МАТЕМАТИЧЕСКИЕ ОСНОВЫ КОМПЬЮТЕРНОЙ БЕЗОПАСНОСТИ

УДК 004.056.53

ОБНАРУЖЕНИЕ СЕРВЕРНЫХ ТОЧЕК ВЗАИМОДЕЙСТВИЯ В ВЕБ-ПРИЛОЖЕНИЯХ НА ОСНОВЕ АНАЛИЗА КЛИЕНТСКОГО JavaScript-КОДА[1]

Д. А. Сигалов, А. А. Хашаев, Д. Ю. Гамаюнов

МГУ имени М. В. Ломоносова, г. Москва, Россия

Рассматривается задача обнаружения серверных точек взаимодействия в динамических веб-приложениях в контексте анализа защищенности веб-приложений в модели «черного ящика». Предложен метод повышения полноты обнаружения серверных интерфейсов на основе статического анализа клиентского кода JavaScript для поиска в нем функций, которые порождают HTTP-запросы к серверной стороне приложения, и определения возможных значений параметров найденных функций. В контексте решаемой задачи статический анализ позволяет находить такие функции в том числе в недостижимом или мёртвом JavaScript-коде, что в ряде случаев позволяет обнаружить серверные интерфейсы, скрытые для динамического анализа. Проведено экспериментальное исследование полноты выявления серверных точек взаимодействия предложенным алгоритмом на синтетическом веб-приложении, уязвимом к SQL-инъекции, и сравнение с популярными сканерами защищенности веб-приложений. Показано, что использование статического анализа клиентского JavaScript-кода в дополнение к традиционному динамическому краулингу приложений может значительно повысить полноту выявления серверных точек взаимодействия в веб-приложениях.

Ключевые слова: *веб-приложения, статический анализ, JavaScript.*

DOI 10.17223/20710410/53/3

DETECTING SERVER-SIDE ENDPOINTS IN WEB APPLICATIONS BASED ON STATIC ANALYSIS OF CLIENT-SIDE JavaScript CODE

D. A. Sigalov, A. A. Khashaev, D. Yu. Gamayunov

Lomonosov Moscow State University, Moscow, Russia

E-mail: {asterite, arthur, gamajun}@seclab.cs.msu.ru

[1]В работе использованы результаты проекта «Система автоматического поиска уязвимостей в веб-приложениях на основе обработки больших данных», выполняемого в рамках реализации Программы Центра компетенций Национальной технологической инициативы «Центр хранения и анализа больших данных», поддерживаемого Министерством науки и высшего образования РФ по Договору МГУ имени М. В. Ломоносова с Фондом поддержки проектов Национальной технологической инициативы от 15.08.2019 № 7/1251/2019.

The problem of server-side endpoint detection in the context of blackbox security analysis of dynamic web applications is considered. We propose a method to increase coverage of server-side endpoint detection using static analysis of client-side JavaScript code to find functions which generate HTTP requests to the server-side of the application and reconstruct parameters for those functions. In the context of application security testing, static analysis allows to find such functions even in dead or unreachable JavaScript code, which cannot be achieved by dynamic crawling or dynamic code analysis. Evaluation of the proposed method and its implementation has been done using synthetic web application with endpoints vulnerable to SQL injections, and the same application was used to compare the proposed method with existing solutions. Evaluation results show that adding JavaScript static analysis to traditional dynamic crawling of web applications may significantly improve server-side endpoint coverage in blackbox application security analysis.

Keywords: *web applications, static analysis, JavaScript.*

Введение

Рассматривается задача обнаружения уязвимостей в веб-приложениях в модели «чёрного ящика», когда для анализа доступна только клиентская часть веб-приложения и интерфейсные точки взаимодействия на его серверной стороне. В этой модели процесс поиска уязвимостей обычно состоит из трёх этапов: поиска доступных функций веб-приложения (их также иногда называют серверными точками взаимодействия, серверными интерфейсами, точками входа в приложение или API-вызовами); вызова функций приложения с различными значениями параметров; анализа результатов выполнения действий [1]. Данная работа посвящена автоматизации поиска доступных функций веб-приложения.

Поиск доступных функций мы сводим к поиску так называемых *точек ввода данных* (DEP — Data Entry Points) [1]. Под DEP будем понимать *спецификацию* допустимого для некоторой функции веб-приложения множества HTTP-запросов, заданную как совокупность фиксированных элементов запроса и ограничения на вариативную часть запроса. Фиксированная часть запроса идентифицирует DEP (часто это URL и метод), вариативная является совокупностью значений его параметров (часто это query-параметры, тело запроса, значения заголовков). Объединение множеств порождаемых всеми DEP запросов веб-приложения даёт множество всех допустимых HTTP-запросов, которые принимает сервер.

Множество допустимых HTTP-запросов полностью определяется кодом серверной части приложения и, возможно, его динамическим состоянием (конфигурацией, историей функционирования). При анализе веб-приложения как чёрного ящика эта информация недоступна, поэтому множество допустимых DEP определить полно и точно не представляется возможным, но можно решать задачу построения аппроксимирующего множества DEP. DEP полностью определяют пространство атаки для исследуемого веб-приложения, поэтому при поиске DEP наиболее важным критерием качества является полнота.

Строить аппроксимацию множества DEP можно следующими способами:

— статическим анализом HTML в ответах сервера и выявлением статических элементов интерфейса, которые могут порождать запросы к серверу;
— динамическим обходом приложения с помощью управляемого браузера (Selenium Web Driver, Headless Chrome, Firefox, WebKit, htmlunit и др.);

— статическим анализом JavaScript для поиска вызовов `XMLHttpRequest()`, `fetch()` и библиотечных обёрток вокруг браузерного API;
— активным управляемым перебором запросов к серверной стороне приложения и анализом ответов для обнаружения DEP, недоступных из интерфейса; примером такого перебора является поиск URL-путей по словарю, так называемый дирбастинг, по результатам работы которого может также использоваться фаззинг.

Дирбастинг или фаззинг в общем случае не позволяют определить имена параметров HTTP-запроса и возможные диапазоны их значений и обычно направлены на поиск функций по словарю значений фиксированной части DEP [2]. Остальные методы так или иначе получают информацию из пользовательского интерфейса приложения, т. е. HTML-страниц и подключаемых ими ресурсов.

Сложность аппроксимации DEP по интерфейсу существенно различается для статических и динамических интерфейсов: в случае статического интерфейса набор видимых серверных точек взаимодействия однозначно определяется HTML, семантика элементов разметки и их реакция на действия пользователя всегда фиксированы.

В случае динамического интерфейса использование встроенных программ на языке JavaScript расширяет его функциональность, в частности программа может инициировать отправку запроса на сервер, а также динамически создавать, изменять и удалять элементы HTML-разметки страницы. В результате часть допустимых HTTP-запросов формируется в результате выполнения JavaScript-программы, и аппроксимация DEP лишь на основе HTML будет существенно неполной. При этом, по данным статистики, 97,1 % веб-сайтов используют на своих страницах JavaScript [3].

Предельным случаем динамического интерфейса в настоящее время являются так называемые *single-page applications* (SPA), в которых при первом обращении к веб-приложению один раз загружается HTML-страница с JavaScript-программой и в дальнейшем все запросы с клиента порождаются только в результате выполнения этой программы.

Функции веб-приложения можно классифицировать по их доступности из веб-интерфейса: публичные, пользовательские и скрытые функции [2]. Интерфейсная часть публичных функций доступна любому пользователю, возможно, после обращения к некоторым другим публичным функциям. Пользовательские функции, как правило, содержатся в защищённой части веб-приложения, которая доступна только аутентифицированному пользователю. Отметим следующую особенность динамических интерфейсов: если для статических интерфейсов, как правило, соблюдается изоляция функций, то в динамических интерфейсах JavaScript-программы часто нарушают такую изоляцию [2]. Наконец, скрытые функции полностью недоступны в интерфейсе.

Примером пользовательских функций является административная панель управления веб-приложением. Анонимный пользователь не увидит ссылку на административную панель или форму смены пароля администратора в своём интерфейсе (по крайней мере, пока не пройдёт аутентификацию). При этом, как упоминается в [2], возможно такое, что клиентский JavaScript-код, соответствующий административной панели, будет доступен анонимному пользователю. Примером скрытых функций может быть старая версия функции поиска по сайту, которая была удалена из пользовательского интерфейса, но при этом продолжает поддерживаться сервером — например, потому, что запланированное обновление кода сервера ещё не произошло, либо потому, что разработчики не сочли нужным удалять её из сервера вообще. Другим примером может служить отладочный интерфейс на сервере, который используется разработчиками

для получения диагностики о состоянии сервера и его отладки и не предназначен для использования рядовыми пользователями сайтов.

В современных сканерах безопасности чаще всего используется динамический обход в качестве основного способа поиска серверных точек взаимодействия, при этом у динамического обхода есть недостатки: он не позволяет обнаруживать скрытые функции, может не обнаружить публичные функции, требующие сложной навигации по интерфейсу, и плохо работает для автоматического обнаружения пользовательских функций, так как требует аутентификации в разных пользовательских ролях, что в модели «чёрного ящика» может быть сложно или невозможно. Мы протестировали ряд популярных сканеров безопасности веб-приложений и убедились, что во многих случаях они не находят пользовательские и скрытые функции, даже когда соответствующий им JavaScript-код доступен на странице. Сканеры не находили даже простые функции, где вызывающий их JavaScript содержал URL в явном виде и параметры запроса в виде констант, локализованных в одном линейном участке клиентского кода (полные результаты описаны в п. 4).

В данной работе предлагается подход к поиску DEP на основе статического анализа JavaScript-кода клиентской части приложения. Алгоритм ищет вызовы функций, порождающих запросы на сервер, и пытается определить возможные значения аргументов найденных функций с помощью статического анализа, работающего над абстрактным синтаксическим деревом JavaScript-кода.

1. Особенности реализации JavaScript DEP в реальных приложениях

Характерной особенностью работ по анализу JavaScript-кода является то, что авторы в явной или неявной форме делают ряд предположений об устройстве реальных JavaScript-программ и этим формируют ограниченное подмножество языка, подлежащее анализу. Эти предположения используются для обоснования практической применимости предлагаемых методов анализа, которые не являются формально полными или корректными — например, не моделируют некоторые особенности и возможности языка либо имеют принципальные технические ограничения по скорости работы и объёму используемой памяти. Иными словами, исследователи предполагают, что на практике в реальных приложениях не используется целый ряд возможностей языка JavaScript, а потому и методы анализа JavaScript могут игнорировать наличие таких возможностей без ущерба для точности и полноты анализа.

В [4] исследуется вопрос о применимости таких предположений к анализу кода общего вида. В качестве анализируемых веб-сайтов и приложений выбраны 100 сайтов из наиболее популярных по версии Alexa [5]. В настоящей работе рассматриваются частные вопросы, относящиеся к анализу точек ввода данных.

HTTP-запрос из JavaScript-кода в браузерном окружении можно сделать следующими способами:

- явным:
  - используя API `XMLHttpRequest`;
  - используя функцию `fetch`.
- неявным:
  - статически: отправив форму, заданную в статической HTML-разметке страницы;
  - динамически: например, создав и отправив форму, изменив свойство `window.location`, выполнив вызов `window.open` и так далее.

Отметим, что неявный статический способ не представляет сложности для анализа, а динамический тяжело поддаётся полному описанию, поэтому в рамках данной работы рассматриваются исключительно явные способы. Стоит также отметить, что разработчики приложений часто используют библиотеки, такие, как jQuery [6], которые предоставляют обёртки (например, функцию `$.ajax`) над браузерным API для совершения HTTP-запросов.

1.1. Мотивирующие примеры

Рассмотрим следующие примеры:

1) На веб-сайте присутствует HTML-форма, при отправке которой вызывается зарегистрированный JavaScript-обработчик, который выполняет HTTP-запрос на сервер. При этом используется решение reCAPTCHA [7] для защиты формы от отправки автоматизированными средствами (ботами). Однако валидация CAPTCHA производится исключительно на стороне клиента, в этом обработчике, что само по себе является уязвимостью. Такая конфигурация не анализируется динамическими средствами, основанными на управляемом браузере, поскольку требует автоматизированного решения CAPTCHA. Вместе с тем статический анализ мог бы проанализировать путь исполнения программы, в которой условие проверки CAPTCHA на клиентской стороне выполнено, и таким образом обнаружить целевую точку ввода данных.

2) На веб-сайте присутствует JavaScript-сценарий из листинга 1. Если предположить, что функция `window.__makePostCallback` не вызывается, то динамический анализ также не обнаружит скрытую функцию, относящуюся к данной точке ввода данных. Отметим также, что в строке 15 присутствует нетривиальная проверка содержимого Cookie, которую статический анализ, в отличие от динамического, может проигнорировать.

```
1  var baseURL = "/site/",
2      config = {
3          section: "forum",
4          apiVersion: "1.1",
5          pageType: "forumPage",
6          trackingSegment: 374000000,
7      },
8      pageInfo = "type=" + config.pageType + "&page=" + location;
9  (function() {
10     function makePost(user, title, postText) {
11         var actionEndpoint = "post.php";
12         var tracking = pageInfo + "&trid=" + trackingID +
13                        "&u=" + user;
14         var sender = function() {
15             if (!document.cookie.includes("uname=" + user))
16                 throw "ERROR";
17             var method = "POST",
18                 apiVer = config.apiVersion,
19                 urlPrefix = baseURL + config.section +
20                             "/api/" + apiVer;
21             $.ajax(urlPrefix + "/" + actionEndpoint +
22                    "?" + tracking, {
23                 method: method,
24                 data: postParams,
25             });
26         }
27         var postParams = {"title": title, "text": postText};
```

```
28          $("#confirm-btn").on("click", sender);
29      }
30      window.__makePostCallback = makePost;
31  })();
32  trackingID = config.trackingSegment + 22293;
```

Листинг 1. Пример кода, трудно поддающегося динамическому анализу

Из данных примеров видно, что динамический анализ обладает рядом ограничений в контексте задачи поиска функций веб-приложения, особенно скрытых.

1.2. Сложность задачи анализа JavaScript-кода

Для построения аппроксимации DEP для JavaScript-кода нужно, как минимум, уметь находить вызовы методов, отправляющих запросы на сервер, и определять возможные наборы значений аргументов этих вызовов. В литературе упоминается ряд трудностей, делающих сложным решение подобных задач для JavaScript-кода [8 – 10]. К ним относятся:

— динамическое выполнение кода с помощью функций `eval`, `Function` и некоторых других;
— динамическая типизация с неявным динамическим приведением типов;
— обращение к полям объектов по динамически вычисленному имени поля;
— динамическое добавление и удаление свойств объектов;
— изменение прототипов встроенных в язык классов, позволяющее изменить семантику встроенных в стандартную библиотеку языка методов;
— работа с функциями как со значениями, в т. ч. присвоение их переменным и полям объектов и передача в качестве аргументов функций;
— динамическое создание областей видимости с помощью выражения `with`.

Эти трудности могут делать анализ особенно сложным в комбинации друг с другом. Например, как отмечается в [11], если в анализируемом участке программы свойство глобального объекта берётся по вычисленному имени и есть вызов взятого значения как функции и если при этом алгоритму анализа не удаётся найти имя свойства и значения аргументов функции (т. е. они остаются неизвестными), то возможны ситуации, когда именем свойства будет строка «`eval`». Тогда последующий вызов приведёт к динамическому выполнению какого-то неизвестного кода. Схожий пример приводится в [10].

Для клиентского JavaScript-кода ещё одной проблемой является взаимодействие анализируемого кода с объектной моделью документа (DOM-моделью) [10] — кроме того, что её моделирование в анализаторе само по себе является трудоёмкой задачей, какие-то части DOM-модели определяются пользователем (например, значение заполненного пользователем поля формы), и в ходе статического анализа эти данные неизвестны, что является принципиальным ограничением для любого метода статического анализа.

Существующие исследования показывают, что осложняющие статический анализ конструкции языка JavaScript существуют не только в теории, но используются в реальных веб-приложениях [4].

1.3. Модельный пример

Рассмотрим следующий модельный пример веб-приложения, в клиентском JavaScript-коде которого присутствует фрагмент из листинга 2. Обработчик `remove(id)` использует библиотеку jQuery для выполнения HTTP-запроса и вызывается при клике

на кнопку удаления сообщения с идентификатором `id`. Обработчик POST-запроса на стороне сервера уязвим к атаке SQL-инъекцией в параметре `ident`.

```
1 var api = "/application/iuT6ei/";
2 function remove(id) {
3     if (prompt("Enter 'yes' to remove") !== "yes") return;
4     $.post(api + "interface/remove/handle", {ident: id});
5 }
```

Листинг 2. Модельный пример DEP

Данный DEP не анализируется динамическими методами, поскольку для выполнения запроса необходимо пройти проверку в условном операторе. Этот же DEP не анализируется статическими методами существующими сканерами безопасности, как показали наши эксперименты (см. п. 4).

2. Существующие работы

Рассмотрим две группы работ, наиболее близкие к решаемой задаче: исследования в области сканеров безопасности динамических веб-приложений и в области статического анализа программ на языке JavaScript, в том числе в контексте задач обнаружения программных уязвимостей.

В 2010 г. Адам Дюпе с коллегами из университета Калифорнии в Санта-Барбаре опубликовали исследование ограничений современных на тот момент сканеров безопасности при работе с динамическими приложениями [12]. Они продемонстрировали, что для обнаружения большинства типов уязвимостей от сканеров требуется качественная поддержка динамического обхода (краулинга) веб-приложений, поддержка исполнения JavaScript, при этом в исследованных сканерах были выявлены ошибки в реализации парсеров HTML, недостаточная поддержка JavaScript, в результате из 16 уязвимостей 8 не были обнаружены ни одним из сканеров. Среди исследованных сканеров были Acunetix, AppScan, Burp, w3af, WebInspect, которые существуют и активно развиваются и в настоящее время. В 2012 г. Адам Дюпе с коллегами опубликовали работу по теме автоматического обнаружения уязвимостей в веб-приложениях [13], а в 2014 г. он защитил диссертацию по той же теме [14]. В диссертации предложена архитектура сканера безопасности для динамических веб-приложений, в которой строится модель состояний анализируемого приложения в виде автомата Мили, где входные символы интерпретируются как запросы сканера, а состояния определяются по ответам. Основной акцент сделан на динамическом краулинге для достижения полноты обхода состояний и никак не затронут вопрос анализа клиентского кода JavaScript.

В эти же годы направление динамического краулинга веб-приложений активно изучается в других группах, в частности авторами IBM AppScan и Crawljax. В [15] исследованы различные стратегии динамического обхода веб-приложений, которые могут быть применимы для полноценного функционального тестирования приложений, т. е. критерием также является полнота покрытия кода серверной части приложения, но при этом обход и анализ выполняются методом «чёрного ящика». Следует отметить работу [16], в которой описан один из широко используемых динамических краулеров — Crawljax. Работа посвящена сложности задачи динамического обхода современных веб-приложений, которые активно используют язык JavaScript на стороне клиентской части приложения. Основная цель работы совпадает с предыдущей: обеспечить полноту функционального тестирования веб-приложений.

Эти работы всецело сфокусированы на задаче динамического краулинга веб-приложений, в том числе в контексте поиска уязвимостей, и не рассматривают статический

анализ клиентского кода JavaScript в качестве альтернативного способа извлечения знаний о структуре серверной стороны веб-приложения. Если поставить перед собой такую задачу, то логичный способ её решения — использование одного из существующих статических анализаторов кода для языка JavaScript, например SAFE, WALA или TAJS. Мы рассмотрели некоторые работы, в которых авторы статических анализаторов или сторонние исследователи пытались применить их для анализа реальных веб-приложений в Интернете.

При сравнительном анализе статических анализаторов для построения графа вызовов по JavaScript-программе отмечено, что далеко не все анализаторы дают адекватный API для доступа к объектам внутреннего представления, в частности SAFE и JSAI по этой причине оказались неприменимы для решаемой задачи [17].

В [18] сделана попытка исследования кода больших интернет-приложений с помощью существующих статических анализаторов JavaScript. Авторы утверждают, что полные (sound) методы анализа неприменимы на практике. Они опробовали схему анализа, которая использует комбинацию двух статических анализаторов — результаты работы быстрых, но неточных и неполных методов анализа из WALA используются далее в качестве начальных данных для более точного контекстно-чувствительного анализа средством SAFE. В экспериментах анализируется код пяти популярных библиотек (jQuery, MooTools, Prototype, YUI, Underscore) и код с заглавной страницы пяти реальных сайтов (live.com, wikipedia.org, facebook.com, youtube.com, baidu.com). Показано, что с анализом успешно справляется только комбинация анализаторов SAFE и WALA. Иными словами, возможность анализа кода реальных веб-приложений достигается только путём заведомого снижения полноты и точности анализа ради его масштабируемости.

В [10] исследована задача повышения контекстной чувствительности анализа циклов для анализатора SAFE на примере корпуса реальных приложений. Эксперименты проводились на библиотеках jQuery, Modernizr, BootStrap, Mootools и Prototype, а также на JavaScript-коде с главных страниц сайтов google.com, facebook.com, youtube.com, baidu.com и yahoo.com. Авторам удалось показать улучшение на части кода библиотек, но для реальных сайтов анализ оказался неприменим, время работы программы составило в среднем 3500–7000 с.

Отдельно следует упомянуть работу [19], в которой предложен способ проверки соответствия API-запросов в коде на JavaScript спецификации соответствующего серверного API. Для этого предлагается при помощи статического анализа JavaScript извлекать из точек обращения к API соответствующие URL, метод и параметры запросов и сопоставлять спецификации в формате OpenAPI/Swagger для проверки корректности использования API в клиентском коде. Авторы использовали построение упрощённого field-based CFG, идентификацию точек обращения к API лексическим поиском конструкций jQuery и определение параметров запроса с помощью построения обратных программых срезов с анализом потока данных средствами WALA. Тестирование показало применимость в том числе для приложений, которые используют jQuery. Однако следует понимать, что из-за специфики анализа (отсутствия чувствительности к путям и контексту) параметры запроса получались как комбинация всевозможных значений различных переменных вдоль разных путей в программе, формирующих данный запрос. На практике это ведёт к большому количеству возможных значений параметров для каждого из запросов, что не критично в контексте решаемой авторами задачи, но может быть сильным ограничивающим фактором для задачи поиска уязвимостей.

Использование полноценных статических анализаторов JavaScript-кода представляется весьма перспективным направлением развития сканеров безопасности веб-приложений, но, к сожалению, в настоящее время их использование для этой задачи представляется непрактичным, в первую очередь из-за большой вычислительной сложности и в некоторых случаях недостаточной точности анализа.

3. Метод анализа JavaScript-кода для поиска DEP

Как уже было сказано, существующие средства анализа JavaScript-кода, как встроенные в сканеры безопасности, так и самостоятельные, плохо справляются с задачей поиска DEP на страницах современных веб-приложений. Опишем новый метод анализа JavaScript-кода с целью поиска серверных точек взаимодействия в приложениях. В п. 4 приведены результаты его сравнения с существующими средствами. Метод обнаруживает только точки взаимодействия, порождаемые вызовами JavaScript-функций.

Метод работает над веб-страницей: он принимает на вход URL страницы, её HTML-разметку и набор подключаемых этой разметкой внешних ресурсов. Результатом является множество спецификаций отправляемых на сервер запросов, найденных в ходе анализа. Это множество считается аппроксимацией DEP, к которым обращается JavaScript-код страницы.

Метод анализа статический, нечувствительный к управляющим конструкциям (*path-insensitive*), однако чувствительный к порядку инструкций (*flow-sensitive*). Делается также попытка в простых случаях определить значения передаваемых в вызываемые функции аргументов с помощью анализа цепочки вызовов, что добавляет некоторую чувствительность к контексту вызова (*context-sensitivity*).

Концептуально метод работает следующим образом: производится поиск вызовов функций, отправляющих запросы на сервер (далее для краткости будем называть их *AJAX-функциями*), и определение аргументов этих вызовов. Затем для каждого найденного вызова с данным набором аргументов определяется, какой вид имеют запросы, сделанные таким вызовом, — формируются спецификации запросов.

Метод работает над абстрактным синтаксическим деревом кода (Abstract Syntax Tree, AST). Поиск вызовов делается с помощью рекурсивного обхода дерева в глубину; вызовы обнаруживаются простым сигнатурным методом — проверяется вхождение имени вызываемой функции (и, если это вызов метода объекта, имени объекта) в заранее заданный список имён (*набор сигнатур*). В этот список входят, например, имена `fetch` и `axios`, пары `($, ajax)` и `($http, post)`.

3.1. Моделирование переменных

Метод корректно моделирует области видимости переменных, используя *модель лексических областей видимости*. Это отображение, сопоставляющее каждому использованию идентификатора — имени переменной в программе — специальное значение, уникальное для переменной (*binding*). Оно одинаково для идентификаторов, относящихся к одной и той же переменной, и разное для идентификаторов, относящихся к разным переменным, в т. ч. в случае, когда имена совпадают. Модель учитывает границы областей видимости, перекрытия одноимённых переменных и вложенные области видимости. В условиях отсутствия конструкций `eval` и `with` (которые могут добавлять имена в области видимости динамически) задача построения подобной модели является достаточно простой и может быть решена статически. В предлагаемой реализации метода для её решения используется библиотка Babel [20]. В дальнейшем, когда мы будем говорить о *переменной*, будем иметь в виду уникально идентифицируемую с помощью модели лексических областей видимости переменную.

Для моделирования значений переменных используется *модель памяти* — отображение, ставящее в соответствие переменным значения. Каждой переменной может быть поставлено в соответствие только одно значение. Это ограничение приводит к тому, что некоторые элементы семантики языка JavaScript не могут быть промоделированы алгоритмом корректно, так как при выполнении JavaScript-программ одной и той же с точки лексики переменной могут соответствовать несколько значений. Примерами являются локальные переменные и аргументы рекурсивных функций, а также переменные, попавшие в замыкание вложенной функции. Для всех таких случаев модель памяти будет содержать для каждой переменной только какое-то одно значение, что является ограничением описываемого метода.

3.2. Вычисление выражений

Метод использует механизм вычисления JavaScript-выражений. Он поддерживает выражения, содержащие литералы, переменные, операцию +, доступ к полю объекта, вызовы некоторых встроенных в браузер функций и ещё некоторые операции. При использовании в выражениях переменных делается попытка взять их значения из модели памяти.

В случае, если конкретное значение выражения вычислить не удалось (например, используется переменная, для которой в модели памяти нет значения, или в выражении используется неподдерживаемая анализатором конструкция), вырабатывается *неизвестное значение* — `Unknown`. Особым случаем является использование формального аргумента функции, значение которого неизвестно — при этом вырабатывается значение `FromArg`, которое является подвидом неизвестного значения, но дополнительно сигнализирует о том, что данные пришли именно из формального аргумента функции. Это используется для того, чтобы принять решение о необходимости анализа с учётом цепочек вызова.

Вычисление выражений делается следующим образом. Для большинства операций, если все участвующие в выражении значения конкретны, то выполняется соответствующая операция в оригинальной семантике JavaScript.

Если в выражении участвуют значения `Unknown` или `FromArg`, то при простом присваивании переменной или полю объекта и при передаче в качестве аргумента функции они остаются без изменений, а в остальных случаях либо приводятся к конкретным значениям (так делается при операциях, в ходе которых значение приводится к строке), либо результат всего выражения становится `Unknown` (или `FromArg`, если это значение `FromArg` участвовало в вычислении). При приведении к конкретным значениям значение `Unknown` становится строкой «`UNKNOWN`», а значение `FromArg` — строкой «`FROM_ARG`». В конце работы алгоритма все вхождения `FromArg` в результирующих данных заменяются на `Unknown`, а все вхождения строки «`FROM_ARG`» — на «`UNKNOWN`».

Ключевое слово `this` текущей версией алгоритма не поддерживается — при его обработке будет выработано значение `Unknown`.

3.3. Алгоритм анализа

Алгоритм анализа (алгоритм 1) получает на вход URL-адрес страницы, AST-дерево содержащегося в ней JavaScript-кода и модель лексических областей видимости. AST-дерево можно получить конкатенацией текстов всех JavaScript-программ на странице и последующим синтаксическим разбором полученной программы с помощью библиотеки Babel [20]. Модель лексических областей видимости также может быть получена с помощью Babel на основе AST-дерева. Результатом анализа является набор пар, описывающих найденные вызовы и их аргументы. Каждая пара состоит из сигнатуры

функции, вызов которой был найден, и списка аргументов её вызова. Этот набор будет далее переработан в набор спецификаций HTTP-запросов, как описано в п. 3.4.

Алгоритм 1. Анализ

Вход: `PageURL`, `AST`, `ScopeModel`.
Выход: `CallDescrs`.

```
1: MemModel := SeedInitialMemModel(PageURL).
2: MemModel := GatherVarValues(AST, MemModel, ScopeModel).
3: Queue := [].
4: CallDescrs, Queue := ExtractDEPs(AST, MemModel, ε, Queue).
5: Пока |Queue| > 0:
6:    вытолкнуть первый элемент из Queue в (callerAST, Chain);
7:    newCallDescrs, Queue := ExtractDEPs(callerAST, MemModel, Chain, Queue);
8:    CallDescrs := CallDescrs ∪ newCallDescrs.
```

Основными шагами анализа являются поиск значений переменных, которому в псевдокоде соответствует функция GatherVarValues (алгоритм 2), и поиск вызовов AJAX-функций и их аргументов, которому соответствует код в строках 3–8 алгоритма 1. Его основная часть, включая непосредственно обход AST для поиска интересующих мест вызова и вычисление их аргументов, содержится в функции ExtractDEPs (её псевдокод приведён в алгоритме 3).

Алгоритм 2. Поиск значений переменных и присваивание значения

```
1: Функция Assign(v, MemModel)
2: Если v это VariableDeclarator и ∃ v.init и v.id это Identifier, то
3:    b := GetBinding(v.id, ScopeModel);
4:    value := EvalExpr(v.init, MemModel, ScopeModel);
5:    MemModel := MemModel ∪ (b ↦ value),
6: иначе если v это AssignmentExpression, то
7:    value := EvalExpr(v.right, MemModel, ScopeModel).
8:    Если v.left это Identifier, то
9:       b := GetBinding(v.left, ScopeModel);
10:      MemModel := MemModel ∪ [b ↦ value],
11:   иначе если v.left это MemberExpression, то
12:      AssignProp(v.left, value).
13: Вернуть MemModel.
14: Функция GatherVarValues(AST, MemModel, ScopeModel)
15: Для всех вершин v ∈ AST в порядке обхода в глубину:
16:   Если v это VariableDeclarator или v это AssignmentExpression, то
17:      MemModel := Assign(v, MemModel),
18:   иначе если v это FunctionDeclaration, то
19:      b := GetBinding(v.id, ScopeModel);
20:      MemModel := MemModel ∪ [b ↦ FuncVal(v)].
21: Вернуть MemModel.
```

Алгоритм 3. Поиск вызовов и их аргументов

```
1: Функция ExtractDEPs(AST, MemModel, Chain, Queue)
2: d := 0.
3: Для всех вершин v ∈ AST в порядке обхода в глубину:
4:   Если v это CallExpression, то
5:     Если ∃ sign ∈ SignatureSet: MatchesSignature(v, sign), то
6:       vals := [EvalExpr(arg, MemModel, ScopeModel) | arg ∈ v.arguments];
7:       vals, hadArgsDependency := CheckAndRemoveFromArg(vals);
8:       Results := Results ∪ {(sign, vals)}.
9:       Если hadArgsDependency, то
10:        Queue := BuildCallChains(v, AST, Queue, ScopeModel, MemModel).
11:    Если |Chain| > 0 и v.callee это Identifier, то
12:      (binding, ast, args), rest := Chain.
13:      Если binding = GetBinding(v.callee, ScopeModel), то
14:        act, MemModel := SetActualArgVals(func, args, MemModel);
15:        newResults, newQueue := ExtractDEPs(ast, MemModel, rest, Queue);
16:        Results := Results ∪ newResults;
17:        Queue := Queue + newQueue;
18:        MemModel := MemModel \ act,
19:  иначе если v это Function то
20:    d := d + 1;
21:    formal := {(GetBinding(a, ScopeModel) ↦ FromArg) | a ∈ v.arguments};
22:    MemModel := MemModel ∪ formal,
23:  иначе если d > 0 и v это {VariableDeclarator, AssignmentExpression}, то
24:    MemModel := Assign(v, MemModel).
25:  Если вышли из вершины v′ и v′ это Function, то
26:    d := d − 1;
27:    formal := {(GetBinding(a, ScopeModel) ↦ FromArg) | a ∈ v.arguments};
28:    MemModel := MemModel \ formal.
29: Вернуть Results, Queue.

30: Функция SetActualArgVals(func, args, MemModel)
31: vals := [EvalExpr(arg, MemModel, ScopeModel) | arg ∈ args];
32: act := {(GetBinding(a_i, ScopeModel) ↦ vals_i) | 0 ⩽ i < |func.arguments|};
33: MemModel := MemModel ∪ act.
34: Вернуть act, MemModel.

35: Функция BuildCallChains(v, AST, Queue, ScopeModel)
36: funcAST := GetCallerAST(v);
37: funcBindings := GetBindingsForAST(funcAST, ScopeModel, MemModel).
38: Для всех binding ∈ funcBindings:
39:   callSites := FindCallSites(AST, binding, ScopeModel).
40:   Для всех site ∈ callSites:
41:     chain := [(binding, funcAST, funcAST.arguments)] + Chain;
42:     callerAST := GetCallerAST(site);
43:     Queue := Queue + [(callerAST, chain)].
44: Вернуть Queue.
```

Можно заметить, что функция GatherVarValues учитывает только присваивания переменных, инициализации и объявления функций. Другие способы передачи значений, например задание явно аргументов самовызывающейся функции (IIFE), обработаны на этом этапе не будут.

Функция ExtractDEPs работает следующим образом: делается рекурсивный обход AST-дерева, при котором вычисляются выражения, при обнаружении присваиваний или инициализаций переменных на неглобальном уровне (т. е. внутри тела какой-то функции) они обрабатываются аналогично GatherVarValues, пополняя модель памяти. При обнаружении вызова функции, отвечающего одной из имеющихся сигнатур AJAX-функции, выражения, задающие аргументы вызова, вычисляются и набор аргументов вместе с сигнатурой запоминается и становится частью результата работы функции. При этом проверяется, встречаются ли в вычисленных аргументах значения `FromArg` или «`FROM_ARG`». Если это так, то регистрируется зависимость передаваемых в AJAX-вызов данных от формальных аргументов вызывающией функции. Обход AST-дерева делается рекурсивно в глубину. Все вершины посещаются при одном обходе только один раз, циклы и ветвления не влияют на него (т. е. тело цикла будет посещено единожды, обе ветви оператора `if` будут посещены безусловно).

Как видно в псевдокоде, функция ExtractDEPs вызывается в алгоритме несколько раз. Это делается для анализа с учётом цепочек вызова — в случае, если обнаружено, что параметры интересующего AJAX-вызова зависят от формальных аргументов функции, делается попытка найти места её вызова и выполнить анализ, начиная с каждого из этих мест (точнее, от начала тела функции, в которой место вызова найдено). В общем случае аргументы функции в месте вызова могут зависеть от формальных аргументов другой функции, тогда будет выполнен поиск, в свою очередь, места её вызова (и, таким образом, будут анализироваться уже цепочки из двух вызовов), и так далее. Самый первый вызов ExtractDEPs в строке 4 алгоритма 1 осуществляет анализ без учёта цепочек вызова, дальнейшие вызовы в строке 7 делаются по одному на каждую выявленную цепочку.

Функция GetBinding получает из модели лексических областей видимости уникальное значение, идентифицирующее переменную, по месту её использования (см. п. 3.1).

Функция EvalExpr вычисляет выражение в соответствии с механизмом вычисления, описанном в п. 3.2. Функция AssignProp принимает значение и вершину AST типа MemberExpression (такие вершины соответствуют обращению к полю объекта), вычисляет объект и имя свойства и добавляет в этот объект свойство с таким именем и данным значением (или заменяет существующее, если свойство с таким именем уже было). Функция MatchesSignature осуществляет сопоставление места вызова с сигнатурой, возвращает значение «истина», когда тип сигнатуры совпадает с типом вызова (сигнатура может быть именем свободностоящей функции или парой из имени метода и имени объекта) и имя в сигнатуре совпадает с именем в вызове (в случае с вызовом метода — имена объекта и метода), в остальных случаях функция возвращает значение «ложь».

Функция GetCallerAST принимает на вход вершину AST-дерева и, если эта вершина соответствует коду внутри функции, возвращает корень поддерева, соответствующего телу этой функции, иначе — корень всего AST-дерева программы.

Функция GetBindingsForAST принимает на вход AST-дерево, соответствующее телу функции, и возвращает набор всех переменных, которым в модели памяти соответствуют функциональные значения этой функции. Эта функция работает с текущим

состоянием модели памяти — на тот момент, когда она была вызвана. В это время она наполнена в результате работы GatherVarValues, а также делавшихся до этого вызовов ExtractDEPs. Делается обход всех пар «ключ–значение», содержащихся в модели памяти, и выбираются те ключи, которым соответствует функциональное значение для функции, чьё AST-дерево подано на вход GetBindingsForAST. Поскольку в модели памяти каждой переменной в один момент времени может соответствовать только одно значение, если в ходе работы программы одной переменной присваивались разные значения, включая интересующее, эта переменная может не найтись и не попасть в результат (так как на момент вызова GetBindingsForAST в модели памяти этой переменной соответствует другое значение).

Функция FindCallSites для данной переменной возвращает набор всех вершин — мест вызова, где в качестве вызываемой функции используется эта переменная. Это делается с помощью модели лексических областей видимости и обхода AST-дерева программы: делается обход AST и для каждого места вызова, где вызываемая функция задана идентификатором, из модели лексических областей видимости берётся переменная, соответствующая этому идентификатору. Если она совпадает с переменной, поданной на вход функции FindCallSites, то рассматриваемое в данный момент место вызова добавляется в список — результат работы функции. Эта функция работает только с местами вызова, где функция дана идентификатором (`f(x, y)`). Если вызываемая функция задана обращением к полю объекта (т. е. это вызов метода, например `o.func(x)`) или другим выражением, то такое место вызова будет проигнорировано. В том числе не будут найдены вызовы самовызывающихся функций.

Функция CheckAndRemoveFromArg выполняет описанное в п. 3.2 удаление значений `FromArg` — она возвращает версию набора аргументов с удалёнными вхождениями `FromArg` (и строки «`FROM_ARG`»), а также булево значение — признак того, были ли такие значения найдены (и удалены) среди аргументов.

3.4. Формирование спецификаций DEP на основе найденных вызовов

Какие HTTP-запросы будут отправлены тем или иным вызовом с некоторым набором аргументов, мы определяем с помощью набора эвристических моделей функций, отправляющих запросы на сервер. Они моделируют работу поддерживаемых библиотек и встроенных в браузер механизмов в части отправки запросов на сервер, т. е. для каждой функции, для которой реализация метода содержит сигнатуру вызова, реализация также содержит написанный вручную код, который на основе аргументов этого вызова и исходного URL-адреса страницы определяет, какой запрос будет отправлен в результате этого вызова.

В результате применения этих сигнатур к набору найденных вызовов и их аргументов получается множество спецификаций запросов. Каждая спецификация содержит следующие поля: имя метода, URL-адрес, список HTTP-заголовков, тело запроса. Это множество является конечным результатом работы метода.

Следует отметить, что полученные спецификации в общем случае не соответствуют серверным точкам взаимодействия веб-приложения один-в-один. Два разных вызова отправляющей запрос на сервер функции могут обращаться к одной и той же серверной точке взаимодействия; также один и тот же вызов может обращаться к нескольким серверным точкам взаимодействия в зависимости от значений параметров — например, значение параметра query string в URL HTTP-запроса может влиять на маршрутизацию по микросервисам. В результате запросы, соответствующие нескольким разным

спецификациям в выводе метода, могут относиться к одной серверной точке взаимодействия, и наоборот, одной выходной спецификации могут соответствовать запросы к разным точкам взаимодействия. Как было сказано, при поиске серверных точек взаимодействия наиболее важным критерием качества является полнота. С точки зрения этого критерия первый из описанных эффектов является несущественным — в результате него один и тот же DEP будет повторен несколько раз. Второй, однако, может привести к проблемам, так как часть запроса, относящаяся к фиксированной части DEP, может быть ошибочно отнесена к вариативной. Заметим, что полностью решить эти проблемы при анализе веб-приложения как чёрного ящика в общем случае невозможно — множество допустимых DEP определить полно и точно нельзя, так как недоступна полная информация о серверной части веб-приложения.

3.5. Иллюстрация работы метода на примерах

Покажем, как работает алгоритм на двух примерах JavaScript-кода.

Сначала рассмотрим код листинга 1. Он содержит вызов функции `$.ajax`, отправляющий запрос на сервер. Этот пример имеет вид, характерный для реального JavaScript-кода на веб-страницах. Запрос делается функцией библиотеки jQuery, использование которой, в том числе для отправки запросов на сервер, чрезвычайно распространено. Кроме того, здесь используются вспомогательные функции-обёртки для отправки запросов, а основная часть кода «завёрнута» в анонимную самовызывающуюся функцию — это также характерно для клиентского JavaScript-кода в Интернете. Ещё одной часто встречающейся в реальном коде особенностью является использование глобальных переменных для задания конфигурации. Вместе с тем этот код представляет сложность для динамического анализа: чтобы управление дошло до вызова `$.ajax`, небходимо, чтобы последовательно произошли вызовы функций `makePost` и `sender`, а также чтобы прошла проверка в строке 15.

Чтобы определить, какой вид будет иметь запрос, анализатору необходимо как можно точнее определить возможные аргументы вызова `$.ajax`, находящегося в строке 21. Эти аргументы задаются как выражения, зависящие от переменных, некоторые из которых заданы выражениями от других переменных. Переменная `urlPrefix` (от которой зависит первый аргумент вызова) задаётся выражением, которое зависит от локальной переменной `apiVer`, а также глобальных переменных `baseURL` и `config`. Выражение, задающее первый аргумент вызова, также зависит от переменных, находящихся в области видимости объемлющей функции `makePost`. В анализаторе есть механизм вычисления таких выражений (см. п. 3.2).

Можно заметить, что `trackingID` и `postParams` в коде задаются после использования. Тем не менее при реальном выполнении они будут использованы уже после того, как получат свои значения — функция `makePost`, в которой используется переменная `trackingID`, может быть вызвана в произвольный момент после того, как присваивание `trackingID` в строке 29 будет выполнено. Функция, в которой используется переменная `postParams`, также может быть вызвана в произвольный момент после того, как будет завершён вызов `makePost`, в котором она задаётся. Алгоритм анализа способен использовать значения этих переменных при вычислении зависящих от них выражений благодаря фазе поиска значений переменных (в псевдокоде ей соответствует функция GatherVarValues, см. алгоритм 2) — на этой стадии при обходе AST-дерева программы будут обработаны присваивания переменных, в том числе инициализация переменных `baseURL`, `config` и `pageInfo`, инициализация `postParams` и присваивание `trackingID`. При этом заметим, что даже объявление переменной `postParams` находит-

ся в коде ниже места её использования. Тем не менее, по правилам языка JavaScript, переменная, объявленная с ключевым словом `var`, находится в области видимости во всём теле функции. Для переменной `trackingID` места объявления в программе вообще нет — в результате при присваивании в строке 32 будет создана глобальная переменная с таким названием. Обе эти особенности учитываются алгоритмом благодаря корректному моделированию с использованием модели лексических областей видимости.

Метод поддерживает специальную переменную `location`, встроенную в браузер и предназначенную для работы с URL-адресом страницы. Напомним, URL-адрес страницы является частью входных данных алгоритма. Для данных примеров кода будем считать, что URL-адрес страницы — это `http://example.com/`. Таким образом, возможно вычисление значения `pageInfo`, это будет строка «`type=forumPage&page=http://example.com/`».

После шага поиска значений переменных будет выполнен шаг обхода AST-дерева с целью поиска вызовов AJAX-функций и их аргументов. В псевдокоде этот обход выполняется в функции ExtractDEPs (см. алгоритм 3). На этом шаге присваивания переменных, находящиеся внутри функций, также обрабатываются (строки 23–24 алгоритма 3). Таким образом, в ходе этого шага будут вычислены значения `actionEndpoint`, `tracking`, `method`, `apiVer`, `urlPrefix` и, наконец, аргументы вызова `$.ajax`. При этом вычисляемые значения зависят от переменных, значения которых неизвестны, — формальных аргументов функции `makePost`. В качестве их значений будут взяты `FromArg`. Они допускают конкатенацию с конкретными строками, причём как в качестве префикса, так и суффикса, при этом информация о конкретной строке не теряется, при конкатенации неизвестное значение приводится к строке. В данном примере это позволяет анализатору найти имена параметров запроса.

Для определения того, какой вид будет иметь HTTP-запрос, отправленный вызовом функции `$.ajax` с такими аргументами, будет использована модель библиотеки jQuery. В итоге определим, что будет отправлен POST-запрос на URL-адрес `http://example.com/site/forum/api/1.1/post.php?type=forumPage&page=http://example.com/&trid=374022293&u=UNKNOWN` с телом `title=UNKNOWN&text=UNKNOWN`. Заметим, что в финальном результате нет вхождений «`FROM_ARG`», они заменены на «`UNKNOWN`», как описано в п. 3.2.

Обнаружение аргументов вызова `$.ajax` в данном примере является нетривиальной задачей для динамического анализа — необходимо определить, что нужно вызвать сначала `makePost`, а затем `sender`. При выполнении `sender` нужно добиться, чтобы прошла проверка в строке 15, зависящая от аргумента `makePost`. Для статического анализа проблема прохождения проверки гораздо менее существенна, описываемый алгоритм анализа её просто проигнорирует.

Теперь рассмотрим, как будет обработан код на листинге 3.

```
1  var l = "ipsum";
2  function i(x, y, z) {
3      $.post("/action/endpoint.php?action=" + x + "&id=" + y, {
4          lorem: l,
5          dolor: "sit amet",
6          sit: z
7      });
8  }
9  var j = function(tok1, tok2) {
10     var xj = "abc",
```

```
11          zj = "[" + tok2 + "]";
12      i(xj, tok1, zj);
13  }
14  function k() {
15      j("123", "amet");
16  }
```

Листинг 3. Второй пример кода, иллюстрирующий работу алгоритма

Этот пример отличается от предыдущего тем, что он содержит вызовы функции, внутри которой находится AJAX-вызов. При этом также есть зависимость параметров запроса от аргументов этой функции. В результате, анализируя места её вызова и то, какие аргументы там передаются, можно более точно определить вид отправляемого на сервер HTTP-запроса.

Пример характерен для кода реальных приложений, при этом он уже представляет сложную задачу для статического анализа, так как для передачи данных через вызовы функций требуется межпроцедурный анализ.

В ходе анализа на этапе поиска вызовов AJAX-функций будет выявлено, что аргументы вызова `$.post` зависят от формальных аргументов функции `i` (соответствующая проверка делается в строках 7 и 9 алгоритма 3). Поэтому далее будет произведён поиск вызовов функции `i` (см. функцию BuildCallChains в алгоритме 3). Это именованная функция, объявленная в глобальной области видимости, в строке 12 она вызывается по своему имени `i` — это место вызова будет обнаружено. В результате будет проделан ещё один обход AST-дерева для поиска вызовов AJAX-функций, который будет начат с корня AST-дерева, соответствующего телу функции, находящейся в строках 9–13 (ему соответствует вызов ExtractDEPs в строке 7 алгоритма 1). При обработке вызова `i` обход перейдёт на дерево, соответствующее коду `i` (в псевдокоде это происходит в строке 15 алгоритма 3), в результате чего анализ её тела будет повторен, но на этот раз с уточнёнными значениями формальных аргументов. В результате будет использовано значение формального аргумента `x` функции `i` (это строка «abc»). Однако значения `y` и `z` будут оставаться неизвестными — точнее, значение `y` будет частично определённым («`UNKNOWN`»), значение `z` — полностью неопределённым. Они также будут зависеть от формальных аргументов функции — на этот раз от аргументов функции в строках 9–13. Хотя эта функция анонимная, она присваивается переменной `j`, что будет обнаружено на этапе поиска значений переменных, в результате её вызов по имени `j` можно будет найти в строке 15. Обход AST-дерева для поиска вызовов AJAX-функций будет проделан в третий раз, на этот раз начиная от тела функции `k`. В ней в вызов `j` передаются конкретные аргументы, что в конечном итоге позволит полностью вычислить аргументы функции `$.post`. Для первого аргумента это будет значение `"/action/endpoint.php?action=abc&id=123"`, для второго — значение `{"lorem": "ipsum"dolor": "sit amet"sit": "[amet]"}`.

4. Экспериментальное сравнение метода с другими

Для оценки эффективности предложенного метода мы провели сравнительное экспериментальное исследование его прототипной реализации и механизмов поиска серверных точек взаимодействия, которыми располагают существующие сканеры безопасности веб-приложений. В эксперименте участвовали следующие сканеры безопасности:

— PT BBS — Positive Technologies BlackBox Scanner [21];
— Acunetix [22];
— Detectify [23];

— Burp Scanner [24];
— HCL AppScan Cloud [25].

Для проведения эксперимента было подготовлено модельное веб-приложение.

4.1. Описание модельного веб-приложения

Будем говорить, что *веб-приложение содержит точку ввода данных* (*DEP*) или *в веб-приложении есть точка ввода данных* (*DEP*), если это приложение реализует некоторую функцию на стороне сервера, которая принимает все HTTP-запросы, соответствующие DEP.

Используемое в эксперименте модельное веб-приложение является синтетическим и содержит на серверной стороне несколько DEP, обращения к которым есть на клиентской стороне (т. е. на веб-страницах приложения).

С точки зрения серверного кода, все DEP устроены одинаково — они делают SQL-запрос в базу данных и выдают ответ на него, причём их реализация содержит тривиальную SQL-инъекцию: в SQL-запрос подставляется значение параметра HTTP-запроса от клиента, которое никак не фильтруется. Исключением являются два DEP, у которых серверный код, помимо описанного, содержит также проверку *контрольного* параметра запроса — при несовпадении его значения с фиксированной константой (*правильным* значением) обработка запроса прерывается с ошибкой, запрос не делается. При этом у каждого из этих DEP URL-адрес запроса и имена параметров (включая имя уязвимого параметра) являются случайными — URL содержит подстроки из не менее чем 12 случайных символов (из набора английских букв в верхнем и нижнем регистрах, а также цифр от 0 до 9), имена параметров также являются строками из 6 таких символов. Это делает подбор правильной комбинации URL и имён параметров невозможным на практике. Пример URL-адреса, соответствующего одному из DEP модельного веб-приложения: `http://test.stand/application/jie8Ye/interface/aesi9X/handle?Po3oom=1`. Пример JavaScript-кода со страницы модельного веб-приложения, отправляющего POST-запрос на сервер, приведён в листинге 4.

```
1 var api = "/application/iuT6ei/";
2
3 function request7() {
4     $.post(api + "interface/Eek0Mu/handle", {"eeNgi6": "1"});
5 }
```

Листинг 4. Код, отправляющий POST-запрос

Кроме URL-адреса и имён параметров, DEP различаются HTTP-методами и способом передачи параметров. Все обращения к DEP находятся либо на главной странице модельного веб-приложения, либо на страницах, на которые с главной страницы ведут прямые ссылки. Исходный код стенда доступен по ссылке `https://github.com/seclab-msu/js-dep-mining-test-app`.

То, какой вид имеют находящиеся на клиентской стороне обращения к DEP модельного веб-приложения, описано в следующем списке:

1) Запрос делается при загрузке страницы вызовом `$.ajax` в теге `<script>`. Все аргументы вызова фиксированы и заданы в месте вызова константными литералами. Вызов находится на глобальном уровне, не в функции или условном операторе, поэтому запрос отправится при загрузке страницы, при обработке тега `<script>`, в котором находится вызов.

2) То же, что 1, но запрос делается при нажатии на кнопку (обработчиком события `onclick` тега `button`).
3) То же, что 1, но запрос делается при нажатии на кнопку обработчиком, зарегистрованным с помощью `addEventListener`.
4) То же, что 1, но запрос делается из JavaScript-функции, которая нигде не вызывается (скрытая функция).
5) Запрос делается вызовом функции `$.post`, находящимся в теле другой функции, причём URL-адрес запроса (задаваемый первым аргументом) сформирован как конкатенация значений глобальной JavaScript-переменной и строкового литерала. Эта глобальная переменная объявлена и инициализирована в том же скрипте выше по коду, ей при инициализации присвоено константное значение (строковый литерал). Параметры запроса (определяемые вторым аргументом вызова) заданы полностью константным литералом.
6) Запрос делается вызовом функции `$.ajax`, находящимся в теле другой функции. В качестве аргументов вызова, опередеяющих URL-адрес и параметры запроса, переданы локальные переменные, которые объявлены и инициализированы в той же функции, при инициализации им присвоены константные значения (строковые литералы).
7) Запрос делается вызовом функции. В запросе передаются два параметра. Значение первого фиксировано прямо в месте вызова (задано строковым литералом), этот параметр подставляется на сервере в SQL-запрос (через него возможна эксплуатация уязвимости). Второй параметр является *контрольным*. Его значение задается глобальной переменной, которая объявляется в одном месте и присваивается в другом, принимая несколько возможных значений. Во всех этих присваиваниях значения, которые присваиваются, — константные (заданы строковыми литералами), одно из них является *правильным*.
8) Запрос делается вызовом функции, параметры которого зависят от переменных, имена которых дублируются в разных перекрывающихся областях видимости.
9) То же, что 7, но значение *контрольного* параметра и иных параметров запроса при этом передаётся через цепочку вложенных функциональных вызовов.
10) Запрос делается вызовом функции, URL-адрес запроса берётся с помощью операции доступа к полю объекта из объекта — глобальной переменной, которая задана на странице через объектный литерал.
11) То же, что 10, но глобальная переменная-объект создаётся не объектным литералом, а посредством `new Object`, после чего её свойства заполняются с помощью присваиваний полей.
12) Запрос делается вызовом функции `$.get`, вызов находится в теле функции. Объект, опередеяющий набор параметров запроса (передаваемый во втором аргументе вызова), создаётся с помощью `new Object`, после чего его поля заполняются с помощью присваиваний полей. Создание этого объекта и присваивание полей делается в той же функции, что и вызов `$.get`, непосредственно перед местом вызова.
13) Запрос делается JavaScript-кодом в теге `<script>`, которого изначально нет в HTML-разметке страницы, этот фрагмент разметки скачивается с сервера JavaScript-кодом и добавляется на страницу с помощью вызова jQuery `$(...).html(...)` во время загрузки страницы.
14) Для отправки запроса используется цепочка вызовов, одна функция передаёт аргументы в другую, а та делает отправляющий запрос вызов. Все объявления

функций завёрнуты в анонимную самовызывающуюся функцию. Значения параметров берутся из локальных переменных объемлющей функции. URL-адрес запроса приходит из глобальной переменной-конфига.

15) POST-запрос, отправляемый вызовом функции, для формирования параметров которого используется `FormData`, так что получается multipart-запрос.
16) Запрос делается вызовом функции, часть параметров которого (определяющих URL-адрес) получается парсингом `location`.
17) Запрос делается кодом, активно использующим современные возможности языка JavaScript: `let` и `const`, arrow-функции и конструкцию `try..catch` без переменной у `catch`.

4.2. Методика эксперимента

Для проведения эксперимента модельное веб-приложение было развернуто на сервере, после чего по очереди запускались все участвующие в сравнении сканеры. На стороне приложения записывались все поступающие от сканеров HTTP-запросы, включая заголовки и тела запросов, причём для каждого сканера собирался отдельный журнал.

Для проверки того, что DEP обнаружен сканером, использован следующий критерий:

— если в отчёте о сканировании была запись о наличии уязвимости с URL и именами параметров, соответствующими реально существующему DEP, то мы считали, что DEP обнаружен;
— иначе мы проверяли, есть ли в журнале запросов попытки эксплуатации с соответствующими DEP URL-адресом и набором параметров. Попыткой эксплуатации является любой запрос со значениями параметров, отличных от тех, которые заданы на странице статически;
— если ни одно из условий выше не выполнялось, считалось, что DEP не обнаружен.

Такой способ проверки выбран по той причине, что сканеры, как правило, не предоставляют в явном виде информацию об обнаруженных DEP, а также для некоторых из них не удалось получить список найденных уязвимостей (удалось лишь получить информацию об их количестве).

Для реализованного прототипа предложенного метода информация о найденных DEP может быть получена непосредственно, поэтому для него результаты получены иначе: произведён анализ каждой из страниц модельного веб-приложения, после чего взято объединение множеств описаний DEP, найденных на всех страницах, и для каждого из искомых DEP проверено, входит ли соответствующее ему описание в полученное множество.

4.3. Результаты эксперимента

В таблице приведены результаты экспериментального исследования — количество связанных с уязвимостями точек ввода данных, найденных сканерами безопасности и прототипной реализацией предложенного метода анализа, которая обозначена как *Прототип*.

Отметим, что только сканеры Acunetix и HCL AppScan Cloud смогли успешно обнаружить часть наиболее тривиальных динамически создаваемых DEP и корректно определить их параметры. Таким образом, все уязвимости, DEP и параметры которых определяются в рамках динамического исполнения JavaScript, не могут быть успешно обнаружены большинством существующих сканеров безопасности.

Прототипная реализация предложенного метода анализа успешно обнаружила все DEP, связанные с уязвимостями модельного приложения.

№ DEP	1	2	3	4–17
PT BBS				
Acunetix	✓	✓	✓	
Detectify				
Burp Scanner				
HCL AppScan	✓	✓	✓	
Прототип	✓	✓	✓	✓✓✓

Заключение

Поиск серверных точек ввода данных является важным этапом работы сканера безопасности веб-приложений в модели «чёрного ящика», так как его результаты определяют, какие функции приложения будут проанализированы сканером, а какие нет.

В данной работе предложен метод повышения полноты поиска серверных DEP за счёт статического анализа клиентского JavaScript-кода. Алгоритм выявляет в клиентском коде функции, которые порождают запросы на сервер, и использует статический анализ, работающий над AST-деревом клиентского кода для определения возможных значений параметров этих функций. Найденные функции в дальнейшем позволяют выполнять динамический анализ серверной части приложения, например с помощью фаззинг-тестирования.

Отметим, что задача поиска серверных точек ввода данных отличается от других задач анализа безопасности клиентской стороны веб-приложения. Например, для поиска DOM-based XSS клиентский JavaScript-код, который никогда не выполняется на странице (т. е. «мёртвый код»), не представляет интереса — даже если он содержит уязвимость, её невозможно эксплуатировать. В то же время для получения информации о сервере информация о запросах из мёртвого кода может быть полезна, но такой код можно проанализировать только статически.

Для проведения экспериментов мы реализовали тестовое приложение, уязвимое к SQL-инъекции в серверной части, и 17 различных DEP, большая часть из которых — скрытые функции, использующие различные способы передачи параметров и формирования HTTP-запросов, взятые из реально существующих веб-приложений. На этом приложении протестировано несколько популярных сканеров безопасности — Acunetix, Positive Technologies Black Box Scanner, Detectify, Burp Scanner, HCL AppScan. Эксперименты показали, что предложенный алгоритм существенно превосходит возможности популярных сканеров в обнаружении скрытых функций веб-приложений.

ЛИТЕРАТУРА

1. *Huang Y. W., Huang S. K., Lin T. P., and Tsai C. H.* Web application security assessment by fault injection and behavior monitoring // Proc. WWW2003. Budapest, Hungary, May 21–25, 2003. P. 148–159.
2. *Раздобаров А. В., Петухов А. А., Гамаюнов Д. Ю.* Проблемы обнаружения уязвимостей в современных веб-приложениях // Проблемы информационной безопасности. Компьютерные системы. 2015. № 4. С. 64–69.
3. `https://w3techs.com/technologies/overview/client_side_language` — Статистика использования языков программирования на клиентской стороне веб-приложений по данным сайта w3techs.com.
4. *Richards G., Lebresne S., Burg B., and Vitek J.* An analysis of the dynamic behavior of JavaScript programs // ACM SIGPLAN Notices. 2010. V. 45. No. 6. P. 1–12.
5. `https://www.alexa.com/topsites` — Alexa Top 500 Global Sites.

6. `https://jquery.com` — Библиотека jQuery.
7. `https://www.google.com/recaptcha/about/` — Система защиты веб-сайтов от интернет-ботов reCAPTCHA.
8. *Kwangwon S. and Sukyoung R.* Analysis of JavaScript programs: Challenges and research trends // ACM Comput. Surveys. 2017. V. 50. No. 4. Article 59.
9. *Andreasen E., Gong L., Møller A., et al.* A survey of dynamic analysis and test generation for JavaScript // ACM Comput. Surveys. 2017. V. 50. No. 5. Article 66.
10. *Ryu S., Park J., and Park J.* Toward analysis and bug finding in JavaScript web applications in the wild // IEEE Software. 2018. V. 36. No. 3. P. 74–82.
11. *Andreasen E. and Møller A.* Determinacy in static analysis of jQuery // ACM SIGPLAN Notices. 2014. V. 49. No. 10. P. 17–31.
12. *Doupé A., Cova M., and Vigna G.* Why Johnny can't pentest: An analysis of black-box web vulnerability scanners // Proc. DIMVA 2010. Berlin; Heidelberg: Springer, 2010. P. 111–131.
13. *Doué A., Cavedon L., Kruegel C., and Vigna G.* Enemy of the state: A state-aware black-box web vulnerability scanner // 21st USENIX Security Symp. 2012. P. 523–538.
14. *Doupé A.* Advanced Automated Web Application Vulnerability Analysis. Diss. UC Santa Barbara, 2014.
15. *Choudhary S., Dincturk M., Mirtaheri S., et al.* Crawling rich internet applications: the state of the art // Proc. Conf. of the Center for Advanced Studies on Collaborative Research, 2012. P. 146–160.
16. *Mesbah A., Deursen A., and Lenselink S.* Crawling Ajax-based web applications through dynamic analysis of user interface state changes // ACM Trans. Web. 2012. V. 6. P. 3:1–3:30.
17. *Antal G., Hegedüs P., Toth Z., et al.* Static javascript call graphs: A comparative study // IEEE 18th Intern. Conf. SCAM. 2018. P. 177–186.
18. *Ko Y., Lee H., Dolby J., and Ryu S.* Practically tunable static analysis framework for large-scale JavaScript applications (T) // 30th IEEE/ACM Intern. Conf. ASE. 2015. P. 541–551.
19. *Wittern E., Ying A. T. T., Zheng Y., et al.* Statically checking web API requests in JavaScript // Proc. 39th Intern. Conf. Software Eng. 2017. P. 244–254.
20. `https://babeljs.io/` — Описание и документация библиотеки Babel.
21. `https://www.slideshare.net/pdug_slides/pt-blackbox-scanner` — Презентация с описанием возможностей средства PT BBS.
22. `https://www.acunetix.com` — Официальный сайт инструментального средства Acunetix.
23. `https://detectify.com` — Официальный сайт средства Detectify.
24. `https://portswigger.net/burp/burp-scanner` — Страница инструментального средства Burp Scanner на официальном сайте платформы Burp Suite.
25. `https://www.hcltechsw.com/wps/portal/products/appscan/home` — Официальный сайт сканера HCL AppScan Cloud.

REFERENCES

1. *Huang Y. W., Huang S. K., Lin T. P., and Tsai C. H.* Web application security assessment by fault injection and behavior monitoring. Proc. WWW2003, Budapest, Hungary, May 21–25, 2003, pp. 148–159.
2. *Razdobarov A. V., Petukhov A. A., and Gamayunov D. Yu.* Problemy obnaruzheniya uyazvimostey v sovremennykh veb-prilozheniyakh [Problems overview for modern web applications vulnerabilities discovery]. Problemy Informatsionnoy Bezopasnosti. Komp'yuternye Sistemy, 2015, no. 4, pp. 64–69. (in Russian)
3. `https://w3techs.com/technologies/overview/client_side_language` — Usage statistics of client-side programming languages for websites.

4. *Richards G., Lebresne S., Burg B., and Vitek J.* An analysis of the dynamic behavior of JavaScript programs. ACM SIGPLAN Notices, 2010, vol. 45, no. 6, pp. 1–12.
5. `https://www.alexa.com/topsites` — Alexa Top 500 Global Sites.
6. `https://jquery.com` — jQuery.
7. `https://www.google.com/recaptcha/about/` — reCAPTCHA.
8. *Kwangwon S. and Sukyoung R.* Analysis of JavaScript programs: Challenges and research trends. ACM Comput. Surveys, 2017, vol. 50, no. 4, Article 59.
9. *Andreasen E., Gong L., Møller A., et al.* A survey of dynamic analysis and test generation for JavaScript. ACM Comput. Surveys, 2017, vol. 50, no. 5, Article 66.
10. *Ryu S., Park J., and Park J.* Toward analysis and bug finding in JavaScript web applications in the wild. IEEE Software, 2018, vol. 36, no. 3, pp. 74–82.
11. *Andreasen E. and Møller A.* Determinacy in static analysis of jQuery. ACM SIGPLAN Notices, 2014, vol. 49, no. 10, pp. 17–31.
12. *Doupé A., Cova M., and Vigna G.* Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. Proc. DIMVA 2010, Berlin; Heidelberg, Springer, 2010, pp. 111–131.
13. *Doué A., Cavedon L., Kruegel C., and Vigna G.* Enemy of the state: A state-aware black-box web vulnerability scanner. 21st USENIX Security Symp., 2012, pp. 523–538.
14. *Doupé A.* Advanced Automated Web Application Vulnerability Analysis. Diss. UC Santa Barbara, 2014.
15. *Choudhary S., Dincturk M., Mirtaheri S., et al.* Crawling rich internet applications: the state of the art. Proc. Conf. of the Center for Advanced Studies on Collaborative Research, 2012, pp. 146–160.
16. *Mesbah A., Deursen A., and Lenselink S.* Crawling Ajax-based web applications through dynamic analysis of user interface state changes. ACM Trans. Web, 2012, vol. 6, pp. 3:1–3:30.
17. *Antal G., Hegedüs P., Toth Z., et al.* Static javascript call graphs: A comparative study. IEEE 18th Intern. Conf. SCAM, 2018, pp. 177–186.
18. *Ko Y., Lee H., Dolby J., and Ryu S.* Practically tunable static analysis framework for large-scale JavaScript applications (T). 30th IEEE/ACM Intern. Conf. ASE, 2015, pp. 541–551.
19. *Wittern E., Ying A. T. T., Zheng Y., et al.* Statically checking web API requests in JavaScript. Proc. 39th Intern. Conf. Software Eng., 2017, pp. 244–254.
20. `https://babeljs.io/` — Babel.
21. `https://www.slideshare.net/pdug_slides/pt-blackbox-scanner` — PT BBS Slides.
22. `https://www.acunetix.com` — Acunetix.
23. `https://detectify.com` — Detectify.
24. `https://portswigger.net/burp/burp-scanner` — Burp Scanner.
25. `https://www.hcltechsw.com/wps/portal/products/appscan/home` — HCL AppScan Cloud.