МАТЕМАТИЧЕСКИЕ ОСНОВЫ КОМПЬЮТЕРНОЙ БЕЗОПАСНОСТИ

УДК 004.056.53

ИСПОЛЬЗОВАНИЕ ОТЛАДОЧНОГО API СОВРЕМЕННОГО ВЕБ-ОБОЗРЕВАТЕЛЯ ДЛЯ ОБНАРУЖЕНИЯ УЯЗВИМОСТЕЙ КЛАССА DOM-BASED XSS

Д. А. Сигалов, А. В. Раздобаров, А. А. Петухов

Московский государственный университет имени М. В. Ломоносова, г. Москва, Россия

Рассматривается решение задачи поиска уязвимостей класса DOM-based XSS через последовательную комбинацию методов динамического анализа и fuzz-тестирования. Для создания поддерживаемого динамического анализатора JavaScript-кода используется современный веб-обозреватель Firefox без модификации его исходного кода. Приводится обзор существующих методов поиска уязвимостей класса DOM-based XSS.

Ключевые слова: *DOM-based XSS, уязвимости веб-приложений, динамический анализ, fuzz-тестирование.*

DOI 10.17223/20710410/35/6

DETECTING DOM-BASED XSS VULNERABILITIES USING DEBUG API OF THE MODERN WEB-BROWSER

D. A. Sigalov, A. V. Razdobarov, A. A. Petukhov

Lomonosov Moscow State University, Moscow, Russia

E-mail: asterite@seclab.cs.msu.su, korse@seclab.cs.msu.su, petand@seclab.cs.msu.su

In recent years, a significant part of web application functionality moves to client side. Increasing complexity of client-side code leads to a considerable growth in the number of client-side vulnerabilities and even to an emergence of new types of vulnerabilities, among which DOM-based XSS is most well-known. In this paper, we present an approach to detect and validate DOM-based XSS vulnerabilities. Our approach leverages dynamic tracking of data flows on the client side of web application to identify insecure ones (those which lead to vulnerability). An insecure data flow is a flow, in which a critical operation is data-dependent on attacker-controlled data, which is not sanitised properly. Data flows are tracked and classified using taint propagation technique (also known as "taint checking"). Potentially insecure data flows are tested for the presence of exploitable vulnerability by means of fuzzing — enumeration of possible attack vectors, which are passed to a data flow source. Vulnerability is confirmed if an execution of any injected payload is observed. Our approach was implemented on top of Firefox browser, controlled via it's debugger API. The paper discusses and justifies advantages of such implementation. The paper also provides an analysis of related work in the subject field, and comparison with other approaches is made. The proposed approach and its implementation are maintainable and extensible, which is

crucial for analyzing applications in constantly evolving environment such as client side web technologies.

Keywords: *DOM-based XSS, web application vulnerabilities, dynamic analysis, fuzzing.*

Введение

В работе предлагается решение задачи автоматизированного поиска уязвимостей типа DOM-based XSS (далее — DOMXSS). Уязвимости DOMXSS являются частными случаями уязвимостей к атакам Cross Site Scripting (XSS) — уязвимостей веб-приложений, которые предоставляют возможность злоумышленникам передать и выполнить подготовленный ими JavaScript-код в веб-обозревателе легитимного пользователя веб-приложения от его имени. Специфика DOMXSS заключается в том, что причиной уязвимости является ошибка в коде клиентской стороны веб-приложения: в JavaScript-коде, который выполняется на веб-странице и реализует интерактивный интерфейс взаимодействия с пользователем.

Более формально, уязвимость к атаке XSS имеет место, когда веб-приложение выводит данные, полученные из недоверенного источника (например, от пользователя), в HTTP-ответ таким образом, что они либо формируют новый контекст выполнения JavaScript-кода на странице, визуализируемой веб-обозревателем, либо нарушают целостность существующего контекста выполнения JavaScript-кода. Далее такой вывод данных в HTTP-ответ будем называть «влиянием на контекст выполнения JavaScript-кода» или просто «влиянием на контекст выполнения».

В случае DOMXSS недоверенным источником данных является окружение JavaScript-кода, на которое может влиять потенциальный атакующий: URL веб-страницы, элементы веб-страницы, значения которых формируются на основе ввода пользователей веб-приложения, и т. д. Влияние на контекст выполнения JavaScript-кода может произойти из-за небезопасного использования разработчиками кода веб-страницы, методов и свойств интерфейса объектной модели веб-страницы (DOM API). К таким методам относятся вызовы, предназначенные для модификации содержимого веб-страницы в интерактивном режиме (например, метод `document.write`), или вызовы, предназначенные для выполнения JavaScript-кода на лету (например, метод `eval`). Полный перечень небезопасных методов и свойств DOM API приведён в документе DOM XSS Wiki [1]. Недоверенные данные, используемые в подобных API-вызовах, открывают возможность злоумышленнику выполнить в контексте веб-страницы жертвы атаки произвольный JavaScript-код.

Пример уязвимости рассматриваемого типа содержится в веб-странице, фрагмент JavaScript-кода которой приведен в листинге 1. В JavaScript-коде продемонстрирован небезопасный вывод URL-адреса веб-страницы в её заголовок.

```
1 <html><body>
2       <h1>Поделиться адресом страницы: </h1>
3       <script>document.querySelector('h1').innerHTML +=
    location.href; </script>
4 </body></html>
```

Листинг 1. Пример веб-страницы, содержащей уязвимость DOMXSS

Если злоумышленник вынудит жертву атаки перейти по ссылке с адресом `http://site.com/page.html#<img src=x onerror=alert('XSS');></img>`, то после выпол-

нения операции `document.querySelector('h1').innerHTML += location.href` структура документа станет следующей:

```
1 <html><body>
2       <h1>Поделиться адресом страницы: http://site.com/page
    .html#<img src=x onerror=alert('XSS');></img></h1>
3       <script>document.querySelector('h1').innerHTML +=
    location.href; </script>
4 </body></html>
```

Листинг 2. Пример веб-страницы, содержащей уязвимость DOMXSS, после её эксплуатации

В результате изменения структуры веб-страницы появился новый контекст выполнения JavaScript-кода, подконтрольный злоумышленнику, который в приведённом примере содержит вызов «`alert('XSS')`». Выполнение внедрённого JavaScript-кода в контексте веб-страницы приводит к появлению окна-оповещения с надписью «XSS», что продемонстрировано на снимке экрана (рис. 1), демонстрация проведена в веб-обозревателе Google Chrome.

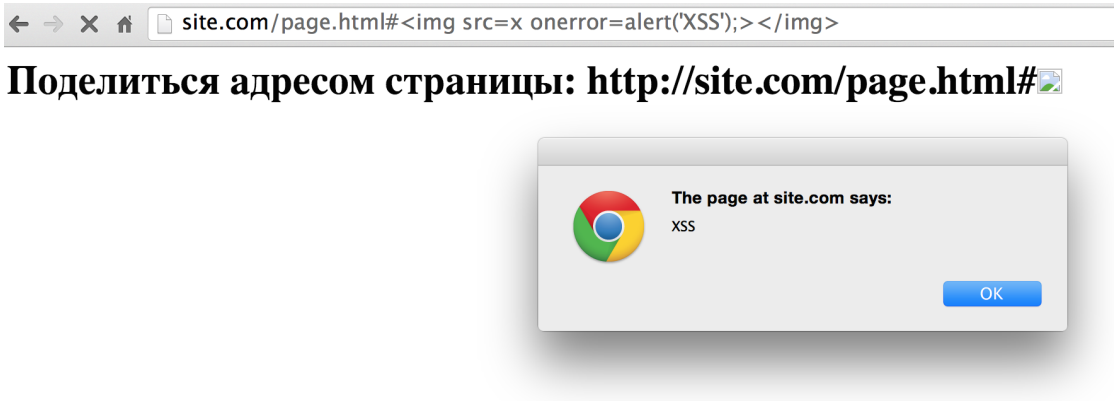

Рис. 1. Результат выполнения JavaScript-кода, внедрённого в веб-страницу с кодом из листинга 1

Добиться перехода по ссылке, что приведёт к эксплуатации уязвимости, можно, например, с помощью веб-страницы со следующим кодом (листинг 3); это проверено на нескольких версиях браузера Google Chrome, включая 55.0.2883.87 m.

```
1 <html><body>
2 Free movies: <a href="http://site.com/page.html#<img src=x
    onerror=alert('XSS');></img>">Click here for free
    downloads!</a>
3 </body></html>
```

Листинг 3. Пример веб-страницы, используемой для эксплуатации уязвимости

Для обнаружения уязвимостей типа DOMXSS необходимо проверить, используются ли данные из недоверенного источника в небезопасных методах DOM API (модификация содержимого веб-страницы или выполнение кода на лету) без предварительной обработки. Данный факт может быть установлен различными способами:

— с помощью fuzz-тестирования веб-страницы, её свойств и способов пользовательского взаимодействия с ней с последующим отслеживанием факта выполнения внед-

рённого JavaScript-кода (например, можно отслеживать вызов функции alert с заданным аргументом);
— с помощью статического или динамического анализа JavaScript-кода.

Каждый из приведённых способов обладает существенными ограничениями. Статический анализ JavaScript-кода является трудноразрешимой задачей в силу следующих обстоятельств:

— из-за динамической типизации языка;
— из-за возможности генерации и исполнения JavaScript-кода на лету (см. функцию `eval`);
— за счёт существенной зависимости логики выполнения JavaScript-кода от свойств DOM веб-страницы, значения которых определяются только на этапе работы пользователя с веб-приложением (данные свойства в общем случае зависят и от состояния веб-приложения на стороне сервера).

Реализация методов динамического анализа сопряжена с необходимостью тесной интеграции с интерпретатором языка JavaScript. Поскольку подавляющее большинство интерпретаторов не предоставляет вовне API для гранулярного контроля своей работы, часто единственным выходом для реализации динамического анализа остаётся модификация кода самого интерпретатора. Авторам не известна ни одна реализация динамического анализатора, который поддерживался бы его создателями параллельно с развитием основного интерпретатора. С учётом динамики появления новых версий стандарта ECMAScript [2] описанное ограничение является существенным.

Реализация fuzz-тестирования сопряжена с большим пространством перебора: учитывая, что данные из недоверенного источника могут небезопасно записываться в свойства DOM в результате самой сложной последовательности действий пользователя в веб-интерфейсе страницы, fuzz-тестирование без эффективной стратегии выбора некорректных входных данных приведёт к низкой полноте анализа. Сопутствующей технической сложностью является необходимость восстанавливать предыдущее состояние веб-страницы при последовательном переборе различных цепочек пользовательских действий.

В данной работе предлагается подход, который позволяет создать поддерживаемое, стабильно работающее средство, сочетающее преимущества динамического анализа JavaScript-кода и fuzz-тестирования, но лишённое их основных недостатков. Предлагаемый подход объединяет в себе следующие техники анализа JavaScript-кода:

— для построения зависимостей по данным во время выполнения JavaScript-кода используется метод распространения метки (taint-анализ) [3, 4];
— для обеспечения поддерживаемости реализации динамического taint-анализа используется Mozilla Debugger API веб-обозревателя Mozilla Firefox [5], позволяющий необходимым образом контролировать выполнение JavaScript-кода;
— для подтверждения небезопасности найденных потоков данных в части возможности проведения атаки класса DOMXSS используется метод направленного fuzz-тестирования, разработанный авторами. Направленное fuzz-тестирование позволяет существенно сократить пространство полного перебора за счёт использования информации о специфике обработки входных данных в тестируемом фрагменте JavaScript-кода, которая собирается на этапе динамического анализа.

1. Существующие методы и средства

Уязвимость типа XSS является примером уязвимости, описываемой моделью невмешательства [6]. Эта модель подразумевает наличие в программе или её фрагменте доверенного и недоверенного каналов ввода данных, а также доверенного и недоверенного каналов вывода данных. Под каналом вывода, в общем случае, понимаются вызовы подпрограмм (функций, процедур, методов) с аргументами или присваивания данных переменным или полям структур и объектов. Наличие уязвимости устанавливается как передача данных из недоверенного канала ввода в доверенный канал вывода. Модель невмешательства удобно описывает типы уязвимостей, которые появляются в результате использования программистами недоверенных данных, поступивших от пользователя, в критичных API-вызовах. Уязвимости типа DOMXSS являются частным случаем таких уязвимостей. Для применения модели невмешательства для поиска уязвимостей класса DOMXSS достаточно недоверенным каналам ввода поставить в соответствие методы и свойства DOM-модели, позволяющие JavaScript-коду получить данные, контролируемые пользователем, а доверенным каналам вывода — методы и свойства DOM-модели, позволяющие изменять структуру веб-страницы или выполнять JavaScript-код на лету.

1.1. Статический анализ

В существующих работах установление факта небезопасной передачи данных без выполнения анализируемого кода рассматривается либо как задача вычисления потоков данных от одних операторов (канал ввода) до других (канал вывода), либо как задача поиска небезопасной последовательности операторов над различными представлениями программы (дерево синтаксического разбора, граф потоков управления), либо как задача определения множества возможных значений переменной в заданной точке программы (в канале вывода).

Достоинством статического анализа является сбор информации обо всём имеющемся коде, вне зависимости от того, выполнялся он или нет. В контексте анализа кода на языке JavaScript, однако, применение статического анализа затруднительно. Язык предоставляет возможность динамически формировать или загружать с сервера код, а затем выполнять его. Данное обстоятельство порождает большую неопределённость при статическом анализе. Как следствие, существующие методы [7 – 10] накладывают существенные ограничения на анализируемый код — например, требуют отсутствия динамически вычисляемых имён свойств, неизменности прототипов объектов и самих объектов после первичной инициализации, отсутствия динамического формирования и выполнения кода и т. п. При вычислении множества возможных значений переменных в заданной точке JavaScript-программы сложность вызывают многочисленные обращения к свойствам DOM-элементов, значения которых становятся известны только во время выполнения. Точность аппроксимации значений, получаемых из страниц или Ajax-запросов, напрямую определяет качество итогового анализа.

Перечисленные характеристики программ на языке JavaScript позволяют сделать вывод о неприменимости статического анализа для поиска уязвимостей типа DOMXSS без расширения его методами динамического анализа.

1.2. Динамический анализ

Известны два подхода для динамического анализа JavaScript-кода.

Первый подход заключается в создании динамического анализатора на языке JavaScript. Анализ JavaScript-программы из другой JavaScript-программы можно осуществить благодаря различным методам интроспекции, которые поддерживаются

стандартом языка. При этом JavaScript-код анализатора встраивается в веб-страницу и контролирует выполнение анализируемого JavaScript-кода с помощью средств (в том числе методами интроспекции), которые предоставляет интерпретатор веб-обозревателя и DOM API. Основные методы, используемые при данном подходе, — это инструментирование анализируемого кода собственными вызовами, а также переопределение стандартных функций и объектов собственными. Последний метод позволяет осуществить вызовы к стандартному API через код собственных наблюдателей (т. н. функций-перехватчиков) и отслеживать передаваемые в них аргументы. Преимущество данного подхода заключается в том, что для его реализации не требуется модифицировать интерпретатор JavaScript-кода. Ограничения подхода связаны с уровнем рассмотрения процесса выполнения: наблюдателю на уровне JavaScript-кода недоступна информация о процессе выполнения программы, которая есть на уровне внутренних структур данных интерпретатора языка JavaScript. В результате описанный подход не позволяет реализовать полноценное отслеживание зависимостей по данным.

В то же время этот подход позволяет существенно повысить эффективность fuzz-тестирования веб-страницы при поиске уязвимостей типа DOMXSS. Динамический анализатор симулирует различные пользовательские действия на веб-странице, порождая тем самым DOM-события. DOM-события вызывают выполнение JavaScript-обработчиков с теми или иными значениями, полученными из свойств DOM-элементов. Инструментирование методов получения свойств DOM-элементов, а также методов вывода данных в DOM позволяет реализовать стратегии более эффективного fuzz-тестирования по сравнению с передачей во все входные каналы предустановленного списка тестирующих значений, таких, как `<script>alert(1);</script>`.

Второй подход заключается в создании динамического анализатора, который работает на уровне внутренних структур данных интерпретатора языка. В этом случае наблюдателю доступна любая информация о ходе выполнения кода, что позволяет корректно реализовать динамический taint-анализ и с его помощью устанавливать факт передачи данных из недоверенного канала ввода в доверенный канал вывода. Данный подход технически сложен в реализации, так как требуется среда, удовлетворяющая следующим требованиям:

— наличие интерпретатора JavaScript-кода актуальных версий стандарта ECMAScript;
— среда должна полноценно реализовывать DOM API в объёме текущих спецификаций, которые постоянно обновляются;
— среда должна предоставлять наблюдателю возможность получать информацию о процессе выполнения кода на уровне внутренних структур данных.

Попытки реализовать такую среду через модификацию кода существующих веб-обозревателей (WebKit, Chromium) или с нуля (HtmlUnit [11]) сталкиваются с трудностями при последующей поддержке полученного решения. Действительно, реализация в веб-обозревателях новых функций DOM API или нового стандарта ECMAScript приведёт к тому, что JavaScript-код, использующий новые возможности веб-обозревателей, перестанет выполняться в модифицированных средах для динамического анализа.

Стоит отдельно отметить, что сам метод динамического taint-анализа не лишён ограничений:

— наличие в коде ограничений на входные данные (множество допустимых значений, длина) может сделать код безопасным даже без явного использования фильтрующих функций при передаче данных из недоверенных каналов ввода в доверенные каналы вывода;

— корректность собственных валидирующих функций невозможно установить с помощью динамического taint-анализа (например, если валидация реализована с помощью регулярных выражений).

Приведённые рассуждения позволяют сделать следующие выводы:

— реализация динамического taint-анализа имеет смысл только в среде, поддержка которой осуществляется сообществом;
— отдельной задачей как при реализации динамического taint-анализа, так и при реализации fuzz-тестирования является максимизация покрытия анализируемого кода, что сопряжено с задачей симуляции на веб-странице возможных последовательностей пользовательских действий;
— динамический taint-анализ и fuzz-тестирование эффективно дополняют друг друга при решении задачи поиска уязвимостей типа DOMXSS.

1.3. Существующие средства

В результате исследования предметной области были проанализированы существующие инструментальные средства, решающие задачу поиска уязвимостей типа DOMXSS.

Средства Codemagi Burp DOM-XSS Scanner [12] и StaticBurp [13] используют поиск мест, содержащих уязвимость, по регулярным выражениям по узлам дерева разбора JavaScript-кода (AST). Предполагается, что результаты поиска проверяются оператором средства вручную.

DOMinator [14] использует динамический анализ потоков данных с помощью taint-анализа. Для сбора информации о выполнении используется изменённый веб-обозреватель Firefox под управлением оператора. В задачи оператора входит навигация по веб-приложению с целью максимизации покрытия анализируемого JavaScript-кода.

В средствах, описанных в [15 – 17], также реализован динамический taint-анализ, информация для которого собирается с помощью изменённого кода интерпретатора веб-обозревателя Google Chrome.

Авторы [18 – 20] применяют динамический taint-анализ, собирая информацию о выполнении с помощью преобразований JavaScript-кода.

Инструмент ra2-dom-xss-scanner [21] осуществляет простое fuzz-тестирование веб-страницы и отслеживание изменений свойств DOM.

Инструмент [22], а также авторы [23, 24] используют статический поиск зависимостей по данным между источниками ввода небезопасных данных и их использованием в небезопасных методах и свойствах DOM API.

В [7, 25] применяется комбинация статического и динамического анализа. Статическому анализатору на вход подаётся уже имеющийся на странице JavaScript-код. Если во время работы статического анализа обнаруживается вызов функции динамического выполнения кода, то её аргумент, содержащий JavaScript-код, подлежащий интерпретации на лету, также подаётся на вход статическому анализатору. Кроме того, в средстве [25], если во время статического анализа в коде страницы используются значения, получаемые из окружения кода (например, свойства DOM), они предоставляются средой выполнения в явном виде.

2. Предлагаемый метод

В данной работе предлагается производить динамический taint-анализ JavaScript-кода для поиска небезопасных потоков данных, а затем подтверждать наличие уязвимости в них через направленное fuzz-тестирование.

В качестве среды для анализа выполнения кода выбран веб-обозреватель Firefox. Использование данного веб-обозревателя позволяет максимально приблизить среду анализа к реальным условиям выполнения анализируемого кода и, как следствие, обеспечить корректную работу JavaScript-кода современных веб-приложений. Использование Mozilla Debugger API позволяет избежать модификации кода веб-обозревателя и, таким образом, обеспечить сопровождаемость реализованного метода и в последующих версиях веб-обозревателя.

После обнаружения потенциально небезопасных потоков данных для каждого из них производится попытка подтвердить наличие уязвимости через fuzz-тестирование значений, поступающих в потенциально небезопасный поток через канал ввода. Конкретный набор передаваемых в канал данных зависит от того, какие именно методы DOM API представляют канал ввода и канал вывода. Такой подход позволяет сократить пространство перебора, так как вместо всех возможных значений векторов атаки на вход передаются только те, которые могут привести к модификации или порождению нового контекста выполнения JavaScript-кода в данном конкретном случае.

Описываемый метод не включает решение задачи максимизации покрытия анализируемого JavaScript-кода. Он реализован в виде инструментального средства, которое представляет собой расширение (плагин) для веб-обозревателя Firefox. Предполагается, что оператор осуществляет навигацию по веб-приложению, используя штатные средства веб-обозревателя, а расширение в фоновом режиме вычисляет небезопасные потоки данных. При обнаружении таковых оператору отправляется сигнал, при получении которого он может инициировать этап fuzz-тестирования. В случае успеха последнего оператору выводится сообщение о найденной уязвимости и предоставляются проверочные данные, её подтверждающие.

Далее перечислены детали реализации динамического taint-анализа и fuzz-тестирования.

2.1. Реализация динамического анализа

Динамический taint-анализ является одним из способов определить зависимости по данным в программе в процессе её выполнения. В настоящей работе в процессе динамического taint-анализа участвуют данные только строкового типа (прототип string). В качестве источников, вывод которых устанавливает строкам флаг tainted, берутся все потенциально контролируемые злоумышленником методы и свойства DOM API, полный список которых может быть найден в энциклопедии DOM XSS [1]. Операциями, распространяющими флаг tainted, считаются любые функции и операторы преобразования строк (конкатенация, взятие подстроки, URL-декодирование и т. п.).

Для реализации отслеживания потоков данных используется отладочный интерфейс интерпретатора JavaScript веб-обозревателя Firefox (Mozilla Debugger API [5]). Для хранения информации о флагах заводится специальная таблица, хранящая ссылки на все строки, помеченные флагом tainted.

Более конкретно, taint-анализ реализован следующим образом:

1) Объекты, отвечающие за источники (например, объект `location`, предоставляющий интерфейс для работы с адресом страницы), заменяются средствами отладчика на специальные объекты-обёртки (proxy-объекты). Обёртки ведут себя так же, как и реальные объекты, однако при считывании из них данных (для `location` это произойдёт, например, при приведении к строке методом `toString` или при обращении к полю `location.href`) соответствующая операция запишет в таблицу строк, помеченных

флагом tainted, запрашиваемое значение прежде, чем возвратить его. Кроме самого значения, в таблицу помещается информация о его источнике (канале ввода).

2) Строковые операции, встроенные в язык, также подменяются на прокси-методы. Подменённые операции, кроме своего основного действия, проверяют, есть ли в таблице строки — аргументы операции. Если это так, то результат операции также помещается в таблицу ссылок на taint-строки. В метку источника у новой записи в таблицу заносятся источники всех помеченных аргументов операции.

3) Доверенные каналы вывода представляют функции и свойства выполнения кода на лету и изменения DOM: `eval`, `document.write`, `setTimeout`, `element.innerHTML`, `script.src`. Все такие функции и свойства заменяются на прокси-методы и свойства, которые перед своим основным действием проверяют, не содержится ли передаваемое им значение в таблице. Если это так, то сообщается об обнаружении потенциально небезопасного потока данных.

2.2. Реализация fuzz-тестирования

Цель этапа fuzz-тестирования для всех потенциально небезопасных потоков данных, найденных на этапе динамического taint-анализа, — подобрать такие входные данные, попадание которых в выходной канал может привести к созданию нового контекста выполнения JavaScript-кода или к модификации существующего. Оракулом такого события является вызов функции `alert` с уникальным случайным числом, которое генерируется для каждого экземпляра проверочных входных данных. В случае положительного ответа оракула средство выводит в журнал информацию о протестированном потоке данных и входные данные, подтверждающие небезопасность этого потока.

Так как потенциально небезопасный поток данных может быть обнаружен в результате сложной комбинации пользовательских действий, выполняемых оператором (например, раскрытие выпадающего меню, выбор в нём определённого пункта, переход в появившемся интерфейсе в определённую вкладку и раскрытие выпадающего меню в ней), актуальной задачей является реализация fuzz-тестирования кода таким образом, чтобы оператору для каждого экземпляра тестовых данных не приходилось повторять действия в веб-интерфейсе, активирующие соответствующий поток в JavaScript-коде страницы.

Более конкретно, повторное выполнение участка JavaScript-кода, который активируется пользовательскими действиями в веб-интерфейсе, может быть осуществлено следующим образом:

1) Перед выполнением кода, активированного DOM-событием, полностью сохраняется контекст выполнения JavaScript-кода. Для выполняемого кода оценивается его безопасность с помощью taint-анализа. В случае, если в результате применения taint-анализа поток помечается как потенциально небезопасный, реализуются следующие шаги с целью подтверждения данного факта.

2) В зависимости от типа API-функции, представляющей доверенный выходной канал, генерируются экземпляры входных данных, которые становятся кандидатами для проверки.

3) Для каждого сгенерированного экземпляра входных данных происходит восстановление контекста выполнения JavaScript-кода и управление передаётся на оператор, соответствующий началу тестируемого потока данных.

4) При достижении в процессе выполнения кода оператора, который соответствует получению пользовательских данных, в качестве результата его вызова возвращается очередной экземпляр тестовых входных данных.

5) Выполнение продолжается до тех пор, пока не выполнится последний оператор, представленный в тестируемом потоке данных, после чего осуществляется переход на шаг 3, перед чем осуществляется восстановление контекста выполнения (см. шаг 1).

Выполнение функции `alert` при достижении канала вывода очередных тестовых данных может происходить как синхронно, так и асинхронно. В первом случае процесс fuzz-тестирования для текущего потока завершается и информация о подтверждённой уязвимости записывается в журнал.

С учётом того, что выполнение функции `alert` может произойти асинхронно, возможно даже после выполнения оператором в веб-интерфейсе приложения каких-то других действий, факт вызова функции `alert` должен отслеживаться на всём протяжении работы средства. При этом уникальный идентификатор, являющийся аргументом функции `alert`, однозначно определит последовательность действий пользователя в интерфейсе приложения и уязвимые фрагменты кода, приводящие к порождению асинхронного контекста выполнения JavaScript-кода с функцией `alert`.

Описанный метод может быть реализован при достаточно большой степени контроля над выполнением кода. Такой контроль предоставляется программным интерфейсом отладчика Firefox, который и используется для реализации анализа процесса выполнения кода в данной работе.

Заключение

Предложен метод поиска уязвимостей типа DOMXSS с помощью динамического taint-анализа и последующего направленного fuzz-тестирования. Метод реализован на основе веб-обозревателя Firefox без модификации его исходного кода. Динамический taint-анализ позволяет обнаруживать потенциально небезопасные места в коде, в том числе динамически сформированном или загруженном на страницу в результате асинхронных запросов. Использование направленного fuzz-тестирования позволяет избавиться от ложных срабатываний, возникающих из-за ограничений динамического taint-анализа, а также получать на выходе проверочные данные, подтверждающие наличие уязвимостей. Использование в качестве среды для динамического анализа немодифицированного веб-обозревателя Firefox позволяет выполнять анализ кода в реальном окружении и избавляет от необходимости осуществлять постоянную поддержку средства при выходе обновлений веб-обозревателя.

ЛИТЕРАТУРА

1. *Stefano D. P., Heiderich M., and Braun F.* `https://code.google.com/p/domxsswiki/` — DOM XSS Test Cases Wiki Cheatsheet Project. 2010.
2. `http://www.ecma-international.org/publications/standards/Ecma-262.htm` — Стандарты языка программирования ECMAScript.
3. *Schwartz E. J., Avgerinos T., and Brumley D.* All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask) // IEEE Symp. Security and Privacy. Auckland: IEEE, 2010. P. 317–331.
4. *Wagner D.* Static analysis and software assurance. SAS'01. Proc. 8th Intern. Symp. Static Analysis. Paris: Springer Verlag, 2001. 431 p.
5. `https://developer.mozilla.org/en-US/docs/Tools/Debugger-API` — Документация отладочного интерфейса Mozilla Debugger API.

6. *McLean J.* Security Models // Encyclopedia of Software Engineering. N. Y.: John Wiley & Sons Inc., 1994. P. 1136–1145.
7. *Chugh R. et al.* Staged information flow for JavaScript // ACM Sigplan Notices. 2009. V. 44. No. 6. P. 50–52.
8. *Guha A., Krishnamurthi S., and Jim T.* Using static analysis for Ajax intrusion detection // Proc. 18th Intern. Conf. on World Wide Web. Madrid: ACM, 2009. P. 561–570.
9. *Madsen M., Livshits B., and Fanning M.* Practical static analysis of JavaScript applications in the presence of frameworks and libraries // Proc. 9th Joint Meeting on Foundations of Software Engineering. St. Petersburg: ACM, 2013. P. 499–509.
10. *Feldthaus A. et al.* Efficient construction of approximate call graphs for JavaScript IDE services // 35th Intern. Conf. Software Engineering (ICSE). San Francisco: IEEE, 2013. P. 752–761.
11. `http://htmlunit.sourceforge.net/` — Документация эмулятора управляемого веб-обозревателя HtmlUnit. 2016.
12. `https://www.codemagi.com/downloads/dom-xss-scanner-checks` — Описание Codemagi Burp DOM-XSS Scanner — расширения для инструмента BurpSuite, предназначенного для поиска DOMXSS.
13. `http://www.ethicalhack3r.co.uk/staticburp-burp-suite-potential-dom-xss-analysis/` — Описание StaticBurp — расширения для инструмента BurpSuite, предназначенного для поиска DOMXSS.
14. `https://dominator.mindedsecurity.com/` — Описание средства поиска DOMXSS DOMinator
15. *Just S. et al.* Information flow analysis for JavaScript // Proc. 1st ACM SIGPLAN Intern. Workshop on Programming Language and Systems Technologies. Portland: ACM, 2011. P. 9–18.
16. *Lekies S., Stock B., and Johns M.* 25 million flows later: large-scale detection of DOM-based XSS // Proc. ACM SIGSAC Conf. Computer & Communications Security. Berlin: ACM, 2013. P. 1193–1204.
17. *Lekies S., Stock B., and Johns M.* Practical blended taint analysis for JavaScript // Proc. Intern. Symp. Software Testing and Analysis. Lugano: ACM, 2013. P. 336–346.
18. *Xiao W. et al.* Preventing client side XSS with rewrite based dynamic information flow // Sixth Intern. Symp. Parallel Architectures, Algorithms and Programming (PAAP). Pekin: IEEE, 2014. P. 238–243.
19. *Jang D. et al.* Rewriting-based dynamic information flow for JavaScript // 17th ACM Conf. Computer and Communications Security. Chicago: ACM, 2010. `http://goto.ucsd.edu/~rjhala/papers/rewriting_based_dynamic_information_flow_for_javascript.pdf`
20. *Prabawa A. and Chin W. N.* Titania: Generic Dynamic Information Flow Analysis Framework for JavaScript. `http://www.comp.nus.edu.sg/~adi-yoga/Titania/Titania.pdf` (дата обращения: 01.09.2016).
21. `https://code.google.com/p/ra2-dom-xss-scanner/` — Документация средства поиска DOMXSS Ra.2.
22. `www.jsprime.org` — Описание средства поиска DOMXSS JSPrime.
23. *Guarnieri S. et al.* Saving the world wide web from vulnerable JavaScript // Proc. Intern. Symp. Software Testing and Analysis. Toronto: ACM, 2011. P. 177–187.
24. *Saxena P. et al.* A symbolic execution framework for JavaScript // IEEE Symp. Security and Privacy. Auckland: IEEE, 2010. P. 513–528.

25. *Tripp O., Ferrara P., and Pistoia M.* Hybrid security analysis of Web JavaScript code via dynamic partial evaluation // Proc. Intern. Symp. Software Testing and Analysis. San Jose: ACM, 2014. P. 49–59.

REFERENCES

1. *Stefano D. P., Heiderich M., and Braun F.* https://code.google.com/p/domxsswiki/ — DOM XSS Test Cases Wiki Cheatsheet Project, 2010.
2. http://www.ecma-international.org/publications/standards/Ecma-262.htm
3. *Schwartz E. J., Avgerinos T., and Brumley D.* All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). IEEE Symp. Security and Privacy, Auckland, IEEE, 2010, pp. 317–331.
4. *Wagner D.* Static analysis and software assurance. SAS'01. Proc. 8th Intern. Symp. Static Analysis, Paris, Springer Verlag, 2001. 431 p.
5. https://developer.mozilla.org/en-US/docs/Tools/Debugger-API
6. *McLean J.* Security Models. Encyclopedia of Software Engineering, N. Y., John Wiley & Sons Inc., 1994, pp. 1136–1145.
7. *Chugh R. et al.* Staged information flow for JavaScript. ACM Sigplan Notices, 2009, vol. 44, no. 6, pp. 50–52.
8. *Guha A., Krishnamurthi S., and Jim T.* Using static analysis for Ajax intrusion detection. Proc. 18th Intern. Conf. on World Wide Web, Madrid, ACM, 2009, pp. 561–570.
9. *Madsen M., Livshits B., and Fanning M.* Practical static analysis of JavaScript applications in the presence of frameworks and libraries. Proc. 9th Joint Meeting on Foundations of Software Engineering, St. Petersburg, ACM, 2013, pp. 499–509.
10. *Feldthaus A. et al.* Efficient construction of approximate call graphs for JavaScript IDE services. 35th Intern. Conf. Software Engineering (ICSE), San Francisco, IEEE, 2013, pp. 752–761.
11. http://htmlunit.sourceforge.net/
12. https://www.codemagi.com/downloads/dom-xss-scanner-checks
13. http://www.ethicalhack3r.co.uk/staticburp-burp-suite-potential-dom-xss-analysis/
14. https://dominator.mindedsecurity.com/ —
15. *Just S. et al.* Information flow analysis for JavaScript. Proc. 1st ACM SIGPLAN Intern. Workshop on Programming Language and Systems Technologies, Portland, ACM, 2011, pp. 9–18.
16. *Lekies S., Stock B., and Johns M.* 25 million flows later: large-scale detection of DOM-based XSS. Proc. ACM SIGSAC Conf. Computer & Communications Security, Berlin, ACM, 2013, pp. 1193–1204.
17. *Lekies S., Stock B., and Johns M.* Practical blended taint analysis for JavaScript. Proc. Intern. Symp. Software Testing and Analysis, Lugano, ACM, 2013, pp. 336–346.
18. *Xiao W. et al.* Preventing client side XSS with rewrite based dynamic information flow. Sixth Intern. Symp. Parallel Architectures, Algorithms and Programming (PAAP), Pekin, IEEE, 2014, pp. 238–243.
19. *Jang D. et al.* Rewriting-based dynamic information flow for JavaScript. 17th ACM Conf. Computer and Communications Security, Chicago, ACM, 2010. http://goto.ucsd.edu/~rjhala/papers/rewriting_based_dynamic_information_flow_for_javascript.pdf
20. *Prabawa A. and Chin W. N.* Titania: Generic Dynamic Information Flow Analysis Framework for JavaScript. http://www.comp.nus.edu.sg/~adi-yoga/Titania/Titania.pdf (access date 01.09.2016).
21. https://code.google.com/p/ra2-dom-xss-scanner/

22. `www.jsprime.org`
23. *Guarnieri S. et al.* Saving the world wide web from vulnerable JavaScript. Proc. Intern. Symp. Software Testing and Analysis, Toronto, ACM, 2011, pp. 177–187.
24. *Saxena P. et al.* A symbolic execution framework for JavaScript. IEEE Symp. Security and Privacy, Auckland, IEEE, 2010, pp. 513–528.
25. *Tripp O., Ferrara P., and Pistoia M.* Hybrid security analysis of Web JavaScript code via dynamic partial evaluation. Proc. Intern. Symp. Software Testing and Analysis, San Jose, ACM, 2014, pp. 49–59.