлению реальными рисками. Для специалистов по безопасности это означает возможность сосредоточиться на стратегических задачах — архитектурном анализе, развитии практик безопасной разработки и поиске действительно сложных уязвимостей, требующих человеческого опыта, вместо бесконечной верификации ложных срабатываний. Разработчики же перестают отвлекаться на шумные уведомления и возвращаются к своей прямой задаче — созданию и поставке продукта. Кроме того, инвестиции в точный, быстрый и интегрируемый SAST — это инвестиции в качество всей цепочки: от коммита разработчика до отчёта для руководства. Если на входе чистые и объяснимые результаты, дальше можно масштабировать оркестрацию, метрики и автоматизацию без самообмана — и действительно ускорять разработку, не теряя безопасность.