На правах рукописи

Сигалов Даниил Алексеевич

Методы выявления поверхности атаки веб-приложений при помощи анализа клиентского JavaScript-кода

Специальность 2.3.5 —
«Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей»

Автореферат
диссертации на соискание учёной степени
кандидата технических наук

Москва — 2025

Работа выполнена на кафедре информационной безопасности факультета Вычислительной математики и кибернетики ФГБОУ ВО “Московский государственный университет имени М. В. Ломоносова”.

Научный руководитель: кандидат физико-математических наук
Гамаюнов Денис Юрьевич

Официальные оппоненты: **Мазин Анатолий Викторович,**
доктор технических наук, профессор,
заведующий кафедрой защиты информации Калужского филиала федерального государственного бюджетного образовательного учреждения высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»

Курмангалеев Шамиль Фаимович,
кандидат физико-математических наук,
ведущий научный сотрудник отдела компиляторных технологий Федерального государственного бюджетного учреждения науки Институт системного программирования им.В.П.Иванникова Российской академии наук

Ведущая организация: Федеральное государственное бюджетное образовательное учреждение высшего образования «МИРЭА - Российский технологический университет»

Защита состоится 9 октября 2025 г. в 12:00 на заседании диссертационного совета 24.1.120.01 при Федеральном государственном бюджетном учреждении науки Институте системного программирования им. В. П. Иванникова Российской академии наук по адресу: 109004, г. Москва, ул. А. Солженицына, д. 25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки Института системного программирования им. В. П. Иванникова Российской академии наук.

Автореферат разослан «___»_____ 2025 года.

Ученый секретарь
диссертационного совета
24.1.120.01,
кандидат физико-математических наук Турдаков Д. Ю.

Общая характеристика работы

Актуальность темы. В настоящее время операционная деятельность организаций, как коммерческих компаний, так и государственных учреждений, существенно зависит от интернета и интернет-приложений. Банковские услуги, интернет-магазины, средства организации рабочего процесса, социальные сети, сервисы отправки показаний счётчиков воды и электричества — в значительной степени реализуются с помощью веб-приложений. При этом, по данным компании Verizon (одна из крупнейших телекоммуникационных компаний США), до 80% инцидентов безопасности в 2023 году, приведших к утечке данных пользователей или коммерческой тайны, были связаны с атаками на уязвимые веб-приложения. На этом фоне чрезвычайно важным является поддержание безопасности веб-приложений, для чего необходимо своевременное обнаружение и устранение уязвимостей в них. Крайне опасными являются, например, уязвимости таких классов, как “SQL-инъекция”, “небезопасная загрузка файлов”, “Server-side template injection” и “Shell injection”. SQL-инъекции могут позволить атакующему получить полный доступ к базе данных приложения, остальные перечисленные уязвимости могут дать возможность выполнить произвольный код на атакуемом сервере. Эксплуатация этих уязвимостей может повлечь кражу пользовательских данных, потерю данных, сбои в работе сервисов.

В данной работе рассматривается задача автоматического обнаружения серверных уязвимостей в веб-приложениях в модели “черного ящика”, когда для анализа доступна только клиентская часть веб-приложения и интерфейсные точки взаимодействия на серверной стороне веб-приложения. Поиск уязвимостей в этой модели востребован, поскольку такие условия близки к условиям, в которых действует реальный атакующий. Кроме того, предоставление исходного кода приложения для анализа может быть нежелательно из соображений конфиденциальности. В модели “черного ящика” процесс поиска серверных уязвимостей обычно состоит из трех этапов:

1. Выявление поверхности атаки — поиск *серверных входных точек* (их также называют доступными функциями веб-приложения, точками входа в приложение или API-вызовами).
2. Вызов найденных функций приложения с различными значениями параметров.
3. Анализ результатов выполнения действий.

Данная работа посвящена автоматизации выявления поверхности атаки. Одним из главных источников информации о серверных входных точках является клиентская часть веб-приложения — то есть его пользовательский интерфейс.

Выявление набора серверных входных точек на основе клиентской части веб-приложения сводят к выявлению набора запросов, которые могут быть отправлены клиентской стороной на сервер. Множество доступных функций сервера полностью определяется кодом серверной части приложения и его конфигурацией. При анализе веб-приложения как черного ящика эта информация

недоступна, поэтому восстановить это множество полно и точно не представляется возможным, но можно решать задачу построения аппроксимирующего множества. Выдвигается гипотеза о том, что отправляемые с клиентской части запросы соответствуют серверным входным точкам (код серверной и клиентской части согласован).

Выявление набора отправляемых на сервер запросов тривиально для статических интерфейсов (построенных с использованием только языка HTML). Семантика элементов HTML-разметки и их реакция на действия пользователя всегда фиксированы. Однако для динамических интерфейсов (построенных с использованием JavaScript-кода) задача намного сложнее. JavaScript — это высокоуровневый язык программирования, полный по Тьюрингу. На HTML-странице JavaScript-код может инициировать отправку запроса на сервер, URL-адрес и другие части которого сформированы самим кодом. По статистике, JavaScript используется на 98,8% веб-сайтов.

Исследования, посвящённые автоматическому выявлению набора серверных входных точек посредством анализа клиентской части веб-приложения, включая анализ клиентского JavaScript-кода, ведутся рядом научных групп. Недавними являются публикации исследователей из австралийского отделения Oracle Labs, в которых были представлены средства BackREST и Gelato, а также статьи группы из австрийского исследовательского центра SBA Research, в которых описаны инструменты XIEv и CHIEv. Работы по этой теме публиковались совместной группой из Университета Британской Колумбии и Делфтского технического университета. Все предложенные на данный момент методы используют динамический анализ, а именно — *динамический краулинг*, автоматизированное взаимодействие с элементами интерфейса на веб-странице с помощью управляемого браузера (симуляция пользовательских действий) и запись всех отправляемых со страницы запросов на сервер.

У этого метода есть ограничения. В некоторых случаях пользовательский интерфейс слишком сложен для того, чтобы осуществить все возможные действия с ним. Перебор всех комбинаций возможных действий может требовать слишком большого количества времени. Более того, существуют случаи, когда JavaScript-код, отправляющий запрос к определённой серверной входной точке, вообще невозможно вызвать действиями в интерфейсе — по сути, это недостижимый код. Такой код, тем не менее, представляет интерес при поиске уязвимостей: часть серверного кода, к которому делается такое обращение, может работать.

Таким образом, актуальной является задача разработки метода выявления серверных входных точек, основанного на *статическом* анализе клиентского JavaScript-кода.

Целью данной работы является разработка метода автоматического выявления поверхности атаки веб-приложений с динамической клиентской частью, реализованной на языке программирования JavaScript.

Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. Провести исследование особенностей реального JavaScript-кода, влияющих на возможность его анализа с целью обнаружения информации о серверных входных точках. На основе результатов проведённого исследования сформулировать требования к инструментам построения поверхности атаки на основе статического анализа клиентского JavaScript-кода, а также составить эталонный набор веб-страниц (бенчмарк), позволяющий оценивать эффективность методов извлечения информации о серверных входных точках из клиентского кода.
2. Разработать методику поиска уязвимостей в режиме “чёрного ящика”, позволяющую обрабатывать как легко доступные из пользовательского интерфейса серверные входные точки, так и труднодоступные.
3. Разработать и реализовать специализированный метод статического анализа клиентского кода веб-приложения для обнаружения серверных входных точек, учитывающий выявленные в ходе проведённого исследования особенности реального кода. Провести эксперименты с разработанной реализацией на реальных данных.

Научная новизна: В работе получены следующие результаты, обладающие научной новизной.

1. Проведено исследование реального кода JavaScript-приложений и выделены его наиболее существенные особенности с точки зрения статического анализа для поиска входных точек: использование упаковщиков модулей, использование непрямого объявления класса, обращение к полю объекта по вычисленному имени и другие. Учет выявленных особенностей позволил сузить задачу статического анализа и предложить более эффективный метод его выполнения.
2. Предложена методика поиска уязвимостей веб-приложений в модели “чёрного ящика”. Методика отличается от существующих повышением покрытия серверных входных точек, обращения к которым сложно вызвать через взаимодействие с пользовательским интерфейсом, за счет применения статического анализа и автоматизированного перебора имён параметров запроса, а также применением анализа кода клиентской стороны веб-приложения для обнаружения уязвимостей.
3. Предложен метод обнаружения информации о серверных входных точках для задачи анализа защищенности приложений методом “черного ящика” с помощью статического анализа клиентского кода. Для типичной страницы реального веб-приложения разработанный алгоритм анализа позволяет решить задачу обнаружения входных точек за несколько минут, что делает его пригодным для практического использования. Это достигается за счёт ограничения числа проходов по коду и добавления встроенной поддержки популярных JavaScript-библиотек, отправляющих запросы на сервер (позволяет не анализировать собственный код

библиотек). Разработанный алгоритм анализирует весь код страницы, включая недостижимый код.

Практическая значимость работы состоит в возможности применения её результатов в анализе защищенности реальных систем: метод позволяет более полно выявлять элементы поверхности атаки — анализируемые на наличие уязвимостей серверные входные точки, тем самым увеличивая количество обнаруживаемых уязвимостей. Реализация предложенного метода была внедрена как часть системы автоматического поиска уязвимостей в веб-приложениях на основе обработки больших данных, разработанной в Центре компетенций НТИ по технологиям хранения и анализа больших данных МГУ (ЦХАБД МГУ) [1]. Также разработанный метод внедрён в систему автоматизированного поиска уязвимостей веб-приложений SolidPoint DAST, разрабатываемую компанией ООО «СолидСофт». Акты о внедрении приведены в приложении А к диссертационной работе.

Методология и методы исследования.

В работе использовались методы статического анализа программ и теории компиляции.

Основные положения, выносимые на защиту:

1. Требования к инструментам построения поверхности атаки на основе статического анализа клиентского JavaScript-кода, которые сформулированы на основе результатов исследования особенностей реального JavaScript-кода, влияющих на возможность его анализа для поиска серверных входных точек. Эталонный набор веб-страниц (бенчмарк), также созданный в результате этих исследований, который может быть использован для автоматизированной оценки эффективности методов извлечения информации о серверных входных точках.
2. Методика обнаружения уязвимостей в веб-приложениях, которая повышает полноту выявления серверных входных точек за счет статического анализа клиентской части приложения. Методика апробирована в реальных системах автоматизированного анализа защищенности приложений.
3. Специализированный метод анализа клиентского кода веб-приложения для обнаружения серверных входных точек и последующего поиска уязвимостей в них. Метод разработан с учётом особенностей реального кода, отправляющего запросы на сервер, которые были выявлены в результате исследования. Проведена апробация реализованного метода на составленном эталонном наборе страниц, а также на наборе приложений, использовавшихся в других работах. Кроме того, проведены эксперименты на сайтах в сети Интернет, в результате которых обнаружены реальные уязвимости.

Апробация работы. Основные результаты работы докладывались на:

– Конференции Positive Hack Days 11, Москва, 18-19 мая 2022 г.
– Конференции OFFZONE 2022, Москва, 25-26 августа 2022 г.

– Конференции Ломоносовские чтения — 2023, Москва, 4-14 апреля 2023 г.
– Семинаре Workshop on Attacks and Software Protection 2023, проведённом в рамках конференции ESORICS 2023, Гаага, Нидерланды, 29 сентября 2023 г.
– Конференции Positive Hack Days Fest 2, Москва, 23-26 мая 2024 г.

Личный вклад. Все представленные в диссертации результаты получены лично автором.

Публикации. Основные результаты по теме диссертации изложены в 6 печатных изданиях, 5 из которых [2—6] изданы в журналах, рекомендованных ВАК, 4 — в периодических научных журналах, индексируемых Web of Science и Scopus [2; 5—7]. Зарегистрирована 1 программа для ЭВМ [1].

В работах [2; 5—7] все результаты были получены лично автором.

В работе [3] автором был предложен алгоритм извлечения кода из JavaScript-комментариев, кроме того, автору принадлежит разработка и реализация базового алгоритма анализа, с которым был интегрирован модуль извлечения кода из комментариев, а также методическое обеспечение экспериментов.

В работе [4] автору принадлежит постановка задачи, методология исследования, а также проведение части экспериментов.

Объем и структура работы. Диссертация состоит из введения, четырёх глав, заключения и одного приложения. Полный объем диссертации составляет 133 страницы текста с 6 рисунками и 3 таблицами. Список литературы содержит 89 наименований.

Содержание работы

Во **введении** обосновывается актуальность исследований, проводимых в рамках данной диссертационной работы, ставятся цели и задачи работы, формулируется научная новизна и практическая значимость представляемой работы, а также приводятся основные положения, выносимые на защиту.

В первой главе рассматривается задача выявления поверхности атаки веб-приложения при поиске в нём уязвимостей методом “чёрного ящика” и производится обзор существующих подходов.

Необходимым первым шагом процесса поиска серверных уязвимостей методом “чёрного ящика” является выявление того, каким образом данные могут быть поданы на вход серверной части. В данной работе рассматривается наиболее частый способ передачи данных на сервер — передача в HTTP-запросах. Формат запросов определяется требованиями протокола HTTP и правилами, специфичными для каждого веб-приложения. *Серверной операцией* будем называть часть кода сервера, выполняемую при обработке определённого запроса. *Серверная входная точка*, соответствующая серверной операции — это шаблон

HTTP-запроса, описывающий множество всех запросов, вызывающих эту серверную операцию.

То, какая именно операция вызывается, определяется некоторой частью запроса. Будем называть *фиксированной* ту часть запроса, которая была учтена внутренними правилами маршрутизации при выборе вызываемой операции (при изменении этой части запрос уже не приводит к выполнению той же операции). Часто к фиксированной части относится URL-путь и метод запроса.

Кроме того, в запросе могут присутствовать значения параметров — их совокупность будем называть *вариативной* частью запроса. Запросы с различными наборами значений параметров (наборами данных в вариативной части) будут вызывать выполнение той же самой серверной операции при условии совпадения фиксированной части. К вариативной части нередко относятся значения query-параметров, тело запроса, значения заголовков.

Серверная входная точка специфицирует то, какие части HTTP-запросов, относящихся к этой входной точке, фиксированы, а какие могут варьироваться. Обращаясь к различным серверным входным точкам, можно вызывать выполнение различных операций. На листинге 1 приведены два запроса, относящихся к одной серверной входной точке, но имеющие различающиеся данные в вариативной части.

Листинг 1: Два запроса, обращающихся к одной и той же серверной входной точке, отвечающей за вход в личный кабинет (вариативная часть выделена синим)

```
POST /user/login HTTP/1.1
Host: website.com
Content-Type: application/x-www-form-urlencoded

login=alex&password=secret123
```

```
POST /user/login HTTP/1.1
Host: website.com
Content-Type: application/x-www-form-urlencoded

login=peter&password=hunter2
```

Имея полный набор серверных входных точек, можно выполнить все серверные операции, выполнение которых можно инициировать HTTP-запросом. Таким образом, набор серверных входных точек полностью определяет пространство атаки для исследуемого веб-приложения. При поиске уязвимостей необходимо обеспечивать как можно большее покрытие кода, поэтому при поиске серверных входных точек наиболее важным критерием качества является *полнота*.

Одним из основных способов поиска серверных входных точек является получение информации о них из клиентской части веб-приложения. Применение этого способа необходимо для получения приемлемого качества поиска уязвимостей в режиме “чёрного ящика”, что отмечается авторами работ, посвящённых автоматическому определению поверхности атаки веб-приложений[1,2].

Задачу определения серверных входных точек по клиентской части веб-приложения сводят к выявлению набора запросов, которые могут быть отправлены клиентской стороной на сервер, что тривиально для запросов, инициируемых элементами HTML-разметки, но нетривиально для запросов, отправляемых JavaScript-кодом. Именно второму виду запросов уделено основное внимание в настоящей работе.

В данной главе содержится обзор и сравнительный анализ методов выявления поверхности атаки и анализа JavaScript-кода. В обзоре рассмотрены методы, использующие как динамический, так и статический анализ. Наиболее важными работами являются методы выявления поверхности атаки Gelato и CHIEv, средства поиска уязвимостей Arachni и Htcap, а также статические анализаторы WALA, SAFE и TAJS.

Практически все средства автоматизированного поиска уязвимостей для выявления отправляемых клиентским JavaScript-кодом HTTP-запросов используют динамический анализ, а именно технику, известную как *динамический краулинг* — автоматизированное взаимодействие с элементами интерфейса на веб-странице с помощью управляемого браузера (симуляция пользовательских действий) и запись всех отправляемых на сервер запросов. У этого метода есть ограничения: пользовательский интерфейс может быть слишком сложен для осуществления всех возможных комбинаций действий с ним. Более того, существуют случаи, когда JavaScript-код, отправляющий запрос к определённой серверной входной точке, невозможно вызвать действиями в интерфейсе (недостижимый код). Реальными примерами таких ситуаций являются обращения ко входным точкам, относящимся к авторизованной зоне сайта (когда пользователь не авторизован), обращения к функциям приложения, которые в данный момент удаляют или, напротив, ещё не до конца добавили, отсутствие на сайте данных, необходимых для функционирования некоторых частей интерфейса (например, кнопка “оценить комментарий” не появляется на страницах форума, где ещё нет ни одного комментария).

Вследствие описанных ограничений динамического краулинга актуальным представляется разработка метода извлечения информации о серверных входных точках, основанного на *статическом* анализе. В этом случае задача сводится к определению возможных аргументов обращений к программным

[1] *Leithner, M.* CHIEv: Concurrent Hybrid Analysis for Crawling and Modeling of Web Applications / M. Leithner, D. E. Simos // SIGAPP Appl. Comput. Rev. New York, NY, USA, 2021. Июль. Т. 21, № 1. С. 5—23. URL: https://doi.org/10.1145/3477133.3477134.

[2] *Doupé, A.* Why Johnny Can’t Pentest: An Analysis of Black-Box Web Vulnerability Scanners / A. Doupé, M. Cova, G. Vigna // Detection of Intrusions and Malware, and Vulnerability Assessment / под ред. C. Kreibich, M. Jahnke. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010. С. 111—131.

интерфейсам отправки запросов на сервер. Статические методы способны анализировать весь код программы, независимо от того, достижим ли он или насколько сложно добиться его выполнения при реальном запуске. Таким образом, статический анализ смог бы выявлять в коде обращения к входным точкам, вызов которых посредством действий в интерфейсе сложен или невозможен. Кроме того, при использовании статического анализа возможна разработка алгоритма, который будет эффективно обрабатывать большие объёмы сложного кода, в частности, за счёт возможности анализировать множество путей одновременно, что позволяет не посещать отдельно все возможные пути. В ходе обзора научной литературы не было найдено работ, где статический анализ кода был бы предложен для обнаружения серверных входных точек в составе сканера безопасности веб-приложений. Современные сканеры безопасности, применяемые в индустрии, также не обладают подобным механизмом.

Одной из вероятных причин отсутствия такого решения является то, что современный JavaScript-код плохо поддаётся статическому анализу — это отмечается исследователями в существующих публикациях[3,4]. Причина связана как с особенностями языка, так и с существующими практиками написания кода на нём. Анализ особенностей популярного JavaScript-кода выполнен в главе 2. Кроме того, применимость статического анализатора в реальной практике разработки будет зависеть от его времени работы. Время работы автоматизированного средства поиска уязвимостей на одном веб-приложении не должно превышать нескольких часов. Это связано с тем, что системы поиска уязвимостей используются в составе цикла автоматического тестирования приложения, выполняемого при внесении изменений разработчиками (цикла “непрерывной интеграции”). Кроме того, анализаторы сканируют целиком вычислительные сети крупных организаций, в которых количество приложений может доходить до нескольких тысяч, и необходимо, чтобы сканирование завершалось быстрее, чем разработчики выпустят новую версию приложений. Современные веб-приложения содержат десятки, часто — до сотен страниц, на каждой из которых, вообще говоря, свой JavaScript-код. Поэтому для реальной применимости время анализа одной веб-страницы должно быть около нескольких минут.

В рамках обзора проведено сравнение существующих методов с точки зрения требований, предъявляемых к методу выявления поверхности атаки и сформулированных по результатам исследований главы 2. К этим критериям относятся требования по поддержке особенностей, которыми обладает реальный клиентский код, а также способность анализировать недостижимый код и отсутствие привязки к старым версиям веб-браузера (это может препятствовать поддержке новых конструкций языка). Показано, что ни один из

[3] *Sun, K.* Analysis of JavaScript programs: Challenges and research trends / K. Sun, S. Ryu // ACM Computing Surveys (CSUR). 2017. Т. 50, № 4. С. 1—34.

[4] *Ryu, S.* Toward Analysis and Bug Finding in JavaScript Web Applications in the Wild / S. Ryu, J. Park, J. Park // IEEE Software. 2019. Т. 36, № 3. С. 74—82.

методов не удовлетворяет полностью всем критериям. Сделан вывод о целесообразности разработки нового метода определения поверхности атаки сервера веб-приложения, использующего специализированный статический анализ клиентского JavaScript-кода и учитывающий особенности реальных приложений. Алгоритм должен отличаться от существующих большей масштабируемостью, а также анализировать весь имеющийся на странице код, включая недостижимый.

Вторая глава посвящена исследованию особенностей реального JavaScript-кода, влияющих на возможность его анализа с целью обнаружения серверных входных точек. На защиту выносятся следующие результаты этой главы: требования к инструментам построения поверхности атаки на основе статического анализа клиентского JavaScript-кода, сформулированные на основе результатов исследования, а также эталонный набор веб-страниц (бенчмарк), который может быть использован для автоматизированной оценки эффективности методов извлечения информации о серверных входных точках.

В связи с тем, что ни один из описанных в научной литературе статических анализаторов JavaScript-кода не применим для анализа клиентского кода на страницах современных веб-приложений, было принято решение провести исследование реального клиентского JavaScript-кода для выделения его особенностей, которые осложняют статический анализ и которые следует учитывать при разработке алгоритма обнаружения отправляемых запросов. Результаты такого исследования делают возможной разработку специализированного анализатора, который будет лучше адаптирован для работы с реальным кодом и за счёт этого обладать большей эффективностью.

В разделе 2.1 описывается корпус кода для исследования. Для проведения исследования был собран корпус веб-страниц реальных сайтов в сети Интернет. Для каждой страницы корпус включает в себя HTML-файл с разметкой самой страницы, а также набор подключаемых ею JavaScript-файлов и CSS-файлов. Для сбора корпуса был использован статический краулер, построенный на основе библиотеки Colly. Корпус был получен в результате запуска краулера на 50169 сайтах из списка Alexa Top 1 Million (это список наиболее популярных сайтов). Корпус содержит 3251056 JavaScript-файлов, относящихся к 15785278 веб-страницам.

Применявшийся метод исследования, описанный в разделе 2.2, заключался в следующем.

1. Производился случайный отбор страниц из собранного корпуса, каждая из которых вручную анализировалась на предмет наличия клиентского JavaScript-кода, отправляющего запросы на сервер. Всего на этом шаге было отобрано и проанализировано около 150 страниц, все из которых относились к разным веб-сайтам.
2. При обнаружении отправляющего запросы кода вручную выделялись его особенности, которые необходимо поддерживать статическому анализатору для того, чтобы отыскать эти запросы.

3. Для выделенных на предыдущем шаге особенностей производилась оценка их встречаемости на сайтах в сети Интернет. Для этого выполнялся автоматизированный поиск кода с выделенной особенностью по всем файлам корпуса.

В разделе 2.3 приведены выделенные особенности кода, а именно:

Использование упаковщиков модулей. Современный JavaScript-код нередко организуется разработчиками посредством разбивки на модули, которая часто выполняется *упаковщиками модулей* (module bundlers). При использовании упаковщика импорт модулей выполняется с помощью функции импорта `require`. Она получает на вход идентификатор модуля и возвращает объект `exports` с экспортируемыми значениями модуля. Для анализа такого кода необходимо точно определять, экспортируемые значения какого именно модуля вернёт функция импорта, вызванная с тем или иным аргументом, иначе межмодульный анализ потока данных будет невозможен. При этом в коде, разбитом на модули, запросы часто формируются на основе данных из нескольких модулей: например, домен и префикс URL-адреса определяют в модуле, отвечающем за конфигурацию, а непосредственно отправка запроса делается в другом модуле. Без межмодульного анализа запросы в таком коде обнаружены не будут.

В то же время статический анализ кода функции `require` нетривиален. Она извлекает функцию-обёртку модуля из объекта-хранилища, в качестве имени поля используя свой первый аргумент, а потом вызывает извлечённую функцию. Происходит косвенный вызов, в котором вызываемая функция определется аргументом функции `require` и полями объекта-хранилища. Объект `exports` формируется как побочный эффект вызова функции-обёртки модуля. Таким образом, требуется контекстно-чувствительный анализ с поддержкой косвенных вызовов, обращений к полю объекта по вычисленному имени и побочных эффектов вызова функций. В ходе эксперимента использование упаковщиков модулей было обнаружено на 67,6% сайтов из выборки.

Использование классов. Код, отправляющий запросы на сервер, бывает написан в объектно-ориентированном стиле — с использованием классов. При анализе методов классов в таком коде анализатору нужно определять, какие данные будут получаться при чтении полей объекта `this`, т. е. объекта-экземпляра класса, чей метод вызывается, и какие данные записываются в эти поля. Также нужно понимать, к какому классу относится метод, и находить код вызываемых методов при анализе мест их вызова. Были выявлены как случаи, когда методы классов непосредственно осуществляли отправку запросов на сервер, так и случаи, когда они участвовали в формировании данных запроса, который затем отправлялся другим кодом. Классы в языке JavaScript могут быть объявлены разными способами. Было выделено 2 подвида описываемой особенности, различающихся способом объявления. В ходе эксперимента код, использующий классы, был найден на 91% сайтов из выборки.

Использование непрямого объявления класса. Такой способ объявления был единственным возможным до появления в языке ключевого слова `class`,

которое, как и ключевые слова `import` и `export`, было добавлено в версии ECMAScript 6. При использовании непрямого способа объявления в качестве класса в коде объявляется функция. Применение оператора `new` к этой функции приводит к созданию нового объекта, причём функция будет вызвана для этого объекта в качестве конструктора. На 90,86% сайтов из выборки был обнаружен код с классами, объявленными таким способом.

Использование прямого объявления класса с помощью ключевого слова ***`class`***. В этом случае основной частью объявления является тело класса, в котором как конструктор, так и остальные методы перечислены явно. Классы, объявленные таким способом, найдены на 32,2% сайтов.

Использование строковых операций. Код, отправляющий запросы на сервер, активно использует строковые операции для формирования частей запросов — URL-адресов, тел запросов, значений заголовков. Для определения того, какой вид имеют запросы, анализаторам нужно как можно точнее моделировать эти операции — причём как с полностью конкретными данными, так и с частично неизвестными. 95,6% сайтов из выборки содержали клиентский JavaScript-код, в котором есть строковые операции.

Передача функциональных значений в программе и косвенные вызовы. Практически все вызовы в JavaScript-коде косвенные. Семантика вызова такова, что то, какая функция вызывается, всегда определяется динамически результатом выражения, задающего вызываемую функцию. Функциональные значения в JavaScript-коде нередко переприсваиваются между переменными (в том числе при использовании упаковщиков модулей), передаются в функции в качестве аргументов (функций обратного вызова).

Поддерживать косвенные вызовы необходимо, но при этом важно, чтобы анализатор отрабатывал за разумное время и при разумных ограничениях по памяти. Анализ кода, активно использующего эту возможность языка, может не быть полностью точным, но должен осуществляться при приемлемом потреблении вычислительных ресурсов. Доля сайтов, в клиентском коде которых были найдены косвенные вызовы, составила 95,68%.

Обращение к полю объекта по вычисленному имени. Язык JavaScript допускает обращение к полю объекта по вычисленному имени — когда имя поля не задано в коде явно, а вычисляется динамически (в качестве имени поля используется строковое значение). Эта возможность языка часто используется в реальном коде, что затрудняет статический анализ кода. Доля сайтов, на страницах которых была обнаружена операция обращения к полю объекта по вычисленному имени, составила 95,2%.

Использование библиотек, код которых сложен для статического анализа. Как и другие виды программ, клиентский JavaScript-код активно использует библиотеки. Код современных JavaScript-библиотек зачастую сложен для анализа. Это касается в том числе и библиотеки jQuery — наиболее популярной библиотеки для клиентского JavaScript-кода. Сложность возникает из-за упоминавшихся выше особенностей — передачи функциональных значений и

косвенных вызовов, использования строковых операций, обращения к полям объектов по вычисленным именам. Также в коде библиотек (например, Angular) используются классы. Более современные библиотеки (например, Axios) разбиты на несколько модулей, и анализ их кода требует поддержки упаковщиков модулей. Доля сайтов, на страницах которых было обнаружено использование библиотек со сложным кодом, составила 78,6%.

В разделе 2.4 описан эталонный набор веб-страниц (бенчмарк), созданный по результатам проведённого исследования. Набор содержит 32 страницы, для каждой из которых есть список входных точек, запросы к которым отправляются JavaScript-кодом страницы. Всего в наборе 884 входных точки. Страницы приведены в виде файла с HTML-разметкой и подключаемых JS и CSS файлов. Набор может быть использован для автоматизированной оценки эффективности методов извлечения информации о серверных входных точках из клиентского кода. Большая часть страниц в наборе (28 страниц) взяты из веб-сайтов, содержащихся в списке Alexa Top 1 Million. Кроме того, добавлены страницы тестовых приложений Juice Shop и Joomla Store, а также страницы популярных веб-приложений с исходным кодом Mattermost и YDB. Для всех особенностей кода, описанных в предыдущем разделе, в эталонном наборе содержатся страницы, для которых обнаружение отправляемых со страницы запросов требует поддержки соответствующей особенности.

Созданный эталонный набор веб-страниц доступен по адресу `https://github.com/seclab-msu/ajax-page-dataset`.

В разделе 2.5 приведены выводы из исследования, а также сформулированы требования к инструментам построения поверхности атаки посредством статического анализа клиентского JavaScript-кода.

Общим выводом исследования стало то, что для определения отправляемых запросов посредством статического анализа необходим межпроцедурный алгоритм анализа, способный определять возможные наборы аргументов вызываемых функций и возвращаемые значения. При этом алгоритм должен как можно точнее различать вызовы одной и той же функции (обладать хорошей контекстной чувствительностью), что требуется для повышения точности получаемых результатов анализа. Встречаемость сайтов, в коде которых используются строковые операции и обращения к полям объекта по вычисленным именам, оказалась высокой, поэтому анализ должен поддерживать такие операции. Для применимости в составе реальных систем автоматизированного поиска уязвимостей время анализа одной веб-страницы должно составлять несколько минут. Это связано с современными требованиями к предельному времени работы систем поиска уязвимостей на одном приложении и тем, что у одного приложения, как правило, есть множество страниц, которые будут обрабатываться анализатором.

Алгоритм анализа должен отвечать следующим требованиям:

1. Поддерживать анализ кода, поставляемого в виде набора модулей, запакованных упаковщиком модулей (module bundler).

2. Поддерживать анализ кода, использующего классы. В том числе:
    а) объявленные посредством непрямого объявления класса,
    б) объявленные посредством прямого объявления класса (с помощью ключевого слова `class`).
3. Поддерживать строковые операции языка JavaScript и вычислять результаты этих операций.
4. Поддерживать анализ кода, содержащего косвенные вызовы и передачу функциональных значений между переменными программы и полями объектов и массивов. Анализатор может не всегда точно анализировать косвенные вызовы, но он должен выполнять анализ программ, содержащих такие вызовы, за приемлемое время и при приемлемом потреблении вычислительных ресурсов.
5. Поддерживать операцию обращения к полю объекта по вычисленному имени, вычисление значения, используемого в качестве имени поля. В случае, когда может быть вычислено точное значение или набор значений, используемых в качестве имени поля, определять, к каким именно полям объекта производится обращение.
6. Поддерживать анализ программ, использующих популярные библиотеки, в том числе сложные для статического анализа, включая библиотеку jQuery.

Третья глава содержит описание методики поиска уязвимостей веб-приложений в модели “чёрного ящика” с использованием статического анализа клиентского JavaScript-кода, которая выносится на защиту. Методика апробирована в реальной системе автоматизированного поиска уязвимостей веб-приложений SolidPoint, разрабатываемой компанией ООО «СолидСофт», а также в системе автоматического поиска уязвимостей в веб-приложениях на основе обработки больших данных, разработанной в Центре компетенций НТИ по технологиям хранения и анализа больших данных МГУ (ЦХАБД МГУ) [1].

Методика основана на открытой методике тестирования веб-приложений с точки зрения безопасности OWASP Web Security Testing Guide версии 4.2 (WSTG), четвёртом разделе[5].

Традиционно при поиске уязвимостей выделяются этапы:

1. Сбора информации об анализируемом приложении. Этому этапу соответствует пункт 4.1 методики WSTG.
2. Непосредственно проверки приложения на наличие уязвимостей с использованием собранной информации. Этому этапу соответствуют пункты 4.2-4.12 методики WSTG.

Раздел 3.1 посвящён сбору информации об анализируемом приложении. Предлагаемая методика состоит в выполнении шагов 4.1.1-4.1.10 методики

[5]Раздел Web Application Security Testing методики OWASP Web Security Testing Guide версии 4.2 [Электронный ресурс]. URL: https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/ (дата обр. 03.07.2024).

WSTG, а также в поиске серверных входных точек посредством следующих шагов:

1. Найти все веб-страницы приложения статическим краулингом.
2. Для каждой найденной страницы выполнить поиск серверных входных точек, к которым обращается страница, следующими способами:
    а) Выполнить поиск отправляемых запросов, инициируемых элементами HTML-разметки, с помощью синтаксического разбора HTML-разметки страницы и анализа её элементов.
    б) Выполнить поиск отправляемых запросов статическим анализом, удовлетворяющим требованиям раздела 2.5. Такой анализ предложен в данной работе в главе 4.
3. Выполнить поиск серверных входных точек с помощью запуска динамического краулера, который должен посетить все страницы, найденные на шаге 1.
4. Выполнить поиск серверных входных точек с помощью активного управляемого перебора запросов к серверной стороне приложения и анализа ответов.
    а) Перебор URL-путей по словарю, так называемый дирбастинг. Такой перебор может быть выполнен с помощью средств OWASP Dirbuster, dirsearch, Gobuster.
    б) Перебор имён параметров запросов и других частей запроса. Примером средства с открытым исходным кодом для этой задачи является ffuf.
5. Определить используемое серверное ПО (набор используемых программ и их версий) по сигнатурам и произвести поиск по открытым источникам или заранее подготовленной базе данных информации об известных серверных входных точках для выявленного ПО.

Раздел 3.2 посвящён проверке приложения на наличие серверных и клиентских веб-уязвимостей. Все серверные входные точки, обнаруженные на этапе сбора информации об анализируемом приложении, должны быть проверены на уязвимости, включая такие, как SQL-injection, Reflected XSS, XXE, Server-side template injection. Для этого должны быть выполнены шаги 4.2-4.12 методики WSTG. Кроме того, предлагаемая методика предписывает поиск уязвимостей в клиентском JavaScript-коде статическим и динамическим анализом. Должна быть проведена проверка на наличие уязвимостей DOM-based XSS, Client-side prototype pollution, а также ошибок в клиентском коде, приводящим к утечкам данных. Существует ряд открытых инструментов, осуществляющих такой анализ (к примеру, DOMDig, esflow, TaintFlow), в научных статьях описаны методы организации анализа[6].

[6] *Kang, Z.* Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites. / Z. Kang, S. Li, Y. Cao // NDSS. 2022; *Lekies, S.* 25 million flows later: large-scale detection of DOM-based XSS / S. Lekies, B. Stock, M. Johns // Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. 2013. С. 1193–1204; Staged information flow for

Раздел 3.3 содержит примеры применения описанной методики на реальных приложениях, а раздел 3.4 — сделанные на их основе выводы. Методика применялась с использованием описанного в главе 4 метода анализа на реальных веб-приложениях из программ вознаграждения за обнаружение уязвимостей (программы Bug Bounty). В результате было отправлено 20 отчётов о найденных уязвимостях, 11 из которых были уязвимостями типа Reflected XSS, остальные относились к классам SQL injection и “небезопасная десериализация данных”. Была обнаружена уязвимость типа SQL injection на сайте компании IBM `www.ibm.com` и уязвимость типа Reflected XSS на сайте Amazon `www.amazon.com`. Можно сделать вывод, что включение в методику алгоритмов статического анализа клиентского JavaScript-кода позволяет обнаружить серверные входные точки, не найденные другими методами анализа, и в итоге найти новые уязвимости.

Четвёртая глава посвящена описанию предлагаемого специализированного метода анализа клиентского кода веб-приложения для обнаружения серверных входных точек, описанию апробации реализованного метода с использованием составленного эталонного набора страниц и на наборе приложений, использовавшихся для сравнения в предыдущих работах, а также описанию проведённых экспериментов с сайтами из сети Интернет, в результате которых были обнаружены реальные уязвимости. Предложенный метод позволяет, имея на входе веб-страницу (данную в виде URL-адреса), получить набор спецификаций HTTP-запросов, отправляемых на сервер JavaScript-кодом этой страницы. Метод удовлетворяем всем требованиям к инструментам построения поверхности атаки, сформулированным в разделе 2.5. Метод заключается в поиске в JavaScript-коде обращений к программным интерфейсам отправки запросов на сервер (практически всегда они имеют вид вызовов функций) и определении аргументов, передаваемых в эти обращения, а также последующем определении для каждого найденного обращения вида HTTP-запроса, который был бы отправлен этим обращением с соответствующими аргументами. Обращения к программным интерфейсам отправки запросов на сервер будем далее называть *AJAX-вызовами*. Для обнаружения AJAX-вызовов и определения их аргументов в работе предложен специализированный алгоритм статического анализа, учитывающий особенности реального кода, выявленные в ходе исследования главы 2. Общая схема работы метода приведена на Рисунке 1. Красной пунктирной линией выделены компоненты, разработанные в рамках диссертационной работы.

Для применения алгоритма необходимо провести предварительный шаг в виде сбора JavaScript-кода, относящегося к анализируемой странице, синтаксического разбора этого кода и получения информации о его лексических областях видимости. Для сбора кода нужно открыть анализируемую страницу в управляемом браузере Headless Chrome и использовать отладочный программный интерфейс браузера для получения всех фрагментов JavaScript, которые

JavaScript / R. Chugh [и др.] // Proceedings of the 30th ACM SIGPLAN conference on programming language design and implementation. 2009. С. 50–62.

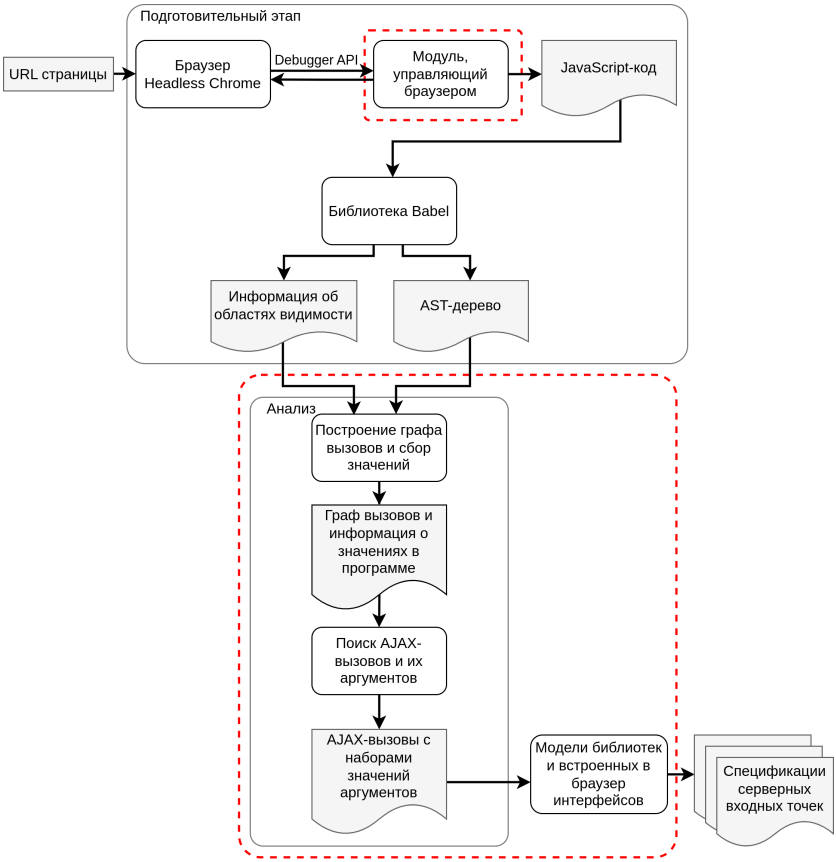

Рис. 1 — Общая схема работы метода

попадают в интерпретатор браузера. Этот метод был предложен и реализован в нашей работе [6]. Полученные фрагменты кода конкатенируются в один большой фрагмент в том порядке, в котором они были загружены в интерпретатор. Для синтаксического разбора используется парсер JavaScript, предоставляемый библиотекой Babel, который строит AST-дерево. Информация о лексических областях видимости, позволяющая различать в программе одноимённые переменные, также получается с помощью библиотеки Babel.

Предлагаемый алгоритм анализа принимает на вход AST-дерево JavaScript-кода и информацию о лексических областях видимости. Результатом работы является набор AJAX-вызовов, каждым элементом которого является пара из названия программного интерфейса отправки запросов и массива значений передаваемых в вызов аргументов. Алгоритм состоит из двух основных шагов:

1. Построение графа вызовов и сбор возможных значений переменных, полей объектов и массивов.

2. Поиск обращений к программным интерфейсам отправки запросов и аргументов, передаваемых в эти обращения.

Исходный код разработанного анализатора доступен по ссылке https://github.com/seclab-msu/relwarc.

Раздел 4.1 содержит описание алгоритма построения графа вызовов и сбора возможных значений переменных, полей объектов и массивов. Построение графа вызовов и сбор значений выполняются совместно с помощью итеративного алгоритма (см. алгоритм 1). Каждая итерация — это рекурсивный обход AST-дерева программы в глубину с обработкой встречаемых вершин для сбора информации о программе. Анализатор поддерживает *абстрактное состояние* программы, содержащее собранную информацию. Итерации выполняются до тех пор, пока абстрактное состояние не перестанет меняться. Кроме того, анализатор поддерживает ограничение на максимальное количество итераций, при достижении которого анализ также прекращается. Алгоритм не обладает чувствительностью к путям, все ветви условных операторов и циклов посещаются безусловно.

Алгоритм анализа поддерживает значения следующих типов (объединение всех поддерживаемых типов будем называть $Value$):

- Значения *примитивных* типов JavaScript: `undefined`, `number`, `string`, `boolean`, а также значение `null`.
- $Unknown$ — *неизвестные* значения, существует всего 2 уникальных неизвестных значения: $Unknown$ и $FromArg$.
- $Value[]$ — *массивы*, элементами которых могут быть значения любых поддерживаемых типов.
- *Объекты* $\{string \mapsto Value\}$, представляющие собой ассоциативные массивы, ключами в которых являются строки, а значениями — значения любых поддерживаемых типов.
- $ValueSet$ — *множество* возможных значений (любых поддерживаемых типов). $ValueSet$ используется для выражения того, что в определённом месте может быть одно из нескольких возможных значений.
- $FunctionValue$ — *функциональное значение*. Каждой функции в программе (т. е. объявлению функции или функциональному выражению) сопоставлено значение $FunctionValue$, причём только одно.
- *Специальные объекты* — объекты, для операций с которыми анализатором используется специальная семантика.

Абстрактное состояние программы представляет собой четвёрку

$$\mathcal{A} = \langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle$$

Абстрактное состояние состоит из *состояния памяти* $\boldsymbol{M}$, *информации о функциях* $\boldsymbol{F}$, *информации о классах* $\boldsymbol{C}$ и *графа вызовов* $\boldsymbol{CG}$. Состояние памяти — это отображение, ставящее значения в соответствие переменным. Информация о функциях $\boldsymbol{F}$ содержит наборы возможных аргументов и возвращаемых значений функций. Информация о классах $\boldsymbol{C}$ представляет собой набор описаний классов

$\mathbf{c}_i$, где каждое описание содержит значение экземпляра класса, объект класса и набор методов класса. Граф вызовов $\boldsymbol{CG}$ сопоставляет каждому месту вызова множество функций, которые могут быть вызваны в этом месте.

Алгоритм 1: Алгоритм построения графа вызовов и сбора значений

Входные данные: $PageURL$, AST
Результат: $\boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG}$
$GatherBundledModuleInfo(AST)$
$\mathcal{A} \leftarrow \langle SeedInitialMem(PageURL), \varnothing, \varnothing, \varnothing \rangle$
$\mathcal{A}_{old} \leftarrow nil$
$i \leftarrow 0$
while $\mathcal{A} \neq \mathcal{A}_{old}$ ***and*** $i < MaxIter$ **do**
    $\mathcal{A}_{old} \leftarrow \mathcal{A}$
    $\mathcal{A} \leftarrow AnalysisPass(AST, \mathcal{A})$
    $i \leftarrow i + 1$
$\langle \boldsymbol{M}, \boldsymbol{F}, \boldsymbol{C}, \boldsymbol{CG} \rangle \leftarrow \mathcal{A}$

---

function $AnalysisPass(AST, \mathcal{A})$
    forall *vertex* v ***traversing*** AST ***in DFS order*** **do**
        if $isLibrary(v)$ **then**
            skip subtree and continue
        $\mathcal{A} \leftarrow ApplyEffect(v, \mathcal{A})$
    return $\mathcal{A}$

В ходе рекурсивного обхода, выполняемого на каждой итерации анализа, анализатор обрабатывает вершины AST, соответствующие инструкциям, которые влияют на абстрактное состояние. За эту обработку в алгоритме 1 отвечает функция $ApplyEffect$. Эта функция при необходимости вычисляет выражения, которыми заданы входные данные инструкции, используя старое значение состояния, после чего применяет к состоянию эффект от инструкции, которой соответствует обрабатываемая вершина. Учитываются следующие вершины: объявления переменных, присваивания переменных и полей объектов и массивов, функции, классы, места вызова функций, места инстанциирования классов, инструкции возврата из функции (`return`).

Перед началом основных итераций алгоритма выполняется предварительный проход по AST-дереву программы с целью сбора информации о модулях, помещённых в код с помощью упаковщиков модулей — за этот шаг в псевдокоде отвечает функция $GatherBundledModuleInfo$. В ходе работы алгоритма перед обработкой вершины проверяется, не является ли вершина корнем AST-дерева кода библиотеки. За эту проверку отвечает функция $isLibrary(v)$. Проверка делается сравнением с синтаксическими сигнатурами. В анализатор встроен ряд

сигнатур известных библиотек, в том числе, сигнатура библиотеки jQuery. Если для вершины v зафиксировано совпадение с сигнатурой, то пропускается эта вершина и всё поддерево, корнем которого является v.

В разделе 4.2 приводится алгоритм поиска обращений к программным интерфейсам отправки запросов на сервер и аргументов, передаваемых в эти обращения. На этом этапе производится ещё один обход AST-дерева программы $FindRequestsAnalysisPass$. Обход делается аналогично тому, как работает функция $AnalysisPass$ из предыдущего раздела, со следующими отличиями:

– при заходе в тело функции при рекурсивном обходе AST ко множеству возможных значений её формальных аргументов добавляется $FromArg$;
– при обработке некоторых вершин (вызовов функций и присваиваний) дополнительно делается проверка, не является ли выполняемая в данной вершине операция AJAX-вызовом.

Если операция распознана как AJAX-вызов, её аргументы вычисляются и в набор результатов работы алгоритма добавляется запись с названием операции и массивом вычисленных значений формальных аргументов. Практически все AJAX-вызовы имеют вид вызовов функций, поэтому рассмотрим подробнее именно этот случай. Для вызова производится проверка, не является ли он вызовом, отправляющим запросы на сервер (вызовом *AJAX-функции*), следующими способами:

– По имени функции в месте вызова (по синтаксическим сигнатурам). В анализатор встроен набор шаблонов, описывающих обычные имена отправляющих запрос функций. К примеру, в этот шаблон входят такие имена, как `fetch`, `$.ajax` и `$http.post`.
– По значению функции, взятому для данной вершины из графа вызовов $\boldsymbol{CG}$. Для этого анализатором поддерживаются специальных функциональные значения, помеченные, как значения AJAX-функций. Они создаются для функций, чьё тело было опознано по синтаксической сигнатуре.

После одного такого прохода по AST-дереву программы анализатор дополнительно производит *анализ цепочек вызова, ведущих к AJAX-вызовам*. Цепочки вызовов строятся для тех AJAX-вызовов, у которых аргументы вызова зависят от формальных аргументов функции, содержащей вызов. Будем называть эту содержащую AJAX-вызов функцию $caller$. Такую ситуацию можно определить по факту наличия значения $FromArg$ среди значений аргументов AJAX-вызова. В этом случае в графе вызовов $\boldsymbol{CG}$ будет выполнен поиск вызовов функции $caller$. Далее происходит анализ всех найденных мест вызова — для каждого найденного места вызова выполняется ещё один проход $FindRequestsAnalysisPass$, начиная не от корня AST-дерева всей программы, а от корня AST-дерева функции, содержащей найденное место вызова $caller$. Когда обход AST дойдёт до вызова $caller$, он перейдёт на начало её кода (на корень её AST-дерева) с подстановкой вычисленных фактический значений аргументов, эмулируя её вызов.

После чего обход AST дойдёт до интересующего AJAX-вызова, но уже с более точными значениями аргументов. Может случиться, что аргументы вызова $caller$ в свою очередь зависели от аргументов вызова функции, содержащей вызов $caller$ (назовём эту функцию $caller'$). Такая ситуация также может быть зафиксирована по наличию значения $FromArg$ среди аргументов AJAX-вызова. В этом случае будет произведён поиск мест вызова $caller'$, а затем — анализ цепочек уже из двух вызовов, и так далее. Алгоритм анализа предполагает ограничение на максимальную длину цепочек вызовов. Это настраиваемый параметр алгоритма, в текущей версии анализируются цепочки не длиннее пяти вызовов.

В разделе 4.3 описывается преобразование найденных вызовов со значениями аргументов в набор спецификаций HTTP-запросов, отправляемых на сервер JavaScript-кодом страницы. То, какие HTTP-запросы будут отправлены тем или иным вызовом с некоторыми аргументами, определяется с помощью набора эвристических *моделей функций, отправляющих запросы*. Они моделируют работу поддерживаемых библиотек и встроенных в браузер механизмов в части отправки запросов на сервер. Для каждой функции модель содержит *сигнатуру вызова* и написанный вручную код, который на основе аргументов вызова и исходного URL-адреса страницы определяет вид отправляемого запроса.

Раздел 4.4 посвящён улучшению алгоритма за счёт анализа закомментированного клиентского кода. Такой код в клиентской части веб-приложения может содержать полезную информацию о серверных точках входа, недоступную в “живом” коде. Алгоритм извлечения закомментированного кода (алгоритм 2) получает на вход текст JavaScript-кода страницы. На первом этапе происходит удаление пустых строк. Поскольку на одном из следующих этапов необходимо будет объединить близлежащие однострочные комментарии в блочные, то пустые строки, встречающиеся между ними, могут повлиять на синтаксическую корректность уже объединенных блоков. На втором этапе происходит синтаксический разбор кода страницы при помощи библиотеки Babel Parser, результатом которого является AST-дерево. Из полученного дерева извлекается массив закомментированных строк и блоков. Этот массив подается на вход функции, которая объединяет находящиеся на соседних строках комментарии в блоки. После слияния комментариев начинается этап парсинга. Аналогично второму этапу, при помощи Babel Parser проводится синтаксический разбор каждого из полученных блоков. Если парсер сообщает об ошибке, то необходимо удалить вызывающий ее символ и повторить этап заново. Если по результатам некоторого числа итераций удается представить комментарий в виде AST-дерева, он будет добавлен в область для последующего анализа.

Раздел 4.5 содержит особенности реализации алгоритма и некоторые полезные эвристики. Анализатор реализован на языке TypeScript и выполняется в среде NodeJS. Операции с примитивными типами выполняются напрямую на интерпретаторе NodeJS, сделанные страницей запросы отслеживаются автоматически, и после окончания всех запросов JavaScript-код страницы

Алгоритм 2: Алгоритм извлечения закомментированного кода

```
function ExtractCommentedCode(Scripts)
    ScriptsAST ← []
    forall script s ∈ Scripts do
        s ← RemoveBlankLines(s)
        mergedComments ← MergeComments(s)
        forall blockComment ∈ mergedComments do
            ScriptsAST.append(ParseComment(blockComment))
    return ScriptsAST
function ParseComment(BlockComment)
    while true do
        ast ← ParseCode(BlockComment)
        if ast = nil then
            BlockComment ← DeleteBadSymbol(BlockComment)
        else
            return ast
```

запрашивается из соответствующих DOM-элементов. Анализатор различает экземпляры класса, инстанцированные в различных местах кода, и эмулирует вызовы с анализом вызываемой функцией на глубину не более двух для определения возвращаемых значений.

В разделе 4.6 описана апробация реализованного метода с использованием составленного эталонного набора страниц, а также на наборе приложений, использовавшихся для сравнения в предыдущих работах.

Раздел 4.6.1 содержит результаты на эталонном наборе страниц (бенчмарке), описанном в разделе 2.4. Среднее покрытие входных точек предложенным методом на эталонном наборе составило 42,6%, тогда как все протестированные статические анализаторы JavaScript-кода (WALA, SAFE и TAJS) не проанализировали ни одну страницу из набора — они либо завершаются с ошибкой, либо зависают. Причины пропуска входных точек были проанализированы, среди основных — использование при формировании запросов данных, считанных из DOM-модели, а также использование таких возможностей языка JavaScript, как наследование классов и применение встроенных в язык операторов `import` и `export`. Поддержку кода с этими особенностями планируется добавить в дальнейшей работе.

Раздел 4.6.2 посвящён эксперименту по сравнению с аналогами на 5 приложениях с открытым исходным кодом, которые также были использованы для тестирования в других недавних работах: DVWA, JuiceShop, MyBB, WebGoat и WIVET. Сравнение производилось с инструментами Arachni, Crawljax, Enemy of the State (EoS), Htcap, w3af и wget. В качестве метрики качества использовалось количество найденных входных точек. Полные данные эксперимента, включая

запросы, сделанные сравниваемыми средствами, и выявленные входные точки, были опубликованы[7]. Результаты эксперимента приведены в таблице 1. Колонка *Всего* содержит количество входных точек в объединении множеств входных точек, найденных всеми инструментами.

Таблица 1 — Уникальные входные точки, найденные каждым из инструментов

Название	*Всего*	Arachni	Crawljax	EoS	Htcap	w3af	wget	*Метод*
DVWA	57	36	43	40	**54**	48	1	**54**
JuiceShop	37	9	9	2	13	1	2	**36**
MyBB	110	68	7	65	63	59	52	**77**
WebGoat	78	**78**	10	1	10	1	10	12
WIVET	51	**51**	5	4	42	25	7	41

Разработанный нами метод обнаружил больше серверных входных точек, чем краулеры, в двух приложениях: JuiceShop и MyBB. Для приложения DVWA количество найденных входных точек оказалось одинаковым у нашего метода и краулера Htcap. На двух приложениях, WebGoat и WIVET, результаты разработанного метода оказались хуже.

JavaScript-код на страницах JuiceShop и MyBB содержит обращения к серверным входным точкам, которые невозможно вызвать из пользовательского интерфейса, но которые смог обнаружить предлагаемый алгоритм анализа. JuiceShop относится к приложениям, у которых весь клиентский JavaScript-код присутствует на каждой из страниц, включая главную, и содержит обращения ко всем серверным входным точкам. При этом часть пользовательского интерфейса, вызывающего обращения к этим входным точкам, не показывается не прошедшему аутентификацию пользователю. Приложение MyBB — это форум. Сразу после установки там нет ни одного обсуждения или поста (что соответствует условиям эксперимента в других работах). Поскольку содержимое в форуме отсутствует, некоторые действия (отмечание поста как прочитанного или запрос автора поста) не могут быть осуществлены из интерфейса — соответствующие управляющие элементы не появляются на страницах. Однако запросы, выполняющие эти действия, обнаружены разработанным анализатором.

Клиентский код приложения WebGoat содержит особенности, не поддерживаемые предложенным методом из-за их низкой распространённости (встречаются менее, чем на 0,8% сайтов). Тем не менее, планируется доработать метод для их поддержки. У приложения WIVET ряд обращений к серверу делается способами, выходящими за рамки данной работы (например, используется Flash-анимация).

Максимальное время анализа одной страницы составило 28 секунд (такое время занял анализ страницы JuiceShop). Наибольшее время полного анализа всего приложения составило 5 минут и 2 секунды (такое время занял анализ приложения MyBB).

[7] https://github.com/asterite3/finding-endpoints-with-analysis-experiment-data

В разделе 4.7 описаны результаты применения разработанного метода к сайтам из сети Интернет и найденные реальные уязвимости.

Раздел 4.7.1 посвящён проведённому эксперименту по поиску реальных уязвимостей в серверных входных точках, обнаруженных разработанным методом. Эксперимент проведён на примере класса уязвимостей “Небезопасная десериализация” (также известного как “Десериализация недоверенных данных”). Уязвимости данного класса до сих пор распространены и несут серьёзную угрозу безопасности сервера. Найденные случаи сериализованных объектов в клиентском коде были разбиты на классы. Для нескольких классов было установлено, что они “уязвимы”, то есть все приложения, относящиеся к ним, содержат уязвимость типа “Небезопасная десериализация”.

Класс *Settings*: в данный класс входят приложения, сделанные на основе PHP–фреймворка OpenCart с использованием одного и того же плагина. В данном плагине присутствует уязвимость небезопасной сериализации, позволяющая выполнить произвольный код на сервере. Данная уязвимость не была ранее известной, поэтому разработчики продукта были оповещены о ее наличии данной уязвимости. Уязвимости присвоен идентификатор CVE-2022-24108[8].

Класс *SimpleAjaxManagerWordpress*: приложения данного класса используют систему управления контентом WordPress (однуиз самых распространенных) и плагин Simple Ajax Manager. Были найдены две уязвимости: небезопасная десериализация и SQL-инъекция через поля сериализованного объекта. Найденная SQL-инъекция не была ранее известной. Информация о ней была добавлена в существующую запись о недостатке “Десериализации недоверенных данных”[9].

Класс *LoadMoreWordpress*: такие приложения используют WordPress, но компонента, в которой находится небезопасная десериализация, не является плагином. Уязвимый код был приведен на одном из форумов для разработчиков. При помощи эксплуатации небезопасной десериализации получилось добиться выполнения произвольного кода на серверной стороне для всех этих приложений. Был отправлен отчет об уязвимости на один из главных агрегаторов Bug Bounty программ HackerOne.

Для всех упомянутых “уязвимых” классов предложенный в работе анализ клиентского JavaScript-кода определил правильный вид HTTP-запроса, отправляющего сериализованный объект на сервер.

Раздел 4.7.2 содержит описание проведённого эксперимента с использованием модуля анализа закомментированного кода, представленного в разделе 4.4. Была исследована гипотеза о том, что закомментированный JavaScript-код клиентской части веб-приложения может содержать полезную информацию о серверных точках входа, недоступную в “живом” коде клиентской части. Эксперимент использовал базу данных из 50291 сайта из списка Alexa Top 1 Million. Всего проанализировано 580056 страниц.

[8]CVE-2022-24108 (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-24108)

[9]Wordpress: Simple Ads Manager vulnerability page (https://wpscan.com/vulnerability/38787b49-c19c-49db-925a-69e2c9cf7a43)

В результате AJAX-вызовы, встречающиеся только в закомментированном коде, оказались на 9948 страницах (1,72%) или в 1396 веб-приложениях (2,78%). Всего найдено 14395 закомментированных вызовов. Из них оказалось 3923 уникальных (27,25%). Соответствующие конечные точки на сервере имели 1536 вызовов (39,15% среди уникальных), относящихся к 1232 ресурсам (0,21%) или 739 различным веб-приложениям (1,47% от общего количества). Закомментированные запросы с оставленной функциональностью на сервере были исследованы на наличие уязвимостей. Из них 5,6% оказались уязвимы. Самые часто встречающиеся виды уязвимостей — Reflected XSS и SQL Injection. Кроме того, были найдены уязвимости типа PHP code injection и некоторых других типов.

В **заключении** приведены основные результаты работы, которые заключаются в следующем:

1. Проведено исследование особенностей реального JavaScript-кода, влияющих на возможность его анализа с целью обнаружения серверных входных точек, на наборе популярных приложений из рейтинга Alexa Top 1 Million. Выделены наиболее существенные особенности, в том числе использование упаковщиков модулей, непрямые объявления классов, обращения к полям объектов по вычисленному имени. С учетом выявленных особенностей поставлена задача разработки специализированного статического анализа и сформулированы требования к нему. На основе исследованных приложений составлен эталонный набор веб-страниц, позволяющий оценивать эффективность методов извлечения информации о серверных входных точках из клиентского кода.
2. Предложена новая методика поиска уязвимостей веб-приложений в модели “чёрного ящика”. Методика отличается от существующих повышением покрытия серверных входных точек, обращения к которым сложно вызвать через взаимодействие с пользовательским интерфейсом, за счет применения статического анализа и автоматизированного перебора имён параметров запроса, а также применением анализа кода клиентской стороны веб-приложения для обнаружения уязвимостей.
3. Разработан и реализован метод анализа клиентского кода веб-приложения для обнаружения серверных входных точек с целью последующего поиска уязвимостей. Метод использует алгоритм статического анализа, разработанный с учётом выявленных особенностей реального кода и способный анализировать весь код страницы, включая недостижимый код. Алгоритм позволяет решать задачу обнаружения входных точек в заданных ограничениях по времени и по использованию вычислительных ресурсов, что достигается за счёт ограничения числа проходов по коду и добавления встроенной поддержки популярных JavaScript-библиотек, что позволяет не анализировать их собственный код.

4. Проведена апробация реализованного метода на наборе приложений, использовавшихся для сравнения в других работах, и на составленном эталонном набора страниц. На трех приложениях из пяти разработанный метод выявил максимальное количество серверных входных точек (при этом в двух приложениях все проверенные аналоги нашли меньше входных точек). Среднее покрытие входных точек на эталонном наборе составило 42,6%, а максимальное время выполнения на одном приложении — 5 минут 3 секунды при работе на одном ядре процессора AMD EPYC 7502. Проведены эксперименты с сайтами из сети Интернет, в результате которых на них были обнаружены реальные уязвимости.

Публикации автора по теме диссертации

1. *Свидетельство о регистрации прав на ПО, базу данных*. Система автоматического поиска уязвимостей в веб-приложениях на основе обработки больших данных / Д. Ю. Гамаюнов [и др.] ; МГУ имени М.В.Ломоносова. — № 2022610972 ; заявл. 18.01.2022 ; опубл. 18.01.2022, 2022610972 (Рос. Федерация).
2. *Сигалов, Д. А.* Обнаружение серверных точек взаимодействия в веб-приложениях на основе анализа клиентского JavaScript-кода / Д. А. Сигалов, А. А. Хашаев, Д. Ю. Гамаюнов // Прикладная дискретная математика. — 2021. — № 53. — С. 32—54.
3. *Назаров, Д. И.* Поиск информации о принимаемых сервером запросах в закомментированном клиентском коде веб-приложений / Д. И. Назаров, Д. А. Сигалов, Д. Ю. Гамаюнов // Программная инженерия. — 2023. — Т. 14, № 5. — С. 245—253.
4. *Миронов, Д. Д.* Исследование встречаемости небезопасно сериализованных программных объектов в клиентском коде веб-приложений / Д. Д. Миронов, Д. А. Сигалов, М. П. Мальков // Труды Института системного программирования РАН. — 2023. — Т. 35, № 1. — С. 223—236.
5. *Sigalov, D.* Dead or alive: Discovering server HTTP endpoints in both reachable and dead client-side code / D. Sigalov, D. Gamayunov // Journal of Information Security and Applications. — 2024. — Vol. 82. — P. 103746. — URL: https://www.sciencedirect.com/science/article/pii/S2214212624000498.
6. *Сигалов, Д. А.* Использование отладочного API современного веб-обозревателя для обнаружения уязвимостей класса DOM-based XSS / Д. А. Сигалов, А. В. Раздобаров, А. А. Петухов // Прикладная дискретная математика. — 2017. — № 35. — С. 63—75.
7. *Sigalov, D.* Finding Server-Side Endpoints with Static Analysis of Client-Side JavaScript / D. Sigalov, D. Gamayunov // Computer Security. ESORICS 2023 International Workshops. — Springer Nature Switzerland, 2024. — P. 442—458.

Сигалов Даниил Алексеевич

Методы выявления поверхности атаки веб-приложений при помощи анализа клиентского JavaScript-кода

Автореф. дис. на соискание ученой степени канд. тех. наук

Подписано в печать ___.___._____. Заказ № _____
Формат 60×90/16. Усл. печ. л. 1. Тираж 100 экз.
Типография _____