Алексей Вишняков

Кандидат физико-математических наук,
старший инженер DevSecOps в Yandex Cloud

Современные вызовы и решения в безопасной разработке облачных сервисов

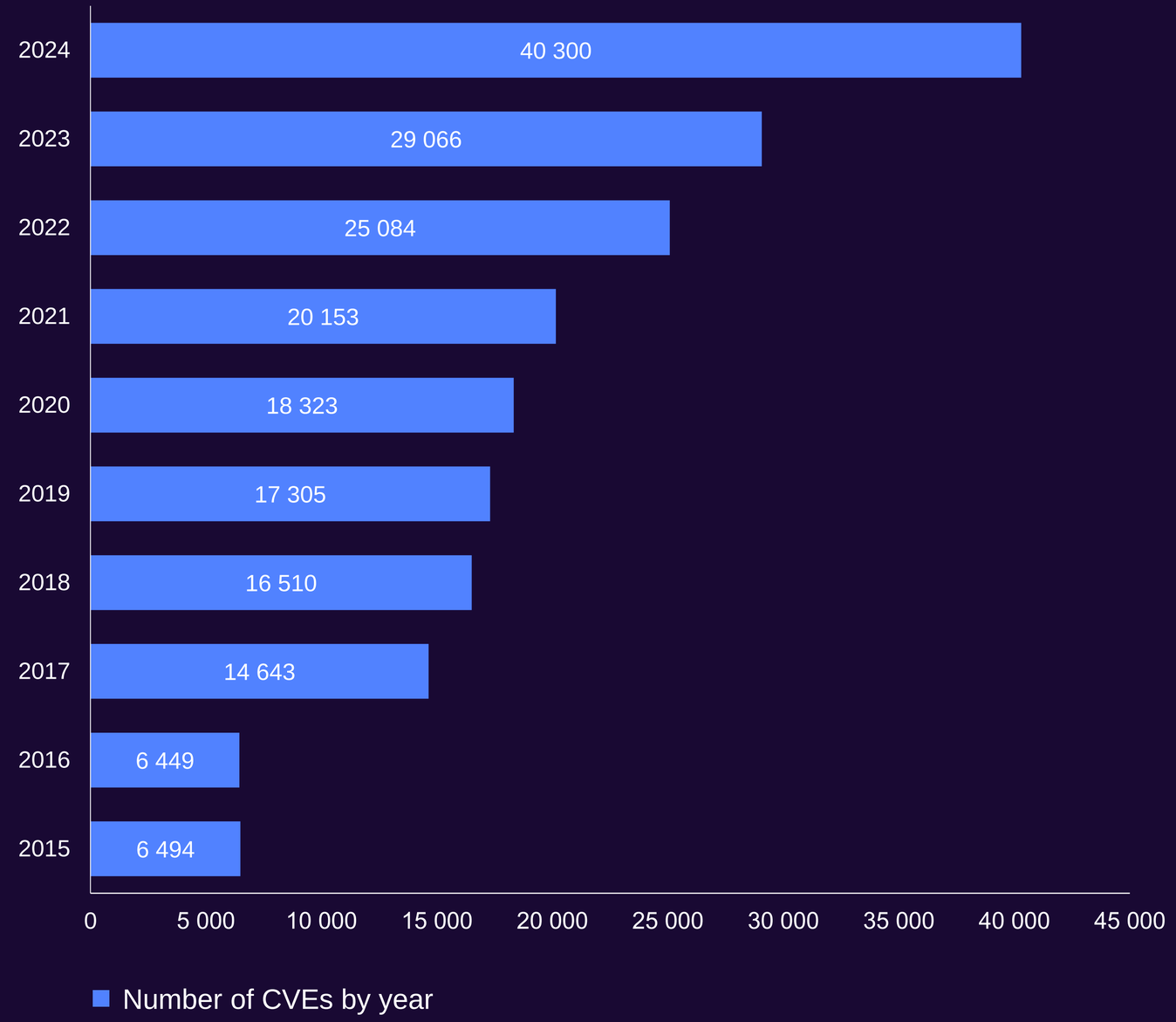
PHDAYS FEST от positive technologies

Whoami

- Руковожу направлением безопасной разработки (SSDLC) в Yandex Cloud
- Внедряю процессы и инструменты SSDLC
- Отвечаю за безопасность VCS/CI/CD
- Закончил бакалавриат и магистратуру ВМК МГУ
- Кандидат физико-математических наук по методам автоматизированного поиска уязвимостей в программах
- Ранее 8 лет в ИСП РАН вел исследования по компьютерной безопасности, динамическому анализу, символьному выполнению, фаззингу и др.
- Мейнтейнер поисковика гаджетов ROPgadget github.com/JonathanSalwan/ROPgadget и анализатора аварийных завершений после фаззинга CASR github.com/ispras/casr

Актуальность

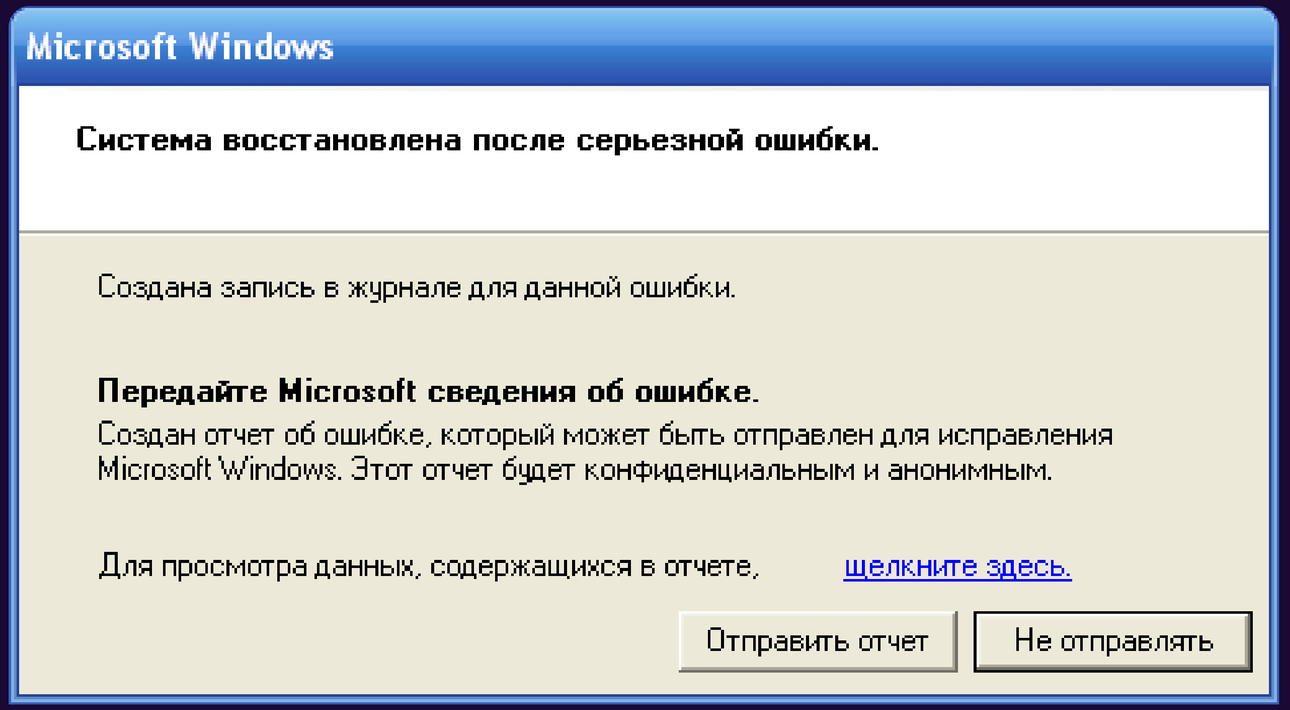
- Число найденных уязвимостей растёт с каждым годом
- 86% программ в 1658 коммерческих кодовых базах содержали хотя бы одну уязвимость (отчёт компании Black Duck® (ex-Synopsys) «2025 OSSRA»)
- Цикл разработки безопасного ПО является стандартом индустрии и позволяет обнаруживать ошибки в сервисе до деплоя



cvedetails.com

Безопасная разработка — стандарт индустрии

- Microsoft SDL: хотите отправить отчёт об ошибке?
- Cisco SDL
- ISO/IEC 15408-3-2008
- ГОСТ Р 56939-2016
- ГОСТ Р 56939-2024



Метод кластеризации отчётов об ошибках Microsoft ReBucket вдохновил на создание анализатора аварийных завершений после фаззинга CASR.

Мотивация

1

Поделиться опытом Yandex Cloud в решении ключевых вызовов безопасной разработки

2

Рассказать о новых подходах и инструментах безопасной разработки в Yandex Cloud

3

Дать практические рекомендации по предотвращению распространённых ошибок и выбору инструментов

Outline

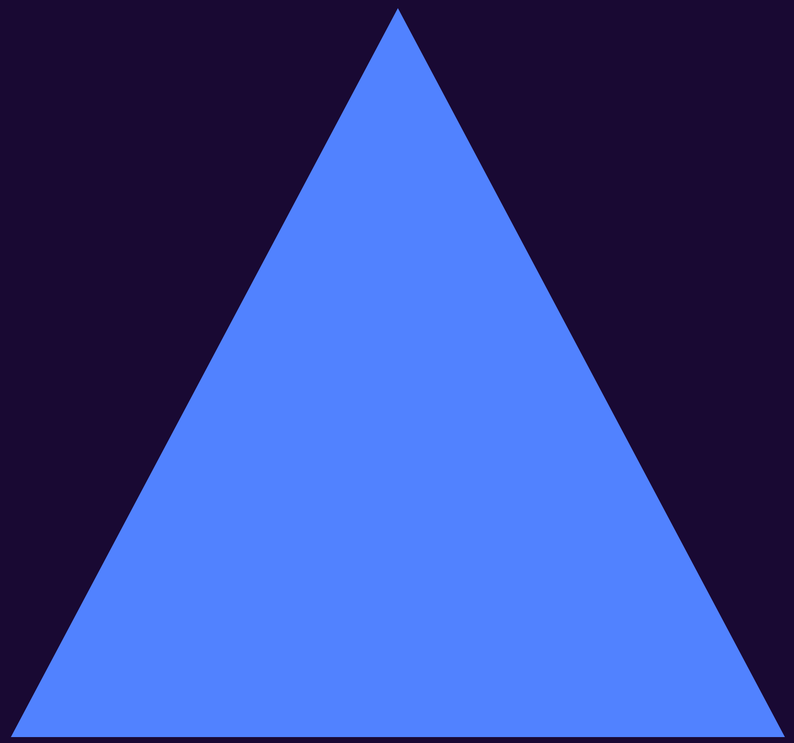
- Формирование требований по безопасности
- Унификация Security Pipelines для уменьшения операционной нагрузки
- Мониторинг здоровья процесса SSDLC
- Оптимизация SAST для снижения нагрузки на разработчиков
- Исследование открытых статических анализаторов
- Автоматизация фаззинга
- Интеграция SBOM с SIEM-системой для автоматизированного реагирования на инциденты безопасности
- Будущее безопасной разработки в Yandex Cloud

Формирование требований

Основы управления процессом SSDLC

Security Baseline

Формулировки требований



Security Cookbook

Техническая реализация требований

Regulations

Регламенты процессов

Yandex Cloud Security Baseline

Единая точка входа для всех внутренних требований к безопасности

Среди прочего охватывает ключевые аспекты безопасной разработки

Содержит только формулировки требований

Может быть использован как чек-лист

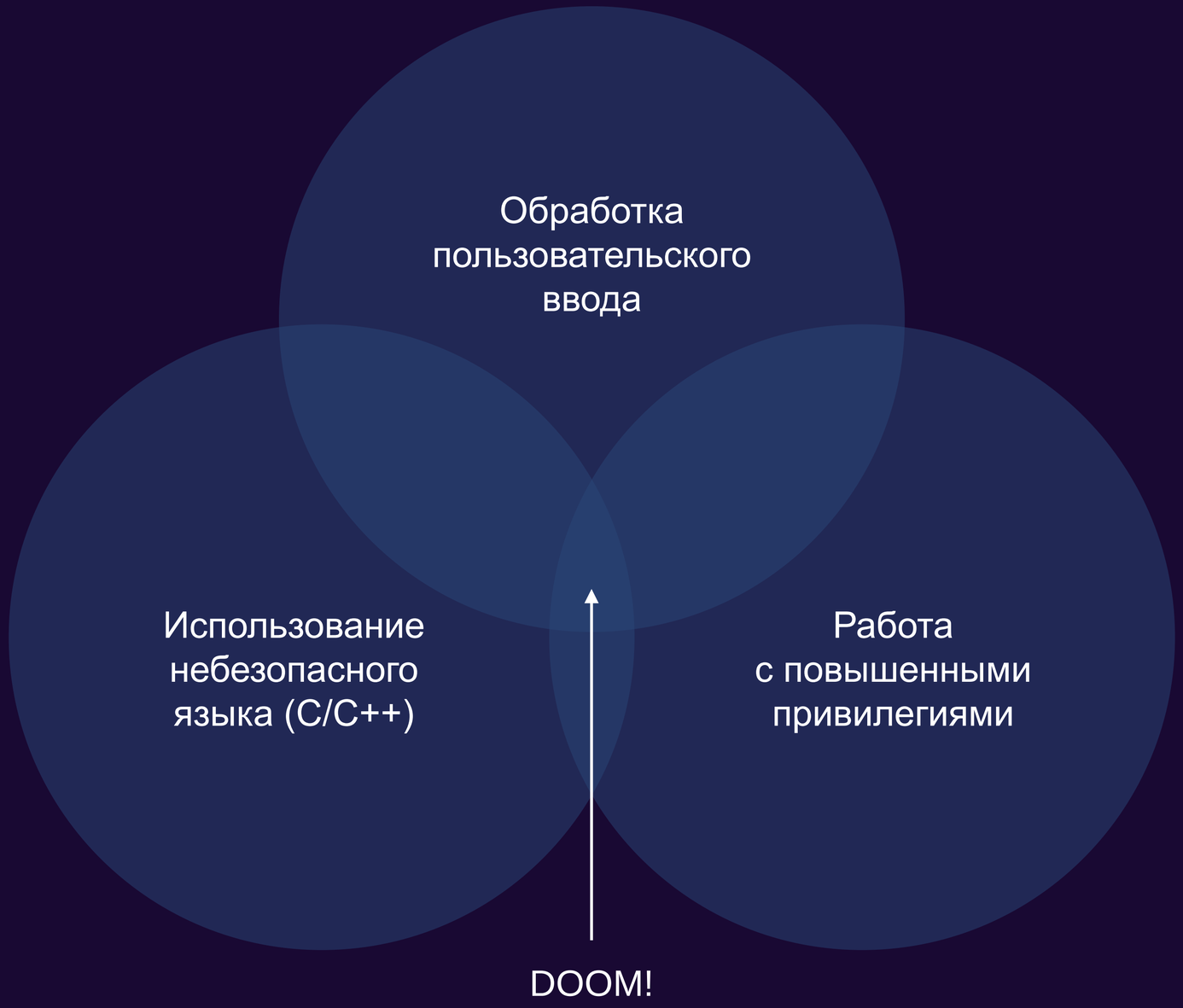
Положения разделяются как в ГОСТ:
на требования (должен), рекомендации (следует) и допустимые действия (может)

Для клиентов существует публичный стандарт по защите облачной инфраструктуры Yandex Cloud: yandex.cloud/ru/docs/security/standard/.

Требования к безопасной разработке

- Обучение
- Безопасность архитектуры
- Внешние зависимости
- Система контроля версий
- Сборка
- Тестирование
- Деплой
- Сканирование уязвимостей
- Жизнь в опенсорсе

The Rule of Two



chromium.googlesource.com/chromium/src/+/master/docs/security/rule-of-2.md

Примеры требований к системе контроля версий

- Разработчики должны вносить изменения (в т. ч. Cherry-Pick) в релизные ветки только через Pull Request (PR). Для этих веток должны быть запрещены внесение изменений без PR, перезапись истории (Force Push) и удаление
- Для каждого каталога, затронутого PR, должен быть approve хотя бы одного ревьювера из соответствующего ему списка ревьюверов
- Для мержа PR должны успешно пройти обязательные тесты в CI
- Разработчики должны контролировать отсутствие чувствительной информации в репозитории с кодом и PR: секретов, данных клиентов, приватных данных (PII)

Примеры требований к сборке

- Сборка должна обеспечивать статическую линковку с библиотеками
- Сборка должна происходить с hardening’ами на уровне компилятора

  Stack Protector, ASLR, RELRO...
- Выкладываемые в прод окружение сборки не должны содержать отладочной информации и исходного кода
- Артефакты сборки не должны содержать чувствительную информацию

  Секреты, токены, приватные ключи, пароли…
- Сборки должны быть герметичны

  Не требовать подключения к интернету
- Изменение конфигурации сборок релизных артефактов должно происходить только через PR

Примеры требований к деплою

- Деплой должен осуществляться только из релизных веток
- Все docker-образы (в т. ч. базовые) должны храниться во внутреннем Container Registry. Образы должны проходить регулярное сканирование на уязвимости
- Контейнеры должны запускаться с указанием опции --read-only для защиты от записи в корневую файловую систему
- Образы для виртуальных хостов должны создаваться из утверждённых базовых образов

Контроль выполнения требований

- Автоматизированные проверки

  Сканирование уязвимостей, Quality Gates, CI, SIEM…

- Конфигурация

  Правила Code Review, ACL, IaC…

- Security Audit новых сервисов и фич

- Внутренние compliance-аудиты

Security Audit

Для новых сервисов и фич в Yandex Cloud проводятся мероприятия

Аудит безопасности архитектуры

Аудит ролевой модели

Анализ диаграммы потоков данных

DFD

Моделирование угроз

Определение поверхности атаки

Составление списка функций под фаззинг

Ручной просмотр кода на предмет проблем с безопасностью

Формирование перечня рекомендаций и доработок безопасности сервиса

Унификация
Security Pipelines

Инструменты SSDLC

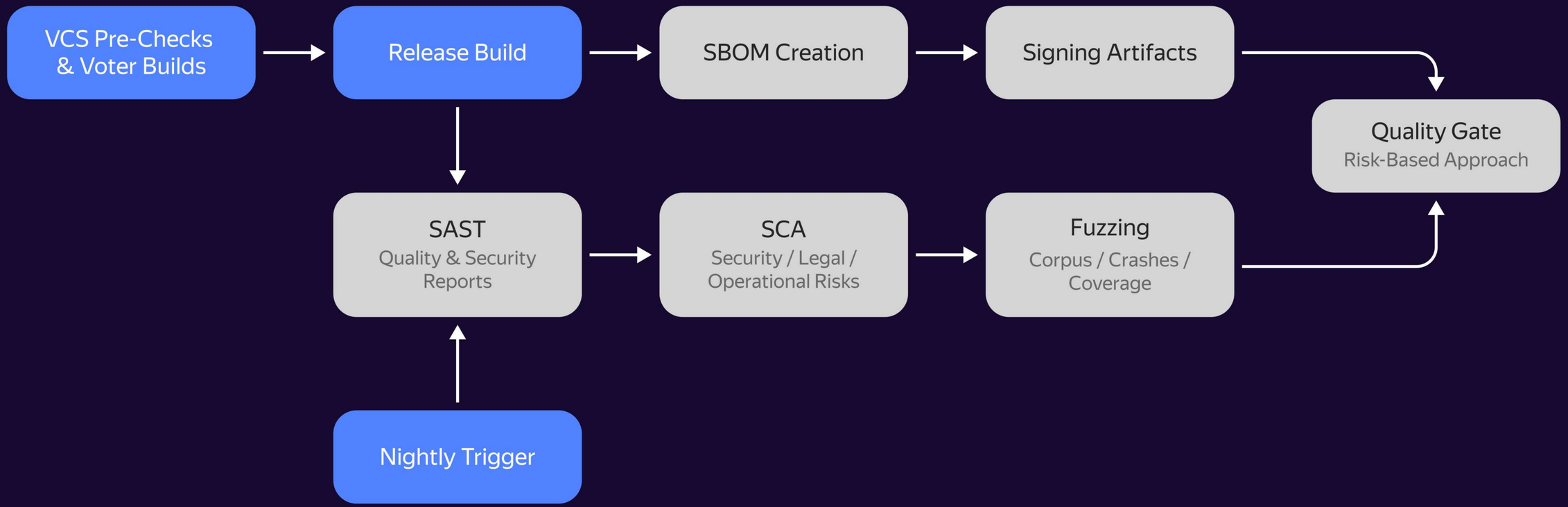
Статический анализ исходного кода (SAST) позволяет находить ошибки в коде, разрабатываемом внутри Yandex Cloud

Композиционный анализ (SCA) ищет известные уязвимости (CVE) в сторонних зависимостях и проверяет лицензионную чистоту

Фаззинг-тестирование (Fuzzing) подаёт сервису мутированные входные данные и обнаруживает аварийные завершения

Формирование CycloneDX SBOM в релизных сборках позволяет генерировать JSON с описанием внешних зависимостей и метаданными сборки

Security Pipelines



Унификация пайплайнов

- Централизованные скрипты для формирования SBOM, подписи артефактов, запуска инструментов SSDLC (SAST/SCA/Fuzzing) и проверки Quality Gates
- SBOM и подписи артефактов внедрены в базовые шаблоны релизных сборок сервисов
- Унифицированные шаги CI шаблонов запускают инструменты SSDLC и проверяют Quality Gates
- Шаблоны для разных языков программирования позволяют конфигурировать SSDLC-пайплайны через параметры
- Автоматика создаёт тикеты на разметку и исправление ошибок, найденных SAST-анализатором

Profit

Снижение операционки

Упрощение процесса подключения новых сервисов к SSDLC

Предсказуемый и управляемый процесс безопасной разработки

Централизация функций контроля требований и удобное внедрение новых проверок

Мониторинг здоровья процесса безопасной разработки

Мотивация

- Мониторинг технической работоспособности Security Pipelines
- Мониторинг Quality Gates и выполнение сервисами политик безопасности
- Отслеживание динамики изменения числа сработок инструментов безопасности (SAST, SCA, Fuzzing)
- Оперативное реагирование на проблемы

  Техническая работоспособность процесса должна поддерживаться в рабочее время

  Критические уязвимости должны быть вовремя исправлены

Метрики

- Число неисправленных находок SAST/SCA/Fuzzing по уровням критичности
- Выполнение требований к лицензионной чистоте
- Выполнение Quality Gates для SAST/SCA/Fuzzing

  Отсутствие срабатываний выше некоторого уровня критичности, который устанавливается отдельно для каждого сервиса

  Процент сервисов/микросервисов, выполняющих Quality Gates

- Работоспособность шагов Security Pipelines
- Число неразмеченных сработок и False Positive Rate для статического анализатора
- Изменение числа срабатываний SAST/SCA/Fuzzing за неделю/месяц
- Число открытых тикетов по SAST/SCA/Fuzzing

Внедрение

Дашборд в DataLens
с метриками на уровне
сервиса/микросервиса

Хранит слепки состояния
за каждый день

Дежурный по SSDLC смотрит
за работоспособностью
Security Pipelines

Vulnerability Manager следит
за числом уязвимостей и ~~пинает~~
ускоряет исправление в тикетах

Оптимизация SAST для снижения нагрузки на разработчиков

Мотивация

Большое количество
ложных срабатываний
и нерелевантных
чекеров SAST приводит
к снижению доверия
к статическому анализу

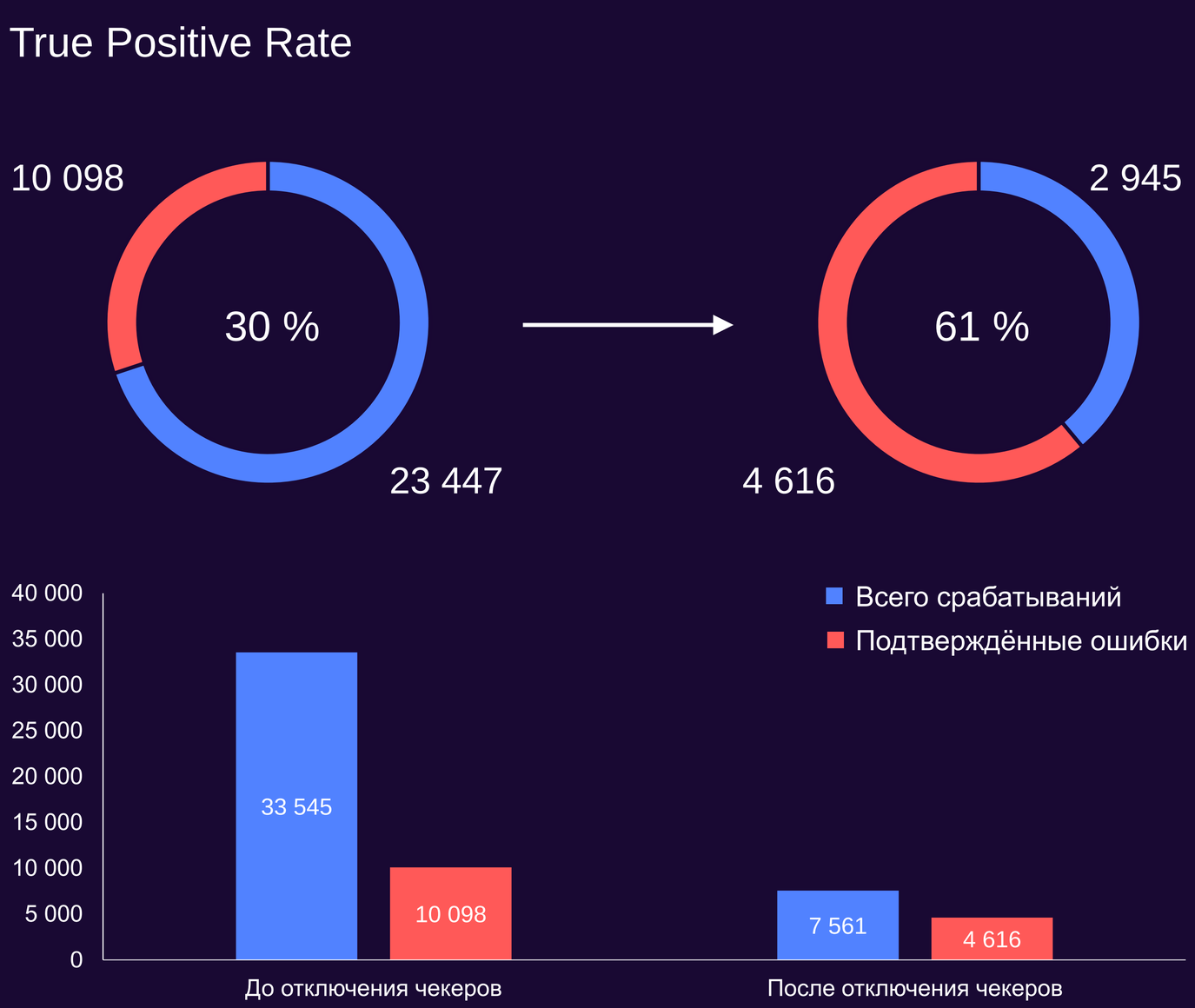
Хотим снизить нагрузку
на разработчиков
без потери качества
обнаружения
уязвимостей

Отключение SAST-чекеров

- Собрали дашборд с False Positive Rate по чекерам
- Проанализировали рекордсменов
- Отключили большинство code-quality-чекеров
- Некоторые группы чекеров дублировали друг друга
- Нельзя отключать критичные для безопасности чекеры, а также те, которые перечислены в ГОСТ Р 71207-2024 по статическому анализу

Результаты

- Увеличили True Positive Rate и снизили суммарное число срабатываний
- Результаты приводятся за полгода до и после отключения SAST-чекеров
- Нагрузка на разметку уязвимостей снизилась с 6 до 2 FTE, если закладывать на разметку одного срабатывания 15 минут



Исследование открытых статических анализаторов

Мотивация

Перейти с тяжёлых enterprise-продуктов на опенсорс

Получить больше контроля над инструментами и процессом SSDLC

Идти в ногу с новыми версиями языков программирования

Сейчас многие коммерческие решения становятся недоступными

Вызовы

- В Yandex Cloud богатый технологический стек (Go, Java, C/C++, TypeScript…) — далеко не все SAST умеют собирать наш код из коробки, что добавляет трудозатраты на доработку
- Надо быть готовыми, что не будет одного коробочного решения — существуют разные анализаторы под конкретные языки
- Анализатор должен выдерживать огромную кодовую базу в монорепозитории

Критерии выбора SAST-анализатора

- Список чекеров и качество анализа кода сервисов Yandex Cloud сопоставимы с текущим решением

  Необходимо делать кросс-валидацию результатов различных анализаторов

- Анализатор поддерживается известными компаниями

  Microsoft, FAANG...

- Есть развитое сообщество вокруг инструмента

  Много звёздочек и внедрений

- Анализатор соответствует требованиям ГОСТ Р 71207-2024 по статическому анализу

  В т. ч. по списку поддерживаемых чекеров

- SAST поддерживает сохранение результатов в формате SARIF
- SAST совместим с текущей сборочной инфраструктурой и предоставляет возможность удобной автоматизации процессов

Go

- Gosec github.com/securego/gosec
  - Ориентирован на поиск уязвимостей
  - Поддерживает SARIF
- Staticcheck github.com/dominikh/go-tools
  - Главным образом ищет ошибки, связанные с качеством кода
  - Однако он ищет некоторые ошибки полезные для ГОСТ, например nil deref
  - Хорошо дополняет Gosec группами чекеров SA1, SA2 и SA5
  - Поддерживает SARIF

SAST удобно запускать через golangci-lint

Multi-Language SAST (WIP)

- GitHub CodeQL github.com/github/codeql
- Semgrep github.com/semgrep/semgrep
- Infer (C/C++/Java) от известной социальной сети

  Поддерживает символьное выполнение
- CodeChecker github.com/Ericsson/codechecker

  Удобная автоматизация запуска Clang Static Analyzer, Clang-Tidy, Infer, SpotBugs (Java), Bandit (Python), Staticcheck (Go)…

  Современный сервер для хранения и разметки срабатываний
- SonarQube github.com/SonarSource/sonarqube

Верным выглядит подход с комбинацией SAST для конкретного языка и CodeQL/Semgrep для поиска актуальных уязвимостей по набору правил

C/C++ (WIP)

- Clang-Tidy clang.llvm.org/extra/clang-tidy

  Хоть и линтер, но некоторые уязвимости искать тоже умеет

- Clang Static Analyzer clang-analyzer.llvm.org

  Ищет широкий спектр уязвимостей

  Межпроцедурный, межмодульный анализ с поддержкой taint и символьного выполнения

- Cppcheck cppcheck.sourceforge.io

  Старый и топорный статический анализатор в дополнение к первым двум

Все эти анализаторы удобно запускать через CodeChecker

Java (WIP)

- SpotBugs github.com/spotbugs/spotbugs
- OWASP Find Security Bugs github.com/find-sec-bugs/find-sec-bugs

  Плагин для SpotBugs
- PMD github.com/pmd/pmd
- Error Prone github.com/google/error-prone

JavaScript/ TypeScript (WIP)

- Eslint-plugin-security github.com/eslint-community/eslint-plugin-security

  Набор правил безопасности для ESLint

  Выглядит так, что других нет

  Можно загружать результаты в CodeChecker

- Не забываем про CodeQL/Semgrep

Автоматизация фаззинга

Определение поверхности атаки

- На Security Audit команда безопасности рекомендует сервису, какие функции стоит пофаззить
- Она опирается на анализ архитектуры и диаграммы потока данных
- Эти функции должны лежать на поверхности атаки
- Функции должны быть доступны через внешний интерфейс

  Необязательно напрямую

- Интересны функции, которые получают через аргументы входные данные и парсят их
- Чем проще добраться до функции (ближе к точке входа), тем она приоритетнее для фаззинга

  gRPC — верхнеуровневая точка взаимодействия с сервисом

  А библиотека архивации лежит глубже

  В любом случае нужно оценить достижимость функции от входной точки

Примеры функций под фаззинг

Функции, обрабатывающие входные данные (parse*, load*, import*, read*…)

Парсеры/кодеки форматов, сеть, межсервисное взаимодействие, конфигурация, формы ввода...

Необходимо оценивать сложность инициализации контекста функции — буфер предпочтительнее сложных объектов

Примеры фаззинг-целей можно смотреть в OSS-Fuzz

Два похода к фаззингу в Yandex Cloud

1

Фаззинг gRPC-ручек

Structure-Aware Fuzzing через Protobuf Mutator

Непосредственно фаззинг поверхности атаки сервиса

Трудоёмкая реализация

2

Фаззинг отдельно взятых функций

Фаззинг узких частей поверхности атаки

Обнаружение глубоких багов

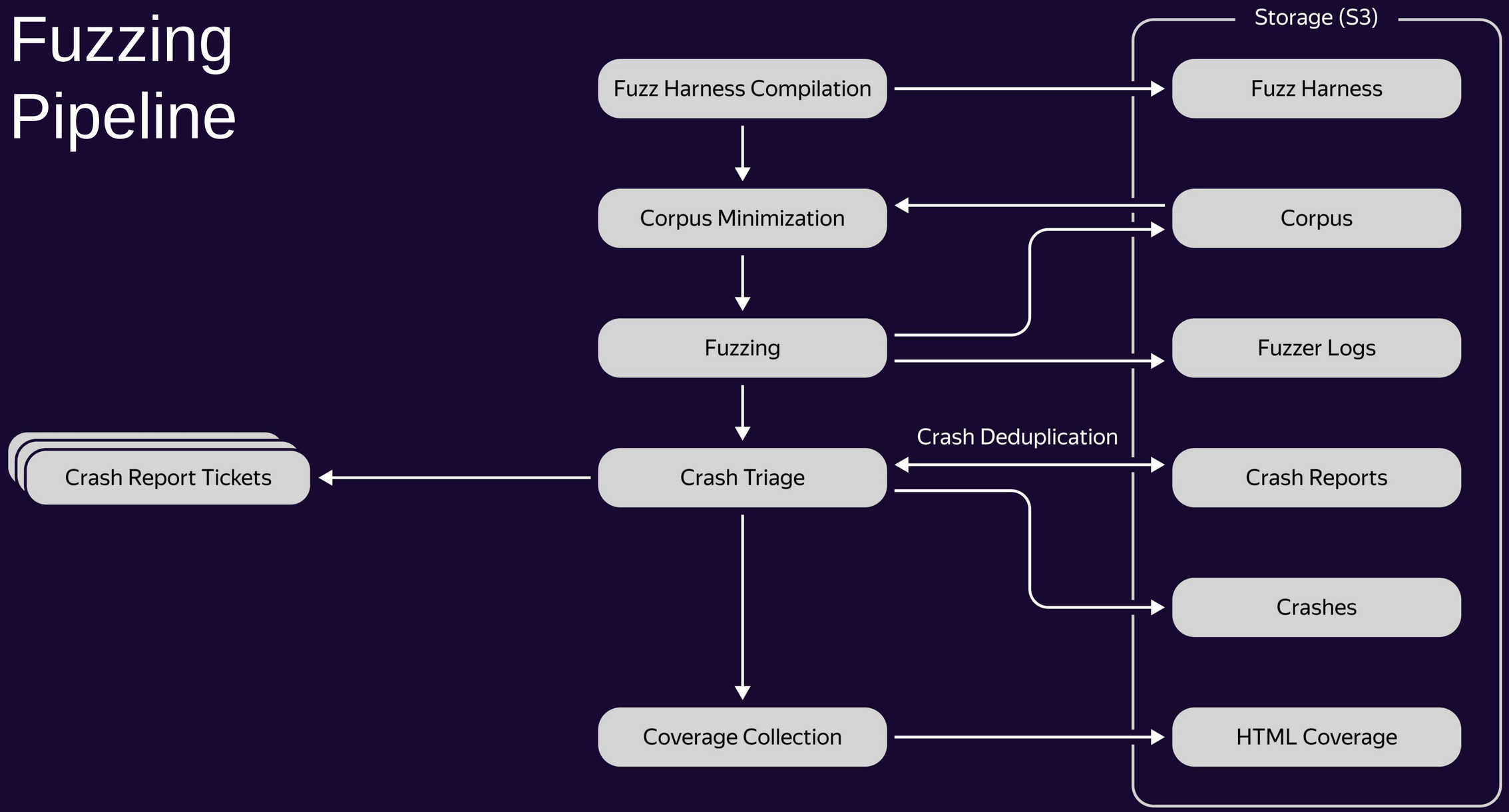
Прост в реализации

Go Protobuf Mutator

```
func LLVMFuzzerTestOneInput(data *C.char, size C.size_t) C.int {
    gdata := unsafe.Slice((*byte)(unsafe.Pointer(data)), size)
    message := NewProtoMessage(gdata)
    Fuzz(message)  // Call your code for fuzzing one message
    return 0
}

func LLVMFuzzerCustomMutator(data *C.char, size C.size_t, maxSize C.size_t, seed C.uint)
C.size_t {
    gdata := unsafe.Slice((*byte)(unsafe.Pointer(data)), size)
    message := NewProtoMessage(gdata)
    mutator := mutator.New(int64(seed), int(maxSize-size))
    if err := mutator.MutateProto(message); err != nil {
        fmt.Printf("Failed to mutate message: %+v", err)
        return 0
    }
    gdata = unsafe.Slice((*byte)(unsafe.Pointer(data)), maxSize)
    newSize := StoreMessage(gdata, message)
    return C.size_t(newSize)
}
```

Fuzzing Pipeline

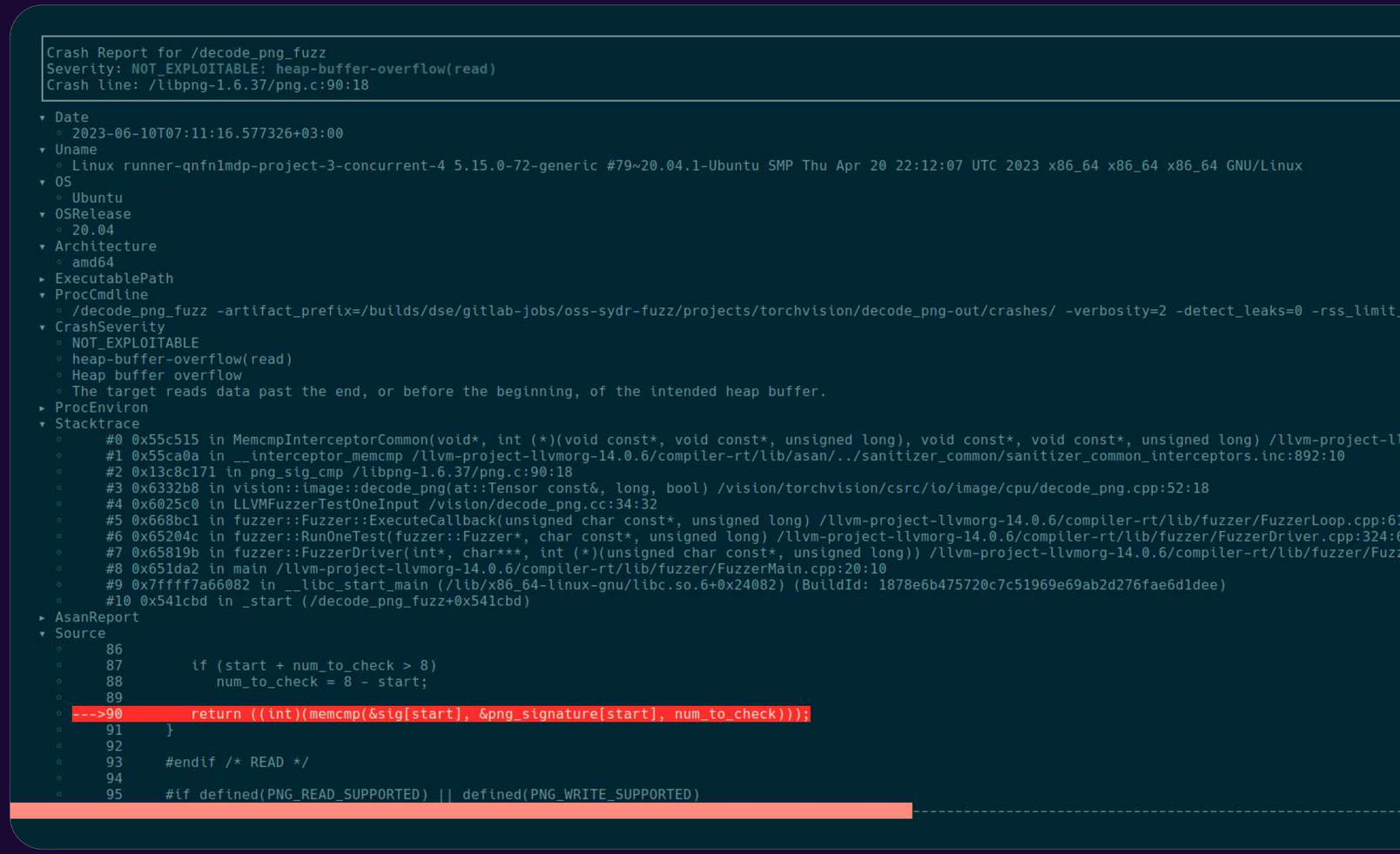


Конфиг фаззинга

```
service: "compute"
target: "./fuzz_target"
initial_corpus: "./initial-corpus"
cov_html: "./result.html"
bucket_name: "compute-fuzzing"
libfuzzer:
  max_total_time: 7200
  rss_limit_mb: 9000
build: |
  make fuzzing
coverage: |
  make coverage
tickets:
  queue: "CLOUD"
  followers:
    - "sweetvishnya"
  assignee: "sweetvishnya"
  tags:
    - "sdl"
    - "fuzzing"
```

CASR — Crash Analysis and Severity Report

- Создаёт отчёты и дедуплицирует аварийные завершения, найденные в результате фаззинга (libFuzzer, go-fuzz, Atheris, Jazzer, luzer, AFL++, SharpFuzz...)
- Поддерживает C, C++, C#, Go, Java, JavaScript, Lua, Python, Rust
- Оценивает критичность уязвимости
- Подробнее в докладе: Andrey Fedotov, Alexey Vishnyakov. CASR: Your Life Vest in a Sea of Crashes. OFFZONE 2023



Дедупликация аварийных завершений

1. `casr-libfuzzer -f -i ./crashes -o ./casr-out --timeout 300 --no-cluster -- ./fuzz_target`
2. Скачиваем накопленные отчёты CASR из S3 в `./prev`
3. `casr-cluster --diff ./casr-out ./prev ./diff`
4. Создаём тикеты для крэшей из `./diff` и загружаем их в S3

SBOM в SIEM для реагирования на инциденты безопасности

Мотивация

- Yandex Cloud одновременно обслуживает множество сервисов, которые регулярно обновляются
- Важно контролировать состояние прода и понимать:

  какие версии сервисов крутятся в проде

  где и кем были собраны пакеты / docker-образы

  из каких компонентов состоят

  подвержены ли они уязвимостям

- Внесённая ошибка в общий код (например, в библиотеку логирования) может привести к нарушению безопасности целой группы сервисов. Как понять, каким сервисам необходимо передеплоиться?
- Вышло очередное критичное обновление RCE в библиотеке (например, Log4Shell). Как быстро найти в проде затронутые сервисы?

Цели

- Алерты в SIEM-системе

  Контроль целостности файлов (FIM) и docker-образов

  Файлы не должны попадать в прод в обход CI/CD-системы

- Политики деплоя

  Запрет на деплой из личных веток и отладочных пайплайнов

- Сканирование уязвимостей

  Регулярное сканирование зависимостей (SCA) сервисов, налитых в прод

- Автоматизированное расследование инцидентов ИБ

  Поиск сервисов в проде, собранных из уязвимого общего кода

  Обнаружение хостов, на которых запущены сервисы с уязвимой сторонней библиотекой

Work in Progress

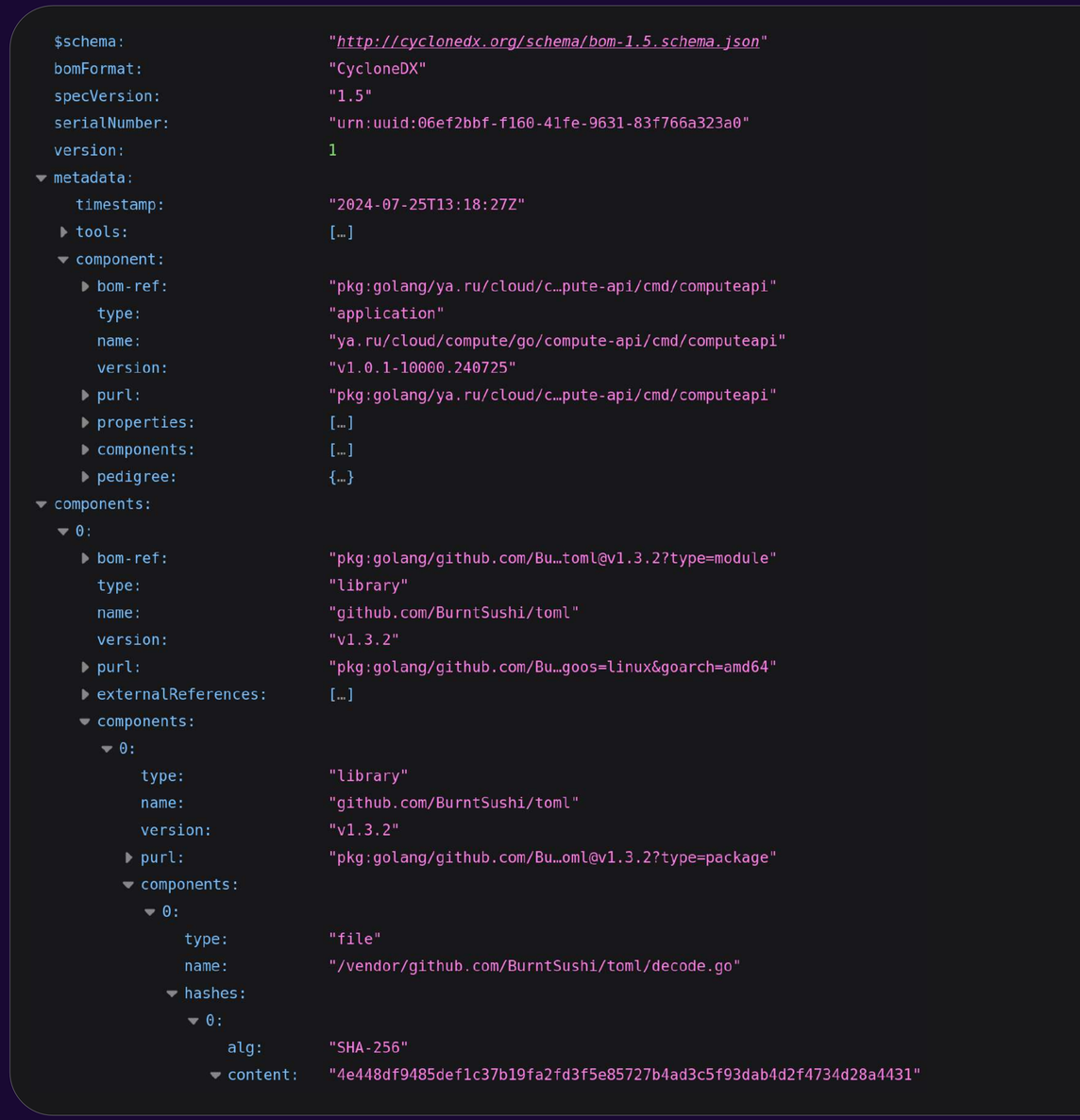
- Обогащённые SBOM формируются для сервисов на Go и Java
- SBOM хранятся в SIEM-системе
- Аналитик SOC может расследовать инциденты ИБ с помощью YQL-запросов к хранилищу SBOM
- Композиционный анализ (SCA) осуществляется во время сборки

  Разрабатываем механизм регулярной проверки зависимостей сервисов, уже запущенных в проде, и порождения алертов

- Для отправки событий с хешами файлов на хосте переходим с Osquery (User Mode) на Tetragon (eBPF/LSM)
- Разрабатываем формирование SBOM для C++ по метаданным из монорепозитория

CycloneDX SBOM

- Software Bill of Materials (SBOM) содержит описание всех внутренних и сторонних зависимостей
- Базовая сущность в формате CycloneDX — компонент (библиотека, deb-пакет, docker-образ, файл и др.)
- `cyclonedx-gomod app -json -files -paths -licenses -packages -std -output bom.json -main ./compute-api/cmd/computeapi`
- Для Maven™ и Gradle есть плагины, которые генерируют SBOM во время mvn package и ./gradlew:service:app:cyclonedxBom
- `cyclonedx-py requirements --sv 1.6 > bom.json`



Обогащение SBOM

Полученный SBOM обогащается недостающими метаданными о сборке и артефактах

- Информация про VCS (хеш коммита, ссылка, автор, ветка, тег, сабмодули)
- Параметры сборки (ссылка, ID, номер, имя, кем запущена, проект, сборочный агент, время)
- Deb-пакеты (имя, версия, описание, хеши файлов)
- Docker-контейнеры (имя, тег, digest, image ID, время создания, хеши файлов)
- Хеши файлов с исходным кодом

VCS Info

Информацию про коммит сохраняем
в pedigree, а остальное в properties:

```
git show --quiet --pretty=format:…
git config --get remote.origin.url
git branch --show-current
git rev-list --count HEAD
git describe --tags --exact-match
git submodule status --recursive
```

Ищем файлы bom.json, .deb и др.
только в изменённых файлах:

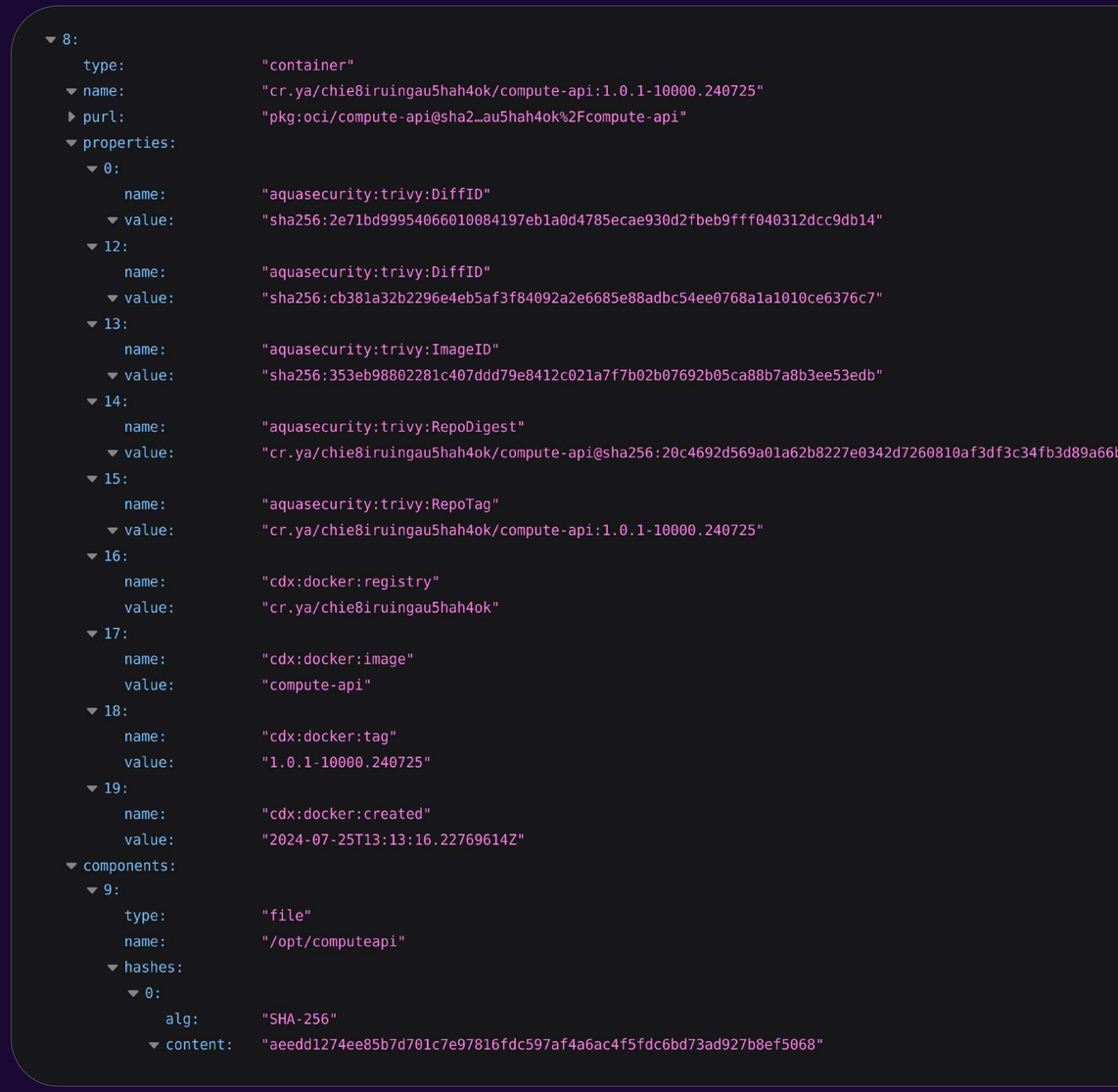
```
git status --ignored=matching -s
```

Deb-пакеты

- Получаем информацию про пакет: `dpkg –I`
- Считаем хеши для содержимого пакета: `dpkg –x`
- Помним, что в пакете могут быть вложенные deb-пакеты и архивы

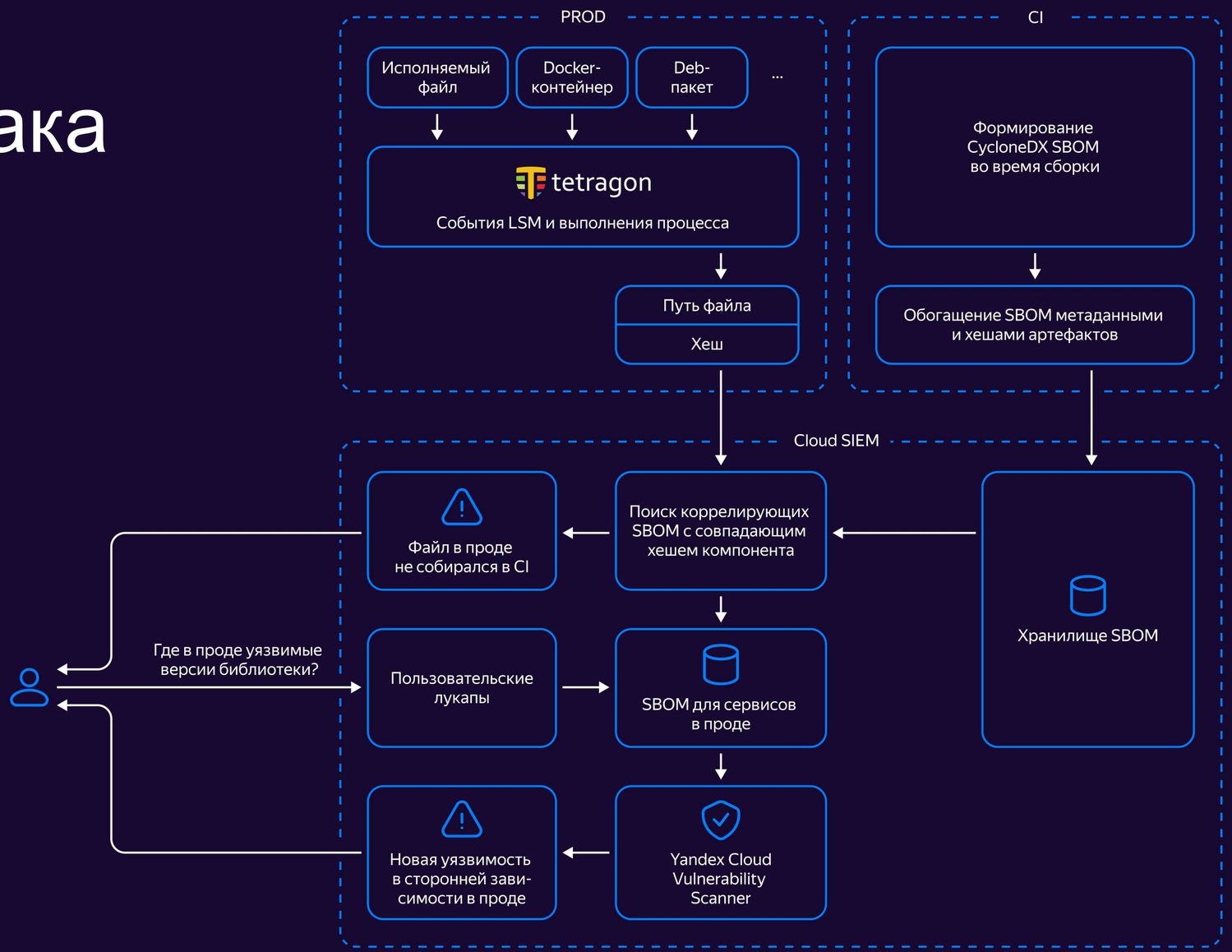
Docker-образы

- Dive тяжеловесен, чтобы приносить его во все сборки
- Проанализируем докер своими руками
- Включим сохранение событий: `docker events --format json`
- Запустим на каждом pushed-образе: `docker inspect`
- Считаем хеши для файлов в слоях: `docker save`
- При этом пропускаем pulled layers



SBOM в SIEM для защиты облака

- Обогащённые SBOM формируются в сборках и отправляются в хранилище внутри SIEM
- Tetragon отправляет в SIEM события с хешами файлов на проде
- SIEM находит SBOM с совпадающими хешами артефактов или порождает алерт об их отсутствии
- Vulnerability Scanner регулярно проверяет SBOM сервисов в проде на наличие новых уязвимостей
- Пользовательские лукапы позволяют получить SBOM, удовлетворяющий заданным свойствам



Будущее безопасной разработки в Yandex Cloud

Внедряем искусственный интеллект в процессы SSDLC

1

Автоматизированная разметка SAST и SCA сработок с ИИ-агентом

2

Анализ достижимости уязвимостей, обнаруженных SCA

3

SourceCraft Code Assistant для генерации фаззинг-целей

Будущее

- Независимые от вендоров (опенсорс) инструменты SSDLC
- Сканирование IaC-деплоя
- Переезд открытых решений Yandex Cloud в SourceCraft с интегрированными механизмами мониторинга безопасности Cloud SIEM
- SBOM Asset Management в Cloud SIEM с отображением всех сервисов, их зависимостей, уязвимостей и этапа жизненного цикла

Оцените доклад!

Спасибо!