DOI: 10.15514/ISPRAS-2024-36(1)-4

Фаззинг полиморфных систем в структурах микросервисов

А.С. Юрьев, ORCID: 0009-0003-0369-0422 <*forhhpurpose@yandex.ru*>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Сегодня фаззинг, фаззинг-тестирование является основной техникой тестирования программного обеспечения, систем и функций, в том числе и как часть динамического анализа. Фаззинг позволяет выявлять дефекты информационной безопасности или отказы. Однако такая практика может требовать привлечений больших ресурсов и вычислительных мощностей для проведения работ в крупных организациях, где количество систем может быть большим. Командам разработки и специалистам информационной безопасности требуется одновременно соблюдать сроки, требования различных регуляторов и рекомендации стандартов. Для решения задач по фаззинг-тестированию при одновременном соблюдении сроков, предлагается метод фаззинг-тестирования, который следует применять сразу ко всей информационно-вычислительной сети крупных организаций, которые используют микросервисы. Под полиморфными системами в настоящей статье понимаются такие системы, которые содержат реализацию различных функций, принимающих на вход различные типы данных, не в рамках одного программного обеспечения, а в рамках подсистем с набором нескольких микросервосов. В этом случае могут использоваться различные сетевые протоколы, форматы и типы данных. При таком многообразии особенностей, возникает проблема выявлений дефектов в составе систем, поскольку при разработке не всегда предусматриваются интерфейсы отладки или обратной связи. Для её решения в настоящей статье предлагается использовать метод сбора и анализа статистики временных интервалов обработки мутированных данных миросервисами. Для фаззинг-тестов предлагается использовать мутированные запросы, где начальное состояние данных для мутации – полезная нагрузка известных или типовых дефектов информационной безопасности. С помощью анализа временных интервалов между клиент-серверным запросом и ответом удалось выявить закономерности, которые показали наличие потенциально опасных дефектов. В рамках статьи рассматривается фаззинг прикладных функций по протоколу HTTP. Предлагаемый подход не оказывает отрицательных влияний на эффективность и сроки разработки. Описанный в статье метод и решение рекомендуется применять в крупных организациях, как дополнительное или основное решение по обеспечению информационной безопасности для того, чтобы предотвращать критичные отказы инфраструктуры и финансовые потери.

Ключевые слова: фаззинг; дефекты информационной безопасности; микросервисная архитектура.

Для цитирования: Юрьев А.С. Фаззинг полиморфных систем в структурах микросервисов. Труды ИСП РАН, том 36, вып. 1, 2024 г., стр. 45–60. DOI: 10.15514/ISPRAS–2024–36(1)–4.

Fuzzing of Polymorphic Systems within Microsevice Structures

A.S. Yurev, ORCID: 0009-0003-0369-0422 <*forhhpurpose@yandex.ru*>
Institute for System Programming of the Russian Academy of Sceinces,
25, Alexander Solzhenitsyn st, Moscow,109004, Russia.

Abstract. Today fuzzing (fuzzing-testing) is the main technique for testing software, systems and code functions. Fuzzing allows identify vulnerabilities or software failures. However, this practice may require the large resources involvement and network performance in large organizations where the number of systems may

be large. Developers and information security specialists are simultaneously required to comply with time-to-market deadlines, requirements of various regulators and recommendations of standards. In current paper is proposed new fuzzing method, which is designed to solve the problem above. In current aproach is proposed use fuzzing testing for whole computing network at ones in large organizations if them operate with microservices. Polymorphic systems in this paper are understood like systems that consist of various API (Application Programming Interface) functions that operate with various types of data, not within single software, but inside subsystems with a set of several microservices. In this case, a lot of various network protocols, data types and formats can be used. With such a variety of features, there is a problem of detecting errors or vulnerabilities inside systems, beacause debugging or trace interfaces are not always developed in the microservice softwares. So, in this paper it is proposed to use also the method of collecting and analyzing statistics of time intervals of processing mutated data by microservices. For fuzzing tests, it is proposed to use mutated lists of exploit payloads. Time analyzing between client-server requests and the responses helps to identify patterns that showed the presence of potentially dangerous vulnerabilities. This paper discribes fuzzing of API functions only in the HTTP protocol (Hypertext Transfer Protocol). Current approach does not have a negative impact on the effectiveness of development or deadlines. Methods and solution described in the paper are recommended to be used in large organizations as an additional or basic information security solution in order to prevent critical infrastructure failures and financial losses.

Keywords: fuzzing; information security; micro-service architecture.

For citation: Yurev A.S. Fuzzing of polymorphic systems within microservice structures. Trudy ISP RAN/Proc. ISP RAS, vol. 36, issue 1, 2024. pp. 45-60 (in Russian). DOI: 10.15514/ISPRAS-2024-36(1)-41.

1. Введение

Сегодня в крупных организациях часто используют микросервисные принципы построения информационно-вычислительной архитектуры [1]. Микросервисы – это набор небольших модулей, с помощью которых выполняется непрерывная поставка и развертывание больших и сложных приложений (программного обеспечения). Чаще всего, это веб-сервис (Web-service), отвечающий за один элемент логики в определенной предметной области. Приложения на таких принципах представляют из себя комбинацию микросервисов, каждый из которых предоставляет определенные функциональные возможности пользователям [2]. Под полиморфными системами в настоящей статье понимаются такие системы и микросервисы, которые состоят из множеств различных API (Application Programming Interface) функций, принимающих на вход различные типы данных, не в рамках одного программного обеспечения (ПО), а в рамках автоматизированных систем (АС) с набором нескольких ПО. В этом случае могут использоваться различные сетевые протоколы, ролевые модели, форматы и типы данных, что порождает проблемы исследований и аудита информационной безопасности (ИБ) в крупных организациях, поскольку количество объектов и их особенностей может быть большим. В первую очередь, проблемы связаны с тем, что при проведении работ по анализу защищенности специалистами ИБ требуется соблюдать сроки (time-to-market). Например, в финансовых организациях соблюдение сроков является острой проблемой, поскольку рынок финансовых услуг очень динамичный и резко меняется. Кроме того, в соответствии с принципами SSDL (Secure Software Development Lifecycle) [3] и требованиями регуляторов, каждая новая версия разработки ПО обязана проходить проверки ИБ на регулярной основе в рамках аудита защищенности различных АС и для решения задач по анализу отказоустойчивости. Для этого проводится фаззинг-тестирование (англ. Fuzzing) [4]. В рамках настоящей статьи фаззинг-текстирование рассматривается, как один из вариантов практики динамического сканирования – DAST (Dynamic Application Security Testing) [5]. В основе техники фаззинга лежит метод подачи на вход функций ПО некорректных, случайных или нестандартных данных с целью обнаружения ошибок [6-7]. Практика фаззинг-тестирования регламентирована рекомендациями различных стандартов [8-10]. Инструменты фаззинг-тестирования входят в стек технологий цикла безопасной разработки, регламентированного нормативными документами ведомств-регуляторов, проведение таких процедур предусмотрено ГОСТ Р

56939-2016, как необходимой меры разработки безопасного ПО [6,11]. Поэтому сегодня вопрос проведения фаззинг-тестирований в крупных организациях является крайне актуальным. Для выполнения рекомендаций и требований при одновременном соблюдением сроков ввода ПО в промышленную эксплуатацию, в рамках данной статьи предлагается использовать новый метод фаззинга для структур микросервисов сразу ко всей АС, поскольку именно такой подход используется злоумышленниками, в соответствии с моделью внешнего нарушителя [12]. В этом случае, с точки зрения тестирования на проникновение, АС будет атакована по методологии «черного» или «серого ящика» [13], так как нарушителю или не известна архитектура систем в организации, или он каким-то образом получил всё множество API-функций и их параметры. В настоящей статье проводится анализ различных инструментов, применительно, к микросервисным структурам и разрабатывается метод обнаружения дефектов ИБ, основанный на анализе времени ответов сервисов на запросы. В рамках статьи рассматривается фаззинг API-функций в составе протокола HTTP (Hypertext Transfer Protocol), поскольку сегодня микросервисы преимущественно работают по спецификациям Swagger [14] или OpenAPI [15]. Доказывается эффективность метода мутации полезных нагрузок (payloads) [16-17] известных дефектов ИБ на практике. Исследуются проблемы фаззинга АС в крупных организациях.

2. Описание проблемы фаззинг-тестирования в крупных организациях

В крупных организациях, где информационные системы построены на микросервисных принципах, в АС с течением времени разрабатывается множество новых API, системных интеграций, обновлений, постоянно ведутся разработки новых функций, микросервисы создают полиморфизм и их количество может возрастать в разных АС непрерывно и независимо. Часто, каждый сервис в составе АС существует только в рамках своего жизненного цикла (SDLC) [18] и не зависит от разработок в других частях АС. Бизнес-функции, команды разработчиков и стек технологий также могут быть разными.

В соответствии с этим, внедрение фаззинг-тестирований в отдельности для каждого микросервиса или ПО создает предпосылки для снижения эффективности разработки. Это обусловлено тем, что может потребоваться:

1) Проводить предварительный анализ исследуемой системы, определение объектов фаззинга, анализ обновления;
2) Тонко настраивать инструменты фаззинга и вносить в контур разработки дополнительные библиотеки;
3) Создавать агенты фаззинга и доставлять их на сервера разных систем;
4) Настраивать авторизацию в системах;
5) Создавать дополнительных тестовые контура для предотвращения отказов в промышленной среде во время фаззинга;
6) Создавать механизмы контроля состояния систем во время проведения работ;
7) Непрерывно взаимодействовать с командами разработчиков, с отрывом от выполнения основных задач;
8) Расширять штат специалистов ИБ;
9) и выполнять другие активности.

Предполагается, что вышеперечисленные особенности не только могут увеличить сроки, но и потребуют дополнительных ресурсов, которые в свою очередь, увеличивают финансовые расходы.

Кроме того, существует и другая проблема, которая связана с наличием обратной связи в исследуемых системах. Не всегда разработчики предусматривают интерфейсы, которые

позволяют выявлять дефекты ИБ или ошибки. К ним можно отнести системные журналы, логи ошибок (syslog), обработчики ошибок в составе конечного ПО. Иногда, в составе различных сервисов используются коммерческие решения, включая аппаратные (hardware), исходный код которых недоступен разработчикам организации. Некоторые прикладные решения, например, такие как Web-серверы, базы данных или ПО, которые работают по протоколу HTTP, подвержены дефектам ИБ, которые не приводят к отказам, однако уязвимы. В этом случае входные данные воспринимаются функциями ПО, как обычные, стандартные. К таким дефектам ИБ можно отнести SQL-инъекции (Structured Query Language Injections) [19], RCE (Remote Command Execution/Remote Code Execution) [19], LFI/RFI (Remote/Local File Inclusion) [19], IDOR (Insecure Direct Object Reference) [19] и другие. Из-за отсутствия обратной связи и возникает проблема выявления дефектов ИБ. Доработки в реализации обратной связи могут повлечь дополнительные расходы в организациях.

Еще одной проблемой могут быть ограничения операционных систем и их низкая производительность. Множество сервисов могут работать на виртуальных компонентах, например, такие как Docker-контейнеры [20]. Часто, для таких компонентов могут не создавать высоких вычислительных мощностей, а память (RAM, ROM) может быть сильно ограничена. Масштабирование систем влечет за собой увеличение расходов в организациях. Некоторые рудиментарные и устаревшие микросервисы могут работать на старых операционных системах, заменить или обновить, ввиду своей экономической и практической значимости, может быть невозможно из-за особенностей АС. Поэтому предлагается рассматривать вопрос фаззинга не с точки зрения подстройки систем, а с точки зрения настройки инструментов фаззинга для всех систем сразу в совокупности.

Также, еще одной проблемой может стать анализ полноты покрытия объектов фаззинга для качественной оценки проведенных работ. В рамках крупной организации – это может быть сложной и объемной задачей, поскольку это требует дополнительных ресурсов для проведения исследований ИБ применительно к каждому ПО или сервису, где потребуется глубокий разбор исходного кода и привлечение команд разработки.

Для решения вышеописанных проблем в рамках настоящей статьи предлагается проводить фаззинг-тестирование:

1) Сразу ко всей полиморфной микросервисной АС, в соответствии с бизнес-сценариями или функционалом, который доступен пользователю в соответствии с его ролью в системе. Критерием полноты покрытия считать количество задействованных API-функций и их параметров;
2) Использовать метод оценки временных интервалов ответов микросервисов на мутированные запросы, где начальное состояние данных мутации – полезная нагрузка (payloads) известных или типовых дефектов ИБ;
3) Проводить фаззинг-тестирование по методологии «черного ящика» или «серого ящика», преимущественно, используя модель внешнего нарушителя.

Такой подход оказался эффективным, что следует из нижеописанных результатов.

3. Анализ существующих решений для фаззинг-тестирования

В рамках настоящей статьи был проведен анализ существующих решений и методов, на предмет их применимости к микросервисным структурам.

В статье [21] приводится общий анализ различных решений, таких как: APIFuzzer [22], bBOXRT [23], Dredd [24], EvoMasterBB [25], RESTest [26], RESTler [27], RestTestGen [28], Schemathesis [29] и Tcases [30]. Для оценки эффективности существующих методов и инструментов тестирования API-функций, авторы статьи [21] проводили эмпирические наблюдения, в ходе которых применяли указанные инструменты к разным Web-сервисам. На основании полученных результатов авторы [21] отвечают на вопросы: какая полнота

покрытия у инструментов фаззинга и как много ошибок способны обнаружить рассматриваемые решения. Анализ этой работы показал, что сгенерированные параметры, которые подавались на вход функций исследуемых объектов имели в основном случайный характер. В результате этого, множество запросов не было правильно обработано сервисами. Авторы [21] использовали метод выявления 500 ошибок (по спецификации HTTP 500 ошибка – это 500 Internal Server Error, внутреняя ошибка, чаще всего отказ), которые могут свидетельствовать о наличии дефектов. Проведенные работы касались сервисов, которые функционировали локально и позволяли контролировать свое состояние и оценить покрытие [21]. Эти методы и инструменты могут, безусловно, использоваться в крупных организациях, но в своем составе не содержат функции оценки времени ответов. В работе авторов [21] не рассматривались вопросы авторизации в системах, а зависимости между микросервисами не были учтены. Соединения с сервисами устанавливались прямо и независимо друг от друга. Кроме того, не рассматривается последовательность отправки запросов, в соответствии с бизнес-сценариями и ролью пользователей.

В рамках текущей работы был проведен еще и анализ существующих открытых и коммерческих решений.

Были рассмотрены следующие открытые решения: honggfuzz [31], radamsa [32], AFL [33], LibFuzzer [34], oss-fuzz [35], sulley [36], boofuzz [37], Bfuzz [38], ffuf [39], wfuzz [40], nuclei [41]. В настоящей статье не проводится исследований эффективности этих инструментов в сравнении друг с другом, предполагается, что это будет описано в дальнейших работах, однако, каждый из инструментов был протестирован в рамках данной статьи на различных микросервисных АС. Описание и сравнение инструментов можно найти в статье [42], где описан анализ эффективности решений ffuf [39] и wfuzz [40], а также приводится описание большинства из вышеуказанных решений. Временная оценка ответов для вышеприведенных решений может быть произведена с помощью дополнительных настроек или с помощью написания специального ПО. Так, например, для инструмента nuclei время ответов можно получить с помощью следующих параметров запуска: *-ts, -timestamp*.

Указанные открытые решения можно использовать, как инструменты отправки запросов с полезной нагрузкой, например, используя списки мутированных последовательностей для фаззинга API-функций. Это может потребовать предварительной их генерации. Для генераций мутированных данных применительно к инструменту wfuzz, например, можно воспользоваться инструментом radamsa "Листинг 1":

```
for((i=1; i<100; i++));
do echo '<script>alert(1)</script>' | radamsa 1>> payloads.txt;
done && wfuzz -c -t 50 -w payloads.txt -u
https://service/api/v1/send?q=FUZZ
```

Листинг 1. Пример генерации мутированных данных с помощью radamsa для wfuzz
Listing 1. Mutated data generation using radamsa for wfuzz

Решения [31-40] содержат модули генерации случайных данных «на лету», когда в списках мутированных данных нет необходимости. Однако, простая подача на вход функций случайных данных с помощью генераторов не эффективна [7]. Большое множество генераций не позволяет проводить исследования за разумное время.

Для выполнения задач фаззинга может быть использовано ПО для тестирования на проникновение: Burp Suite [43], OWASP ZAP [44]. Однако, такие решения не обладают полноценными фаззинг-модулями и требуют использовать отдельные большие списки сгенерированных фаззинг-нагрузок или подключать дополнительные расширения. Модули детекции временных интервалов присутствуют в функционале этих инструментов, однако в них нет автоматического временного анализа и выявлять отклонения требуется эмпирически (вручную). Возникает дополнительная задача автоматизации процесса.

Существуют коммерческие решения для DAST-сканирований. В рамках текущей статьи не проводились исследования эффективности различных коммерческих решений по отношению друг к другу и в целом, а только приводится несколько примеров таких решений: PT BlackBox [45], Netsparker [46], appScreener [47], Acunetix [48]. Предполагается, что и они могут проводить фаззинг микросервисов. Вышеперечисленные решения могут быть эффективно встроены в процесс CI/CD [49], как DevSecOps [50] решения. Они могут использоваться на этапе разработки, тестирования или сопровождения, в соответствии с методологией SSDL [3]. Несмотря на то, что эти решения можно применять сразу ко всей микросервисной архитектуре, было выявлено, что при применении различных сканеров динамического анализа не обеспечивается полное покрытие АС проверками, поскольку сигнатурно, в составе таких решений, анализ производится в основном тестовыми запросами для поиска известных дефектов ИБ (CVE – Common Vulnerabilities and Exposures) [51]. Лучше всего они решают задачи анализа соответствия требований настроек систем (анализ настроек безопасности, выявление небезопасных функций), задачи по обнаружению устаревших версий ПО, выявлению слабостей используемых протоколов и обнаружению известных (1-day) дефектов ИБ [51]. Некоторые коммерческие решения имеют функции обнаружения DOS, когда задержки ответов сервера велики. Однако, автоматизированного модуля оценки временных интервалов нигде не содержится, а исследования по методологии «серого ящика» требуют созданий правил сканирования, когда необходимо создавать множество шаблонов и осуществлять их контроль, а это, в свою очередь, требует привлечения дополнительных ресурсов. Как правило, при применении коммерческих решений, отсутствует прозрачность в рамках того, какая именно полезная нагрузка была отправлена микросервису, так как производители предпочитают не разглашать свои технологии. Специалисту ИБ, как правило, будет доступен только отчет или электронная форма после проведения сканирований, которые будут не информативны при выполнения указанных задач в рамках фаззинг-тестирования.

В результате проведенного анализа литературы и апробации различных решений, в рамках текущей статьи делается следующий вывод: на текущий момент не существует универсальных решений для фаззинга сразу всех систем (микросервисов) с автоматизированным модулем оценки временных интервалом, контролем тестовой полезной нагрузки, модулем мутации известных дефектов ИБ. Требуется разработать решение, которое бы позволило выполнять все указанные задачи для фаззинг-тестирований.

4. Высокоуровневое описание объекта исследования

Для проведения исследований был использован тестовый стенд информационной сети, где функционировала полиморфная АС, в составе которой работало множество микросервисов (более 500), обрабатывающие различные типы данных с помощью API-функций. Они работали как с бинарными массивами данных, так и с типизированными, такие как JSON (JavaScript Object Notation) или XML (eXtensible Markup Language). При этом в тестовой среде велись непрерывно разработки ПО различными командами. Для проведения исследований был выбран сетевой протокол прикладного уровня – HTTP, как наиболее популярный. Исследуемая система является распределенной. Микросервисы могут быть связаны интеграциями. В качестве примера, на нижеприведенном рис.1, потоки данных показаны соединительными линиями. Расположение линий, как и подсистем может быть любым.

5. Описание процесса фаззинга

В рамках статьи, фаззинг проводился с помощью разработанного решения для указанных задач, которое состоит из модуля мутации, подстановки и детектирования, а процесс был разделен на несколько этапов.

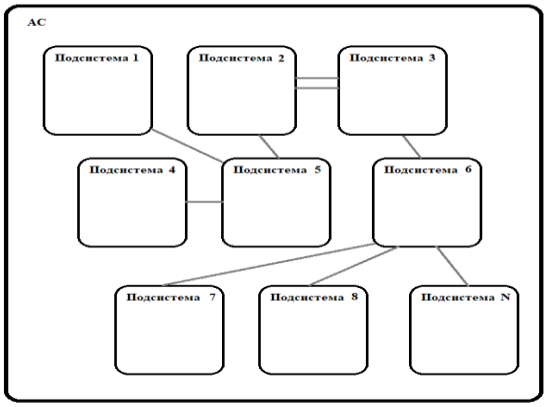

Рис. 1. Высокоуровневая схема полиморфной АС
Fig. 1. High-level scheme of polymorphic system

5.1 Сбор данных об АС и создание шаблонов тестирования

Сбор информации проводится по методологии «черного ящика» или «серого ящика». Результатом этого этапа является набор IP-адресов, DNS (Domain Name System) имен и сформированный список URI (Uniform Resource Identifier) для каждой конечной API функции. Для методологии «серого ящика», кроме того, формируется список всех запросов в рамках АС с параметрами и заголовками (см. Листинг 2).

```
POST /personal/api/v1/personal/setBalance HTTP/1.1
Host: ***
Cookie:
sid=111C5wM2LY4Sfc5LACOTAAAlNLABQtMjE4NzkwNjU1MDc3NjQ2NTk2OQACUzEAAjgz;
Content-Length: 73
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/100.0.4896.127 Safari/537.36
Content-type: text/plain
Accept: application/json, text/plain, */*
Connection: close

{"metadata":{"channel":"ib"},"data":{"balanceIsVisible":true},
"balanceSet":7788}


POST /personal/api/v1/personal/getBalance?query=first HTTP/1.1
Host: ***
Cookie: sid=111C5wM2LY4Sfc5LACOTAAAlNLABQtMjE4NzkwNjU1MDcsddsdk2OQACUzEAAjgz;
Content-Length: 93
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/100.0.4896.127 Safari/537.36
Content-type: text/plain
Accept: application/json, text/plain, */*
Connection: close

{"metadata":{"channel":"ib"},"data":{"balanceIsVisible":true},
"balanceSet":7788}
```

Листинг 2. Типовые шаблоны HTTP запросов
Listing 2. Typical templates of HTTP requests

Полученные исходные данные сохраняются в соответствующий шаблон в текстовом формате с эталонами всех HTTP-пакетов по спецификации микросервисов, где указываются все параметры в составе пакета для фаззинга. По умолчанию, таковыми являются URI, с разделителем “/”, заголовки протокола HTTP и данные (*data*, параметры и их значения).

5.2 Подготовка типовых тестовых проверок информационной безопасности

На этом этапе создаются списки различных полезных нагрузок для эксплуатации известных дефектов ИБ (CVE), например, такие как команды, выполняемые при работе с базами данных SQL, системные команды Linux, различные инъекции (вредоносные, ненадежные данные для системы, баз данных или кода), ссылки на ресурсы операционной системы в виде системных файлов и другие. Такой подход, как правило, применяется различными DAST-сканерами [43-48]. В итоге, формируются входные параметры в виде списков эталонных данных для тестирования и списка полезных нагрузок для модуля мутации, который является частью решения (см. Листинг 3).

```
%00../../../../../../etc/passwd
%00../../../../../../etc/shadow
-1.0
') or ('x'='x
0 or 1=1
' or 0=0 --
' UNION SELECT
t'exec master..xp_cmdshell 'nslookup localhost'--
%20$(sleep%2050)
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE foo [<!ELEMENT foo
ANY><!ENTITY xxe SYSTEM "file://c:/boot.ini">]><foo>&xxe;</foo>
{77*88}
```

Листинг 3. Пример списка значений полезных нагрузок для модуля мутации
Listing 3. Payloads examples for mutation

Указанный список полезных нагрузок будет использован в качестве параметра для модуля мутации. В основе работы модуля мутаций используются следующие функции изменения данных:

1) инверсия данных: случайным образом выбираются биты входных данных и их количество, затем над ними производится замена с 0 на 1 и наоборот;
2) сокращение данных или переполнение: сокращается количество переменных для параметров запросов или сокращаются длины передаваемых данных. Для переполнения используется подача на вход данных с длиной, превышающей допустимую или с помощью пошаговой инкременции, где каждое последующее тест-значение увеличивается по длине на 1 бит или 1 байт;
3) преобразуется числовой формат;
4) внесение интервалов: во входные данные добавляются множества пробелов, знаков форматирования, например, переходы на новую строку (“/n”, “%0a”), а также используется техника кодирования – Percent-encoding (добавления знаков % в URI или в других параметрах запросов, например в данных);
5) преобразование форматов: изменяется кодировка или производится сериализация объектов (преобразование объекта (параметра) в поток байтов (битов) для сохранения или передачи в память, базу данных или файл);
6) модификация строковых или числовых значений: строковые значения заменяются на типовые значения для обнаружения дефектов ИБ, при этом, изменяется расположение входных данных случайным образом в запросе. Числовые значения заменяются на отрицательные или на значения с плавающей точкой.
7) расширение данных: производится дублирование данных, повторение параметров запроса, случайное увеличение числа значений для одного и того же параметра;
8) рандомизация данных: на вход параметров запросов подаются случайные последовательности в требуемом формате или с модификациями формата;
9) конкатенация или сдвиг: производится объединение данных или линейный сдвиг на случайную величину.

Типы данных для каждого шаблона микросервисов могут отличатся. Ввиду этого реализована автоматическая подстройка мутатора к требуемому формату данных.

Также, в составе решения, реализован модуль подстановок. В основе модуля лежит алгоритм, который на основании шаблона и полезной нагрузки, генерирует последовательность вызова функций API в составе АС и регламентирует положение сгенерированной последовательности из модуля мутаций, без повторений. Примеры сгенерированных выходных данных для тестирования могут иметь вид, показанный на Листинге 4.

```
{"channel":"0%20'../../../etc/passwd.."}
{"channel":"inform,0+123"}
{"channel":"""""""""""""""}{""}
{"-dl-.echo'1';sleep 9999999":[]}
{"channel":"0%20'../../../etc/passwd.."}
GET /api/v1/service_n?%%%%%%%%sleep%%%2009999%30001
```

Листинг 4. Примеры сгенерированных выходных данных для тестирования
Listing 4. Examples of generated output for testing

Для контроля состояния тестирования отвечает модуль детектирования, который сохраняет полученные результаты фаззинг-тестов, автоматически анализируя следующее:

1) аномальные ответы сервера;
2) обрабатываемые ошибки;
3) интервалы времени между запросом и ответом;
4) статусы ответов;
5) падение (DOS – Denial of Service) подсистемы;
6) наличие типовых дефектов ИБ;
7) сигнатуру ответа.

При проведении фаззинг-тестов какой-либо API-функции, применяется метод вызова связанных цепочек данных, когда выходные мутированные данные снова подаются на вход модуля мутации. Применение такого конечного автомата обеспечивает более полное покрытие [7]. Если в результате какого-либо теста было получено время ответа, которое отличается от предыдущих запросов для исследуемой API-функции, то производится отправка стандартного запроса, без полезных нагрузок, а затем повторяется мутированный запрос, для которого была замечена аномалия. Если отклонение повторяется, то такая нагрузка помечается, как потенциальный дефект ИБ.

Количество мутаций выходных данных задается с помощью параметра глубины мутации (повторений мутации) – *J*. Одна мутация равна одному тесту. Чем больше глубина мутаций и, соответственное им количество тестов, тем большая вероятность обнаружения дефектов. Также можно настроить максимум *J,* чтобы данные параметров не имели слишком случайный характер. Реакция ошибок кода микросервисов для описываемого метода не учитывается из-за ограничений обратной связи. В целом, предлагаемый метод фаззинга можно описать схемой, показанной на рис. 2.

В процессе фаззинга именно модулем детектирования происходит измерение времени ответа для каждого запроса к АС. Собираются все значения временных интервалов для каждой API функции всех подсистем. В ходе анализа такой статистики было замечено, что при отправке к системе некоторых мутированных запросов, содержащих полезную нагрузку, ряд значений времени *T* увеличены по сравнению с другими запросами. Автором статьи была выдвинута гипотеза о том, что такое поведение связано с тем, что микросервисы используют дополнительный ресурс для обработки ответов в виде выполнения системных команд, чтения файлов или выполнения иных математических операций, ввиду чего и возникает временная задержка. Также, автором статьи была выдвинута и другая гипотеза о том, что такое поведение может быть связано с наличием дефектов ИБ в микросервисах.

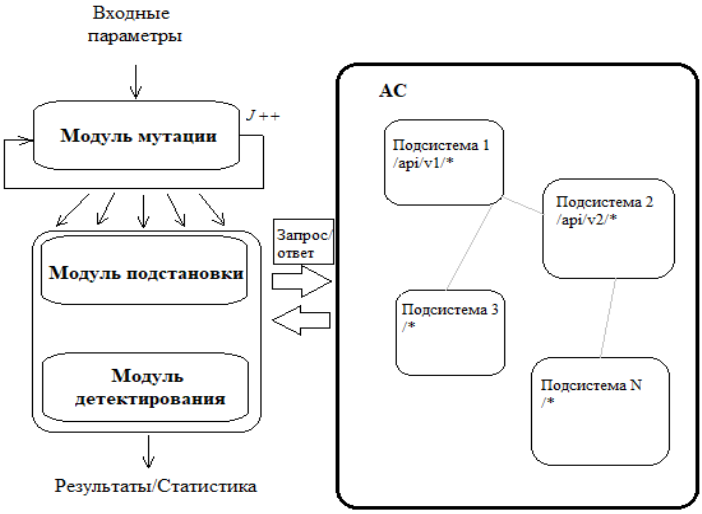

Рис. 2. Высокоуровневая схема процесса фаззинга
Fig. 2. High-level scheme of fuzzing process

Для проверки гипотез в модуле детектирования был реализован следующий алгоритм:

1) для каждой полезной нагрузки в составе передаваемого параметра на API функцию для соответствующей глубины J сохраняется время ответа T системы на запрос;
2) собирается и анализируется статистика задержек запросов/ответов в виде:

$$\Delta TJn = T1 + T2 + T3 + \ldots + Tn\ ;$$

3) если ΔTJn - ΔTJn-1 < 0, то такой результат тестирования в виде сигнатуры HTTP запроса и ответа сохраняется, как гипотетически содержащий дефект.

Вышеприведенная схема была реализована автором в составе программы, написанной на языке Golang, которая не использует в своем составе открытые (Open Source) решения или библиотеки. Это позволило избавиться от зависимостей. Для отправки запросов использовался стандартный пакет – *net/http*. Для достижения целей фаззинг-тестирования в организациях могут использоваться описанные выше открытые, коммерческие решения или могут создаваться собственные инструменты командами разработки, где будут реализованы функции детектирования и анализа времени ответов микросервисов. В рамках описанных задач автором рекомендуется использовать за основу открытое решение nuclei [41], но при этом потребуется разработать модуль анализа полученных результатов с особенностями, применительно к различным организациям и их микросервисам.

6. Анализ полученных результатов и выводы

Результатом работы решения является собранное множество запросов (HTTP-пакетов) и ответов на эти запросы, полученные от АС. Важно отметить, что модуль детектирования не сохраняет одинаковые ответы системы, а только те, которые отличаются от эталонных или те, которые не были обнаружены ранее. Отдельно сохраняются запросы, для которых время ответов было аномальным (большим). Такой подход позволяет сократить размер результата-листинга, который может быть большим в зависимости от глубины *J*. Происходит сравнение каждого нового ответа с каждым сохраненным, пока не будет получен уникальный результат. В качестве параметра сохранения результата, для обеспечения скорости обработки данных, была выбрана в том числе и длина ответа. Такие методы фильтрации результатов обеспечивают минимизацию ложноположительных тестов. В целом результаты проведенных

работ собраны в табл. 1, полученной при тестировании микросервисной АС в тестовой среде (см. раздел 4).

Табл. 1. Результаты проведенных фаззинг-тестов
Table1. The results of the fuzzing tests

Количество микросервисов	Количество API функций	Глубина мутации	Время фаззинг-тестирования	Количество уникальных ответов	Количество уникальных временных задержек
1	5	10	~ 5 минут	1	1
10	93	30	~ 60 минут	2	2
100	1068	20	~ 500 минут	11	13
200	3015	10	~ 3000 минут	37	23

Конечно, приведенные в табл. 1 данные субъективны и значения будут сильно зависеть от информационной архитектуры различных организаций. Поэтому при применении описываемого подхода, требуется эмпирически рассчитать среднее время обработки запросов и разработать критерии анализа уникальных ответов. Иногда, возникают случаи, когда микросервис из-за обновлений, влияния других процессов, высокой нагрузки может выдавать ложноположительные результаты, в виде ошибок (500 статус) или в виде высоких временных задержек на ответы, поэтому все подобные результаты фаззинг-тестирования проходят повторное сканирование. После проведения работы для исследуемой АС, набор гипотетических дефектов анализируется в рамках одного микросервиса, где применяется такой же набор входных параметров на этапе *J*, при которых была обнаружена задержка: проводится направленный повторный одиночный тест для того, чтобы опровергнуть или подтвердить наличие дефектов.

В результате проведения исследования, было получено, что временные задержки – действительно обусловлены тем, что в ряде систем содержатся потенциально опасные дефекты ИБ. Поэтому возникают отказы или время ответа возрастает по сравнению с другими ответами. Несколько таких тестовых примеров, которые действительно являлись на момент исследования дефектами ИБ, приведены в табл. 2. Конечно, по этическим соображениям, в рамках статьи, некоторые данные об объекте исследования были изменены.

Табл. 2. Примеры запросов с полезной нагрузкой, ответов и время ответов
Table 2. Requests examples with payloads, answers and response time

№	Пример HTTP запроса (тест)	Пример ответа на запрос	Время ответа (миллисекунды)
1	GET /aa/settings/?sender=user&pl=web HTTP/2 Host: app.mobile.xxxxxx.com:8899 Content-Type: text/plain; charset=utf-8 Content-Length: 50 {"data":"input", "command": "; /n ping 127.0.0.1"}	HTTP/2 200 OK Server: nginx/1.20.2 Date: Tue, 17 Oct 2023 20:18:30 GMT Content-Type: application/json Content-Length: 91 {"data":"message", "print": "Ошибка! Проверьте входные данные."}	6413 мс
2	POST /api/v2/change/?%20%0a'%2d%2dselect%20*from%20users HTTP/2 Host: app.mobile.xxxxxx.com:8899	HTTP/1.1 500 Internal Server Error Server: Apache/2.4.25	5422 мс
3	POST /s1/trace?start=%7dAAAAAAAA%7b HTTP/2 Host: app.mobile.xxxxxx.com:8899	Нет ответа	-

На рис. 3, рис. 4 и рис. 5 приведен вывод результатов для запросов 1, 2 и 3 из табл. 2, соответственно. Как видно из рисунков, ответы на запросы, которые не содержали дефектов, имеют меньшие величины времени ответа *T*. Также следует заметить, что статус или текст

ответа микросервисов, не всегда является корректным и явно не свидетельствует о наличии в АС какого-либо дефекта ИБ.

```
app.mobile.xxxxxx.com | port: 8899 |GET  |ping app.mobile.xxxxxx.com" | --->   HTTP/2 200 OK  TIME: 0430 ms
app.mobile.xxxxxx.com | port: 8899 |GET  |ping 127.0.0.1 --->   HTTP/2 200 OK  TIME: 0223 ms
app.mobile.xxxxxx.com | port: 8899 |GET  |/n ping 127.0.0.1 --->   HTTP/2 200 OK  TIME: 0321 ms
app.mobile.xxxxxx.com | port: 8899 |GET  |%0a ping 127.0.0.1 --->   HTTP/2 200 OK  TIME: 0050 ms
app.mobile.xxxxxx.com | port: 8899 |GET  |;%20%0a%20ping%20127.0.0.1 --->   HTTP/2 200 OK  TIME: 0131 ms
app.mobile.xxxxxx.com | port: 8899 |GET  |; /n ping 127.0.0.1 --->   HTTP/2 200 OK  TIME: 6413 ms
app.mobile.xxxxxx.com | port: 8899 |GET  |%20%0a%20ping%20127.0.0.1 --->   HTTP/2 200 OK  TIME: 0896 ms
app.mobile.xxxxxx.com | port: 8899 |GET  |%20%0a%20ping%20127%2e0%2e0%2e1 --->   HTTP/2 200 OK  TIME: 0761 ms
app.mobile.xxxxxx.com | port: 8899 |GET  |%20%0a%20ping%20127%2e0%2e0%2e1 --->   HTTP/2 200 OK  TIME: 0277 ms
```

Рис. 3. Пример № 1 обнаружения дефекта ИБ среди множества запросов
Fig. 3. Example No. 1 of detecting a defect among many requests

```
app.mobile.xxxxxx.com | port: 8899 |POST  |select * from users --->   HTTP/2 400 Bad Request TIME: 0322 ms
app.mobile.xxxxxx.com | port: 8899 |POST  |--select * from users --->   HTTP/2 400 Bad Request  TIME: 0901 ms
app.mobile.xxxxxx.com | port: 8899 |POST  |`--select * from users --->   HTTP/2 400 Bad Request  TIME: 0895 ms
app.mobile.xxxxxx.com | port: 8899 |POST  |%20%0a%19%2d%2dselect%20*from%20users --->   HTTP/1.1 500 Internal Server Error  TIME: 5422 ms
app.mobile.xxxxxx.com | port: 8899 |POST  |%3b%20%0a%19%2d%2dselect%20*from%20users --->   HTTP/1.1 500 Internal Server Error  TIME: 0231 ms
app.mobile.xxxxxx.com | port: 8899 |POST  |%3b%20%0a%19%2d%2d%73%65%6c%65%63%74%20%2a%66%72%6f%6d%20%75%73%65%72%73 --->   HTTP/1.1 500 Internal Server Error  TIME: 0190 ms
app.mobile.xxxxxx.com | port: 8899 |POST  |%0a%3b%20%0a%19%2d%2d%73%65%6c%65%63%74%20%2a%66%72%6f%6d%20%75%73%65%72%73 --->   HTTP/1.1 500 Internal Server Error  TIME: 0781 ms
```

Рис. 4. Пример № 2 обнаружения дефекта ИБ среди множества запросов
Fig. 4. Example No. 2 of detecting a defect among many requests

```
app.mobile.xxxxxx.com | port: 8899 |POST  |AAAAAAAAA  --->   HTTP/2 400 Bad Request  TIME: 0238 ms
app.mobile.xxxxxx.com | port: 8899 |POST  |AAAAAAAA ->(33) AAAAAAAAAAAAAAAAAAAA --->   HTTP/2 200 OK  TIME: 0563 ms
app.mobile.xxxxxx.com | port: 8899 |POST  |{} --->   HTTP/2 400 Bad Request  TIME: 0322 ms
app.mobile.xxxxxx.com | port: 8899 |POST  |{AAAAAAAAA} --->   HTTP/2 400 Bad Request  TIME: 0199 ms
app.mobile.xxxxxx.com | port: 8899 |POST  |%7dAAAAAAAAA%7b --->   null  TIME: 99999 ms
app.mobile.xxxxxx.com | port: 8899 | | --->   null >>>>>>>>>>DoS Detected<<<<<<<<<
```

Рис. 5. Пример № 3 обнаружения дефекта ИБ среди множества запросов
Fig. 5. Example No. 3 of detecting a defect among many requests

В результате, было установлено, что для каждого типа дефекта ИБ соответствуют разные ΔT. Например, для дефектов типа DoS – ΔT, как правило, принимает самое большое значение, а самые низкие показатели ΔT обнаруживались для дефектов типа IDOR (Insecure direct object reference).

При применении разработанного решения фаззинг-тестирования для исследуемой микросервисной АС было получено следующее:

- отсутствует большая нагрузка на тестируемые микросервисы;
- эффективность разработки и время не снижается, несмотря на то что на тестовом контуре постоянно проводились работы различными командами;
- обеспечивается высокое покрытие микросервисов, поскольку используется универсальный шаблон, применяемый ко всей АС сразу, где исследуется каждая API функция по спецификации самого сервиса;
- автоматизированно производится анализ ответов сервера и их временных интервалов для каждой сгенерированной последовательности, переданной на вход API функции, что сокращает время при ручном (эмпирическом) анализе;
- производится автоматическая настройка в соответствии с протоколом сервера и типом обрабатываемой информации, что позволяет производить проверки непрерывно, в том числе в рамках процесса CI/CD [49];
- нет привязки к формату данных, а тип данных определяется на основании анализа данных из шаблонов запросов/ответов на всех этапах тестирования, что позволяет не тратить время и ресурсы на запуск инструментов фаззинга применительно к каждому микросервису в отдельности.

7. Заключение

В рамках настоящей статьи было установлено, что различные API в структурах микросервисов, могут содержать различные типы дефектов ИБ и ошибок. Для их обнаружения необходимо проводить фаззинг-тестирование, в соответствии с требованиями регуляторов и рекомендациями стандартов. При проведении фаззинг-тестирований отдельно, для каждого микросервиса могут возникать проблемы, связанные с соблюдением сроков разработки и снижением эффективности в крупных организациях. Описанный в настоящей статье метод фаззинг-тестирования для выявления дефектов ИБ, основанный на оценке временных интервалов ответов различных микросервисов, подаче мутированных данных полезных нагрузок на вход API-функций, при использовании методологий «серого» или «черного ящика» - оказался эффективным. При таком подходе нарушений сроков разработки не выявлено. Также, оказалось эффективным и применение процедуры фаззинга сразу ко всей микросервисной структуре в рамках АС, где обеспечивается полное покрытие всех доступных API-функций. Предлагаемый метод и решение сегодня применяется, как один из вариантов проведения фаззинг-тестирования в крупной организации [52]. Сам метод позволил выявить следующие типы дефектов ИБ:

- RCE (Remote Command Execution/Remote Code Execution);
- LFI/RFI (Remote/Local file inclusion);
- SQL-инъекции;
- DoS (Denial of Service);
- IDOR (Insecure direct object reference);
- XXE (external entity injection);
- ошибки логики и ряд других.

Описанный в статье метод рекомендуется применять в крупных организациях, где используется микросервисная структура, как дополнительное или основное решение по обеспечению информационной безопасности для того, чтобы предотвращать критичные отказы инфраструктуры и финансовые потери.

Список литературы / Reterences

[1]. Ниньо-Мартинес В., Очаран-Эрнандес Х., Лимон К., Перес-Арригата Х. Развертывание микросервисов. Труды Института системного программирования РАН. 2023;35(1):57-72. DOI: 10.15514/ISPRAS-2023-35(1)-4.

[2]. Вальдивия Х., Лора-Гонсалес А., Лимон К., Кортес-Вердин К., Очаран-Эрнандес Х. Паттерны микросервисной архитектуры: многопрофильный обзор литературы. Труды Института системного программирования РАН. 2021;33(1):81-96. DOI: 10.15514/ISPRAS-2021-33(1)-6.

[3]. Umeugo, Wisdom. (2023). Secure software development lifecycle: a case for adoption in software smes. International Journal of Advanced Research in Computer Science. 14. 5-12. 10.26483/ijarcs.v14i1.6949.

[4]. Li J., Li J., Zhao B., Zhang C. Fuzzing: a survey // Cybersecurity, 2018, Vol. 1, No 1, p. 6, DOI: 10.1186/s42400-018-0002-y.

[5]. Методика динамического сканирования приложений. DAST. Available at: https://owasp.org/www-project-devsecops-guideline/latest/02b-Dynamic-Application-Security-Testing, accessed 04.01.2024.

[6]. Шарков И.В., Падарян В.А., Хенкин П.В. Об особенностях фаззинг-тестирования сетевых интерфейсов в условиях отсутствия исходных текстов. Труды Института системного программирования РАН. 2021;33(4):211-226. DOI: 10.15514/ISPRAS-2021-33(4)-15.

[7]. Саргсян С.С., Варданян В.Г., Акопян Д.А., Агабалян А.М., Меграбян М.С., Курмангалеев Ш.Ф., Герасимов А.Ю., Ермаков М.К., Вартанов С.П. Платформа автоматического фаззинга программного интерфейса приложений. Труды Института системного программирования РАН. 2020;32(2):161-173. DOI: 10.15514/ISPRAS-2020-32(2)-13.
[8]. ISA/IEC 62443-4-1. Available at: https://www.isa.org/standards-and-publications/isa-standards/isa-iec-62443-series-of-standards, accessed 04.01.2024.
[9]. ISO/IEC/IEEE 291119. Available at: https://cdn.standards.iteh.ai/samples/81291/6694557ff8304df8841bb191a00ecc6f/ISO-IEC-IEEE-29119-1-2022.pdf, accessed 04.01.2024.
[10]. ISO 27001. Available at: https://www.iso.org/standard/27001, accessed 04.01.2024.
[11]. ГОСТ Р 56939-2016. Разработка безопасного программного обеспечения. Общие требования. Дата введения 2017-06-01.
[12]. Методический документ. "Методика оценки угроз безопасности информации". Утвержден ФСТЭК России. Москва. 5 февраля 2021 г.
[13]. ГОСТ Р 58143-2018. Информационная технология. Методы и средства обеспечения безопасности. Детализация анализа уязвимостей программного обеспечения в соответствии с ГОСТ Р ИСО/МЭК 15408 и ГОСТ Р ИСО/МЭК 18045. Часть 2. Тестирование проникновения. ОКС 35.020. Дата введения 2018-11-01.
[14]. The Swagger API project. Apache License 2.0. Available at: https://swagger.io, accessed 04.01.2024.
[15]. OpenAPI Specification v3.1.0. Published 15 February 2021. Available at: https://spec.openapis.org/oas/latest.html, accessed 04.01.2024.
[16]. OWASP Top-10. 2024. Available at: https://owasp.org/www-project-top-ten/, accessed 04.01.2024.
[17]. OWASP. Fuzz Vectors. 2024. Available at: https://owasp.org/www-project-web-security-testing-guide/stable/6-Appendix/C-Fuzz_Vectors, accessed 04.01.2024.
[18]. Software Development Life Cycle (SDLC) Methodologies for Information Systems Project Management - Mohammad Ikbal Hossain - IJFMR Volume 5, Issue 5, September-October 2023. DOI: 10.36948/ijfmr.2023.v05i05.6223.
[19]. Payloads All The Things. Web Application Security, Pentest and Red Team Cheatsheet. 2023. Available at: https://swisskyrepo.github.io/Payloads-AllTheThings/, accessed 04.01.2024.
[20]. Docker. Available at: https://www.docker.com/, accessed 04.01.2024.
[21]. Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated Test Generation for REST APIs: No Time to Rest Yet. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea. ACM, New York, NY, USA, 13 pages. DOI: 10.1145/3533767.3534401.
[22]. APIFuzzer. Available at: https://github.com/KissPeter/APIFuzzer, accessed 04.01.2024.
[23]. Laranjeiro, Nuno & Agnelo, João & Bernardino, Jorge. (2021). A Black Box Tool for Robustness Testing of REST Services. IEEE Access. PP. 1-1. DOI: 10.1109/ACCESS.2021.3056505.
[24]. Dredd. Available at: https://github.com/apiaryio/dredd, accessed 04.01.2024.
[25]. Andrea Arcuri. 2020. Automated Black-and White-Box Testing of RESTful APIs With EvoMaster. IEEE Software 38, 3 (2020), 72–78. DOI: https://doi.org/10.1145/3293455.
[26]. Martin-Lopez, Alberto & Segura, Sergio & Ruiz-Cortés, Antonio. (2020). RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs. 459-475. DOI: 10.1007/978-3-030-65310-1_33.
[27]. Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: Stateful rest api fuzzing. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, Montreal, QC, Canada, 748–758. DOI: 10.1109/ICSE.2019.00083.
[28]. Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RestTestGen: automated black-box testing of RESTful APIs. In 2020 IEEE 13th International Conference on Software

Testing, Validation and Verification (ICST). IEEE, 142–152. DOI: 10.1109/ICST46399.2020.00024.
[29]. Zac Hatfield-Dodds and Dmitry Dygalo. 2021. Deriving Semantics-Aware Fuzzers from Web API Schemas. arXiv preprint arXiv:2112.10328 (2021). Available at: https://www.researchgate.net/publication/357202018_Deriving_Semantics-Aware_Fuzzers_from_Web_API_Schemas.
[30]. tcases REST API tool. Available at: https://github.com/Cornutum/tcases/tree/master/tcases-openapi, accessed 02.01.2024.
[31]. honggfuzz. Available at: https://honggfuzz.dev/, accessed 02.01.2024.
[32]. radamsa. Available at: https://gitlab.com/akihe/radamsa, accessed 02.01.2024.
[33]. AFL. Available at: https://github.com/google/AFL, accessed 02.01.2024.
[34]. LibFuzzer. Available at: https://llvm.org/docs/LibFuzzer.html, accessed 02.01.2024.
[35]. oss-fuzz. Available at: https://github.com/google/oss-fuzz, accessed 02.01.2024.
[36]. sulley. Available at: https://github.com/OpenRCE/sulley, accessed 02.01.2024.
[37]. boofuzz. Available at: https://github.com/jtpereyda/boofuzz, accessed 02.01.2024.
[38]. Bfuzz. Available at: https://github.com/RootUp/Bfuzz, accessed 02.01.2024.
[39]. ffuf. Available at: https://github.com/ffuf/ffuf, accessed 02.01.2024.
[40]. wfuzz. Available at: https://github.com/xmendez/wfuzz, accessed 02.01.2024.
[41]. nuclei. Available at: https://github.com/projectdiscovery/nuclei, accessed 02.01.2024.
[42]. Matheos Mattsson 40476. Master Thesis in Computer Engineering. Supervisor: Dragos Truscan. Faculty of Science and Engineering. Åbo Akademi University. 2021. A comparison of FFUF and Wfuzz for fuzz testing web applications. Available at: https://www.doria.fi/bitstream/handle/10024/181265/mattsson_matheos.pdf, accessed 07.01.2024.
[43]. Burp Suite. Available at: https://portswigger.net/burp/pro, accessed 05.01.2024.
[44]. OWASP ZAP. Available at: https://www.zaproxy.org/, accessed 05.01.2024.
[45]. PT BlackBox. Available at: https://www.ptsecurity.com/ru-ru/products/blackbox/, accessed 02.01.2024.
[46]. Netsparker. Available at: https://github.com/netsparker, accessed 03.01.2024.
[47]. appScreener. Available at: https://rt-solar.ru/products/solar_appscreener/, accessed 02.01.2024.
[48]. Acunetix. Available at: https://www.acceron.net/index.php/products/acunetix, accessed 02.01.2024.
[49]. Jeffrey Fairbanks, Akshharaa Tharigonda, Nasir U. Eisty. Analyzing the Effects of CI/CD on Open Source Repositories in GitHub and GitLab. 2023. https://doi.org/10.48550/arXiv.2303.16393.
[50]. Myrbakken, Håvard & Colomo-Palacios, Ricardo. (2017). DevSecOps: A Multivocal Literature Review. 17-29. DOI: 10.1007/978-3-319-67383-7_2.
[51]. База данных общеизвестных уязвимостей информационной безопасности. Available at: https://cve.mitre.org/, accessed 03.01.2024.
[52]. АО Газпромбанк, https://www.gazprombank.ru/, accessed 03.01.2024.

Информация об авторах / Information about authors

Артемий Сергеевич ЮРЬЕВ – исполнительный директор, департамент развития технологий защиты информации, АО Газпромбанк, аспирант ИСП РАН. Научные интересы: информационная безопасность, фаззинг информационных систем, анализ защищенности, тестирование на проникновение, динамическое сканирование, безопасная разработка.

Artemiy Sergeevich YUREV – executive director, Department of Development Information Security Technologies, Gazprombank (JSC), postgraduate student of ISP RAS. Research interests: information security, fuzzing of information systems, security analysis, penetration testing, DAST, SSDL.