ОГЛАВЛЕНИЕ

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

DOM – document object model, объектная модель документа.

OWASP – Open Web Application Security Project.

XSS – cross-site scripting, межсайтовый скриптинг.

АСД – абстрактное синтаксическое дерево.

СЛОВАРЬ ТЕРМИНОВ

code-review: неавтоматизированный просмотр исходного кода программы разработчиками на предмет ошибок.

DOM-based XSS: уязвимость межсайтового скриптинга, основанного на объектной модели документа.

XSS-вектор: специально сформированные входные данные, позволяющие определить наличие уязвимости XSS.

sink: метод языка программирования, который позволяет реализовать атаку в случае передачи ему в качестве аргумента непроверенных входных пользовательских данных.

source: «источник», метод языка программирования, который позволяет получить входные пользовательские данные.

инструментация: процесс преобразования программы, во время которого в программу добавляются специальные функциональные вставки, позволяющие отслеживать ход ее выполнения.

маршрут: последовательность имен функций (методов), получающих XSS-вектор в качестве аргумента, которая получена при помощи разработанного средства.

ВВЕДЕНИЕ

На настоящий момент JavaScript является одним из наиболее используемых языков программирования в области разработки веб-приложений. Интернет-издание Cleverism в январе 2015 года составило рейтинг[20] наиболее популярных языков программирования, используемых в веб-разработке, и поставило JavaScript на первое место. На каком бы языке не была написана серверная часть приложения, за клиентскую чаще всего отвечает именно JavaScript. Этот язык позволяет достигнуть нескольких целей, включая управление браузером, редактирование содержимого документа, выведенного на экран, взаимодействие между клиентскими сценариями и пользователем, а также асинхронную передачу данных. Язык был разработан компанией Netscape и заимствует большую часть своего синтаксиса у языка C.

Одним из главных плюсов языка JavaScript для разработчиков является то, что это - один из нескольких языков программирования, которые поддерживаются всеми наиболее используемыми браузерами без потребности в каких-либо компиляторах или плагинах. Кроме того, он может использоваться не только для работы с веб-платформами, но и, например, с графическими модулями для рабочего стола и PDF-документами. JavaScript поддерживает стили функционального и объектно-ориентированного программирования и является динамическим.

С появлением в 2009 году платформы Node.js, позволяющей использовать JavaScript для разработки и серверной стороны приложения, этот язык превратился из узкоспециализированного в язык общего назначения.

Под уязвимостью программы понимается такой недостаток в программе, который способен привести к реализации угрозы информационной

безопасности. Одна из этих угроз – угроза нарушения целостности. Под целостностью информации понимается такое состояние информации, при котором её изменение осуществляется только преднамеренно субъектами, имеющими на него право[4].

Одной из самых распространенных уязвимостей в программах на языке JavaScript является межсайтовый скриптинг (Cross-site scripting, XSS). Данная уязвимость находится на третьем месте в рейтинге уязвимостей от OWASP (Open Web Application Security Project) за 2015 год[21].

В последнее время фокус злоумышленников смещается с серверной части приложения на клиентскую[13]. Здесь им помогает достичь своих целей сравнительно новый тип XSS (по отношению к отраженным и хранимым) – межсайтовый скриптинг, основанный на объектной модели документа. В отличие от первых двух типов, DOM-based XSS не требует взаимодействия с сервером: атака может быть реализована посредством изменения окружения DOM в браузере жертвы. Сама по себе страница (то есть ответ на HTTP-запрос) не изменяется, однако скрипт, содержащийся на клиентской стороне выполняется по-другому из-за изменения DOM.

При выполнении JavaScript-программы в браузере, код программы получает доступ к некоторым объектам DOM, главным из которых является объект document. Оказывая влияние на свойства этого объекта, злоумышленник может реализовать атаку. В результате атаки, вредоносный код, внедренный злоумышленником может получить доступ к файлам cookies, токенам сессии и другой критичной информации, хранимой браузером и используемой для работы с сайтом. Таким образом, атакующий сможет получить несанкционированный доступ к профилю пользователя и совершать различные действия от его лица.

Поскольку при эксплуатации уязвимости DOM-based XSS злоумышленник так изменяет клиентский сценарий ,чтобы он выполнился по-

другому, то данная уязвимость является уязвимостью нарушения целостности. Эта уязвимость особенно опасна, так как в случае ее эксплуатации нарушение целостности данных, может привести к нарушению конфиденциальности. Распространенность уязвимости делает задачу ее поиска в веб-приложениях крайне актуальной.

Знания разработчиков в области безопасности зачастую ограничены и, чтобы обезопасить свои веб-приложения, они либо пользуются услугами аудиторов, проводящих тестирование на проникновение (крупные компании по разработке программного обеспечения), либо самостоятельно проводят анализ защищенности с помощью автоматизированных средств.

Целью данной работы является разработка средства обнаружения уязвимостей нарушения целостности, доступного любому разработчику и позволяющего сократить время, затрачиваемое на локализацию уязвимости. Актуальность доступности подобного средства обусловлена большим количеством стартапов и фрилансеров в области разработки веб-приложений. Такие разработчики зачастую не могут себе позволить дорогостоящие автоматизированные средства анализа или аудит сторонним специалистами.

Ввиду большой распространенности в веб-приложениях и высокого риска эксплуатации для поиска была выбрана уязвимость межсайтового скриптинга, основанного на DOM.

Для достижения поставленной цели были поставлены три задачи. Первая – обзор существующих методов поиска уязвимостей: выявление их преимуществ и недостатков для определения нового подхода к решению следующей задачи, имеющего некоторые преимущества над существующими. Этой задачей являлась непосредственная разработка средства с использованием знаний, полученных на первом этапе. Последней задачей было подтверждение корректности функционирования разработанного

средства и достоверности полученных результатов, что подтверждается сравнением выходных данных с уже подтвержденными.

Новизна данной работы заключается в решении второй задачи, так как разработанное средство сочетает в себе преимущества различных методов, позволяя получить более точные результаты, по сравнению с другими средствами, на любом этапе разработки проекта.

В главе 1 рассмотрены основные положения в исследуемой области.

В главе 2 производится обзор существующих методов поиска уязвимости межсайтового скриптинга, основанного на объектной модели документа. Рассматриваются преимущества и недостатки этих методов, определяются функции, которыми должно обладать разрабатываемое средство.

Глава 3 посвящена деталям практической реализации средства.

В четвертой главе проводится испытание разработанного средства на реальных данных. Полученные результаты сравниваются с результатами анализа другими средствами поиска уязвимости DOM-based XSS.

1 ОСНОВНЫЕ ПОЛОЖЕНИЯ

На сегодняшний день количество людей, использующих в повседневной жизни Интернет, а соответственно и веб-приложения, достигает трех миллиардов[16]. Большая часть этих приложений[20] разработана с использованием языка программирования JavaScript.

JavaScript – динамический язык программирования. Наиболее часто он используется как часть веб-страниц, реализации которых позволяют клиентскому сценарию взаимодействовать с пользователем и создавать динамические страницы. Это - интерпретируемый язык программирования с объектно-ориентированными возможностями. Впервые JavaScript был представлен компанией Netscape в 1995 году под именем LiveScript.

Общие характеристики языка:

- легковесный;
- интерпретируемый;
- ориентирован на сетевые приложения;
- открытый;
- кроссплатформенный.

Наиболее распространенной формой языка является клиентский JavaScript. Документ HTML должен включать сценарий (или ссылаться на него), который затем будет интерпретироваться браузером. Это означает, что веб-страница не обязательно должна быть статической: она может включать программы, которые взаимодействуют с пользователем, управляют браузером и динамически генерируют содержимое страницы.

В связи с высокой популярностью JavaScript появилась необходимость в его безопасном использовании. В частности, программы, написанные на

этом языке наиболее подвержены уязвимости межсайтового скриптинга (Cross-site scripting, XSS).

Атакующий может использовать XSS, чтобы отправить вредоносный сценарий легитимному пользователю. Браузер конечного пользователя никаким образом не сможет определить, что сценарию получен из недоверенного источника и выполнит его. Поскольку подразумевается, что сценарий прибыл из доверенного источника, вредоносный сценарий может получить доступ к любым cookie, токенам сессии или другим приватным данным, сохраненным браузером. Если эти данные Эти сценарии могут даже переписать содержание страницы HTML. Ошибки, которые позволяют атакам успешно выполняться, довольно широко распространены и могут находиться в любой части веб-приложения, где используются входные пользовательские данные, которые затем используются в выводе данных без каких-либо проверок и кодирования.

Существует три типа межсайтового скриптинга:

1) отраженный (reflected);
2) хранимый (stored);
3) основанный на объектной модели документа (DOM-based).

Данная работа сфокусирована на последнем типе уязвимости - межсайтовом скриптинге, основанном на объектной модели документа, который является уязвимостью нарушения целостности исходя из определения целостности информации. Атака происходит на клиентской стороне приложения и не требует взаимодействия с сервером, поэтому отследить такую атаку затруднительно. Изменения в выполнении клиентского сценария вызывает модификация злоумышленником объектной модели документа. Атака может быть реализована посредством передачи на вход приложению специально сформированной строки – XSS-вектора.

Через следующие методы языка JavaScript, названные исследователями[8] “source”, непроверенные пользовательские данные могут попасть в веб-приложение:

- document.getElementById();
- document.cookie;
- document.documentURI;
- document.referrer;
- document.URL;
- window.name;
- методы объекта location: href, search, hash, pathname.

Методы, которые выполняют JavaScript-код, переданный через source, получили название sink. Эти методы можно разбить на следующие группы.

1) Методы, позволяющие реализовать инъекцию JavaScript-кода, например, «eval» и «setTimeout». Примером XSS-вектора, позволяющего определить наличие уязвимости может быть строка «alert();».

2) Методы, позволяющие инъекцию тегов HTML, например, «document.write» и «document.innerHtml». Примером XSS-вектора, позволяющего определить наличие уязвимости может быть строка «<script>alert();</script>».

Однако, следует заметить, что в веб-приложении может быть реализована фильтрация входных данных, которую при определенных обстоятельствах не сможет корректно отфильтровать вредоносное содержимое (например, если экранированы специальные символы).

Автоматизированные средства поиска уязвимостей делятся на две группы:

1. Средства статического анализа, которые осуществляют поиск уязвимостей в исходном коде программы без ее непосредственного выполнения.
2. Средства динамического анализа, которые позволяют анализировать ход выполнения программы и влиять на него.

2 СУЩЕСТВУЮЩИЕ СРЕДСТВА ПОИСКА УЯЗВИМОСТЕЙ

Существует три способа поиска уязвимостей в программах:

1. Статический анализ исходного кода.
2. Динамический анализ хода выполнения программы.
3. Тестирование на проникновение, проводимое специалистами.

В этой главе среди автоматизированных средств рассматриваются только бесплатные, доступные каждому разработчику.

2.1 Статический анализ

Статическим анализом называют анализ исходного кода программы, проводимый без ее непосредственного выполнения. Также статическим анализом чаще всего называют анализ, проведенный автоматизированными средствами – статическими анализаторами.

Чаще всего статический анализ выполняется в два этапа: построение абстрактного синтаксического дерева (АСД) программы, затем его анализ.

Перед непосредственным выполнением анализа исходного кода статический анализатор должен представить код в понятной и удобной для него форме, разбить на код на элементы, создать его структуру. Эта структура называется абстрактным синтаксическим деревом. Построение абстрактного синтаксического дерева в свою очередь также делится на два этапа. На первом этапе исходный код программы проходит через лексический анализатор (лексер). Лексер последовательно разбивает строки исходного кода на лексемы – зарезервированные слова, идентификаторы, константы. Затем он определяет тип каждой лексемы, после чего передает их на вход

синтаксическому анализатору (парсеру). В процессе лексического анализа программы могут возникнуть лексические ошибки, если анализатор не сможет определить тип лексемы. В таком случае лексер передает информацию об ошибке парсеру, который затем ее обрабатывает.

Правила, по которым лексер разбивает исходный код на лексемы, а парсер затем строит синтаксическое дерево и проверяет его на правильность, называются грамматикой языка.

При реализации статического анализатора могут использоваться различные техники анализа абстрактного синтаксического дерева, которые могут быть объединены в единое решение. В большинстве своем эти техники заимствованы из технологии компиляции. Следующие техники являются наиболее популярными при разработке статических анализаторов:

1) Поиск по сигнатурам[12]. На этой технике основаны простейшие статические анализаторы. Они содержат базу сигнатур – потенциально опасных конструкций, – с которыми сравниваются конструкции из исходного кода программы. Сигнатура найдена в исходном коде, то он помечается как уязвимый.

2) Анализ потоков данных[12]. Используется, чтобы собрать динамическую информацию о выполнении программы, в то время как она находится в статическом состоянии. Существует три распространенных термина, используемых в анализе потока данных: базисный блок (код), анализ потока управления (Control Flow Analysis) и путь потока управления (Control Flow Path).

3) Taint analysis[11]. Данная техника анализирует все точки входа в приложение, подконтрольные пользователю. Taint анализ позволяет проследить путь пользовательских данных от точек входа (source) до потенциально опасных методов и функций языка программирования (sink).

Можно выделить следующие плюсы статических анализаторов:

а) Возможность выявления уязвимости на начальном этапе разработки приложения. Чем раньше уязвимость обнаружена, тем легче и дешевле ее закрыть. Стив Макконнелл в книге «Совершенный Код»[2] приводит данные, согласно которым исправление ошибки на этапе написания кода обойдется в десять раз дешевле, чем на этапе тестирования.

б) Полное покрытие кода. Некоторые фрагменты программы выполняются крайне редко при определенных обстоятельствах. При статическом анализе даже такие фрагменты проходят проверку, что позволяет обнаружить ошибки например, в обработчиках ошибок или в системе журналирования, а также позволяет обнаружить «мертвый код».

Главным минусом статических анализаторов являются ошибки второго рода, то есть ложноположительные срабатывания. Статический анализатор в состоянии показать только подозрительные места и конструкции в исходном коде программы. Окончательный вывод относительно наличия уязвимости разработчик или исследователь должен делать сам. И если у специалиста по информационной безопасности достаточно знаний для принятия подобного решения, то у обычного разработчика такая задача вызовет определенные трудности. Кроме того, количество ложных срабатываний может быть настолько велико, что поиск уязвимостей вручную (code-review) может оказаться более эффективным по времени способом.

На настоящий момент существует множество статических анализаторов, в том числе специализирующихся на поиске уязвимостей. Примером таких анализаторов для языка программирования JavaScript являются сканеры ScanJS и JSPrime.

Продемонстрировать основной недостаток статических анализаторов можно с помощью результатов анализа JavaScript-библиотеки OpenUI5[18]

при помощи сканера ScanJS. Данным сканером было обнаружено 1002 строки исходного кода, содержащие потенциально опасные конструкции. Очевидно, что на обработку данных результатов уйдет значительное время.

2.2 Динамический анализ

Динамическим анализом называют анализ, основанный на реальном выполнении программы. Чаще всего программа выполняется в заранее подготовленном безопасном окружении – песочнице. При этом у пользователя есть возможность получать данные о выполнении программы и даже влиять на него.

Процедуру динамического анализа можно условно разделить на три этапа:

- инструментация программы;
- профилирование программы;
- анализ полученной информации.

На первом этапе осуществляется подготовка программы к анализу. В процессе профилирования (т.е. на втором этапе) собирается информация, которая затем обрабатывается соответствующим образом на заключительном этапе динамического анализа.

Непосредственно перед анализом программы необходимо подготовить ее к этому процессу. Различные модификации программы в этих целях называют инструментацией. Эти модификации представляют собой вставку особых анализирующих функций, которые чаще всего реализованы в виде функций обратного вызова, срабатывающих при определенных событиях. Инструментация бывает трех видов:

1) Инструментация исходного кода.
2) Инструментация байт-кода.
3) Инструментация исполняемого бинарного файла.

В основном JavaScript-программы выполняются встроенными в веб-браузеры программами для преобразования HTML-разметки сайта в представление в браузере. Хотя эти программы зачастую используют байт-код, сами программы передаются в виде исходных кодов. Поэтому для анализа программ на языке JavaScript используется инструментация исходного кода. При данном виде инструментации анализирующие функции вставляются непосредственно в исходный код программы.

Плюсом динамического анализа является низкая вероятность ложных срабатываний. Если уязвимость обнаружена с помощью динамического анализа, то вероятность того, что она на самом деле отсутствует мала. Ошибки такого рода называются ложно-позитивными (false-positive) срабатываниями.

Кроме того, с помощью динамического анализа можно обнаруживать ошибки в контексте конкретной работающей системы, как реальной, так и моделируемой.

Однако, для обнаружения уязвимости в программе необходимо, чтобы кодовый путь, содержащий ошибку, был пройден во время выполнения программы, следовательно необходимо высокое покрытие кода.

Главной проблемой применения динамического анализа для поиска уязвимостей DOM-based XSS является отсутствие бесплатных средств анализа, основанных на этом методе и специализирующихся конкретно на информационной безопасности.

2.3 Тестирование на проникновение

Третьим способом поиска уязвимостей является тестирование на проникновение, проводимое специалистами в области информационной безопасности (аудит). Услугами таких специалистов часто пользуются крупные компании, дорожащие своей репутацией.

При тестировании веб-приложения на наличие уязвимости DOM-based XSS аудитор действует по следующему алгоритму. Сначала он определяет точки входа приложения, через которые он сможет передать специально сформированные данные – XSS-вектора. Такими точками входа обычно являются различные формы на сайте (поиск, форма регистрации и так далее), а также строка URL-адреса. На следующем этапе аудитор начинает передавать различные XSS-вектора на вход приложению. Эти вектора всегда содержат JavaScript-код, выполнение которого будет служить для тестировщика сигналом о существовании уязвимости. Чаще всего это функция alert(), вызывающая всплывающее окно с текстом, переданным в качестве аргумента.

В результате тестирования на проникновение аудитор может однозначно определить наличие уязвимости, но зачастую не может точно указать уязвимое место в исходном коде программы.

Ввиду малого количества ложно-позитивных срабатываний для реализации цели данного исследования был выбран динамический анализ хода выполнения программ. Однако, при разработке представленного в данной работе средства поиска уязвимостей DOM-based XSS автор стремился сохранить такое преимущество статического анализа, как возможность анализировать программы на любом этапе разработки. Кроме того, разрабатываемое средство призвано сократить время поиска конкретного местоположения уязвимости в исходном коде программы.

3 РАЗРАБОТКА СРЕДСТВА ПОИСКА УЯЗВИМОСТЕЙ НАРУШЕНИЯ ЦЕЛОСТНОСТИ

Все действия, необходимые для подготовки исходного кода к анализу (инструментирование), выполнялись на основе программной платформы Node.js[17] в операционной системе OS X Yosemite. Анализируемая программа выполнялась в браузере Google Chrome.

За основу разрабатываемого метода был взят алгоритм поведения аудитора, проводящего тестирование на проникновение. Точками входа считались те места в исходном коде, где программа получала входные значения и следующих методов:

- document.getElementById();
- document.cookie;
- document.documentURI;
- document.referrer;
- document.URL;
- window.name;
- методы объекта location: href, search, hash, pathname.

Для инструментирования исходного кода программы сначала необходимо построить ее абстрактное синтаксическое дерево. Эта задача была выполнена с помощью парсера для многоцелевого анализа Esprima[14]. В Приложении 1 представлено абстрактное синтаксическое дерево, построенное на основе простейшей программы, представленной в Листинге 1. Далее в данной работе пример из Листинга 1 также будет использоваться для демонстрации деталей технической реализации разрабатываемого средства.

```
var a = document.referrer;
document.write(a);
```

Листинг 1 – пример простейшей программы на языке JavaScript.

Следующим шагом инструментирования являлась модификация абстрактного синтаксического дерева. Она была реализована при помощи библиотеки falafel[15], позволяющей изменять элементы АСД, построенного при помощи парсера Esprima.

Модификация исходного кода проводилась в два этапа. На первом этапе все значения, полученные из “источников” заменялись на XSS-вектор. Для этого совершался обход синтаксического дерева, во время которого определялось наличие следующих ситуаций:

1. Переменной присваивается значение, полученное из source.
2. Метод, являющийся «источником», используется в качестве аргумента другого метода или функции.

В случае обнаружения подобных ситуаций метод-«source» заменялся на строку, содержащую XSS-вектор.

Для передачи XSS-векторов на вход программе было необходимо выполнить следующую команду в терминале:

node falafelPassVector.js input.js output.js,

где falafelPassVector.js – сценарий, выполняющий модификацию абстрактного синтаксического дерева;

input.js – файл, содержащий исходный код анализируемой программы;

output.js – файл, в который будет записан модифицированный исходный код.

На втором этапе модификации исходного кода производилось дополнение исходного кода функциональными вставками, позволяющими отслеживать ход выполнения программы. Первым делом совершался новый

обход абстрактного синтаксического дерева, при котором все переменные, содержащие XSS-вектор записывались в массив sources. Затем рассматривались все функции и методы, используемые в анализируемой программе. Если функция (метод) содержала в качестве аргумента XSS-вектор или переменную из массива sources, то эта функция (метод) оборачивалась в конструкцию try..catch во избежание возможных ошибок при выполнении программы. Это было необходимо, так как в определенных случаях данные, переданные в качестве XSS-вектора могли вызвать исключение, которое, будучи, необработанным, привело бы к остановке выполнения программы. Далее имя функции (метода) добавлялось в массив route. Таким образом, к концу выполнения программы массив route содержал последовательность из имен функций (методов), через которые прошли потенциально опасные данные. Далее автор будет называть данную последовательность маршрутом. Заключительными шагами инструментирования исходного кода были вывод маршрута в консоль по завершению выполнения программы, а также заключение всего исходного кода теги <script>…</script> для последующего запуска в браузере Google Chrome. Для завершения инструментирования исходного кода программы необходимо выполнить следующую команду в командной строке:

node falafelInstrument.js output.js instrumented.html,

где falafelInstrument.js – сценарий, выполняющий дополнение исходного кода функциональными вставками;

output.js – файл, полученный применением сценария falafelPassVector.js;

instrumented.html – файл, содержащий инструментированную программу, готовый к запуску в браузере.

Листинг 3 содержит пример инструментированной программы.

```
<script>
ver route = [];
var a = '<script>alert();<\/script>';
try {
document.write(a);
route.push('document.write')
} catch (err){}
console.log(route.join(' '));
</script>
```

Листинг 3 – пример инструментированного исходного кода программы.

Изменение XSS-вектора производилось непосредственным изменением переменной «vector» в исходном коде сценариев. Исходный код сценариев falafelPassVector.js и falafelInstrument.js содержат Приложение 2 и Приложение 3 соответственно. Последовательность действий, которые выполняются при анализе программы разработанным средством, отражена на Рисунке 1.

Рисунок 1 – Анализ программы разработанным средством

4 ПОИСК УЯЗВИМОСТЕЙ ПРИ ПОМОЩИ РАЗРАБОТАННОГО СРЕДСТВА

Разработанное средство тестировалось в два этапа. На первом этапе автором данного исследования были сформированы простейшие тесты, призванные обеспечить высокое покрытие кода. Данные тесты проводились непосредственно во время разработки. Тесты содержали все методы «источники», приведенные в данной работе, а также различные конструкции языка, как безопасные, так и приводящие к появлению уязвимости.

Чтобы подтвердить корректность работы и эффективность разработанного средства поиска уязвимостей DOM-based XSS на втором этапе тестирования было необходимо обнаружить уязвимость в реальном проекте веб-приложения. Поиск уязвимого веб-приложения проводился в рамках веб-сервиса для хостинга ИТ-проектов и их совместной разработки GitHub. Уязвимость была обнаружена в репозитории проекта tiantiio.github.io[19]. Для удобства автор будет называть файл, содержащий уязвимый сценарий Real.js.

Результатом анализа программы Real.js являлся маршрут XSS-вектора, выведенный в консоль браузера Google Chrome. Для анализируемой программы маршрут состоял всего из двух методов: «document.write» и «match».

На Рисунке 2 графически представлены результаты анализа программы Real.js с помощью разработанного средства. Узлы на схеме обозначают функции (методы), используемые в данной программе, а связи - пути данных от функции к функции. Точка входа обозначена зеленым цветом. Узлы, обозначенные красным цветом, указывают на то, что указанные функции (методы) получили в качестве аргумента потенциально опасные данные.

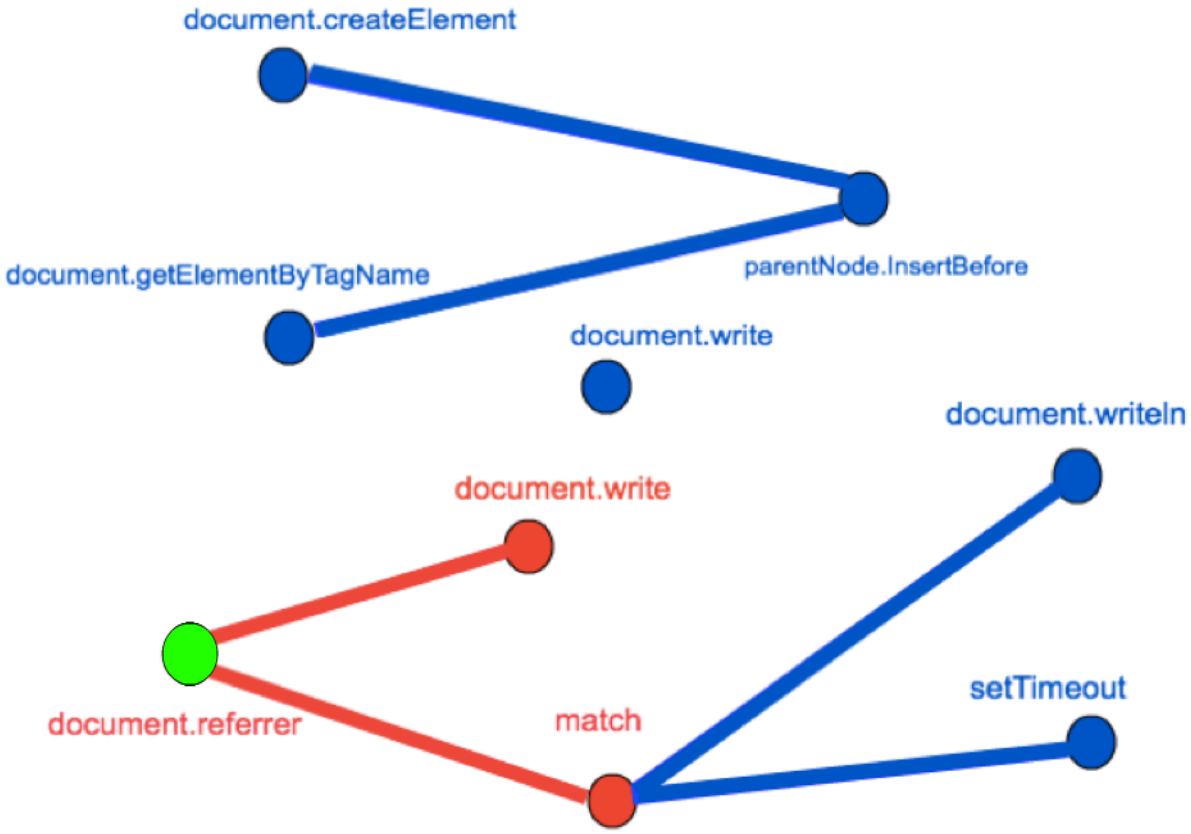

Рисунок 2 – Результаты анализа программы Real.js, полученные при помощи разработанного средства

Метод «match» определяет соответствие данных указанному регулярному выражению, следовательно к уязвимости межсайтового скриптинга приводит метод «document.write».

Для сравнения на Рисунке 3 приведены данные, полученные анализом программы Real.js с помощью средства статического анализа ScanJS. Сканер уязвимостей ScanJS отметил, как потенциально опасные, строки, содержащие методы «document.write», «document.writeln» и «setTimeout». Все три метода могут привести к появлению уязвимости - «document.write», «document.writeln» позволяют внедрять теги HTML на веб-страницу, а «setTimeout» напрямую выполняет JavaScript-код. Поэтому для определения конкретного местоположения уязвимости в исходном коде программы

понадобится дополнительное исследование без использования автоматизированных средств (code-review).

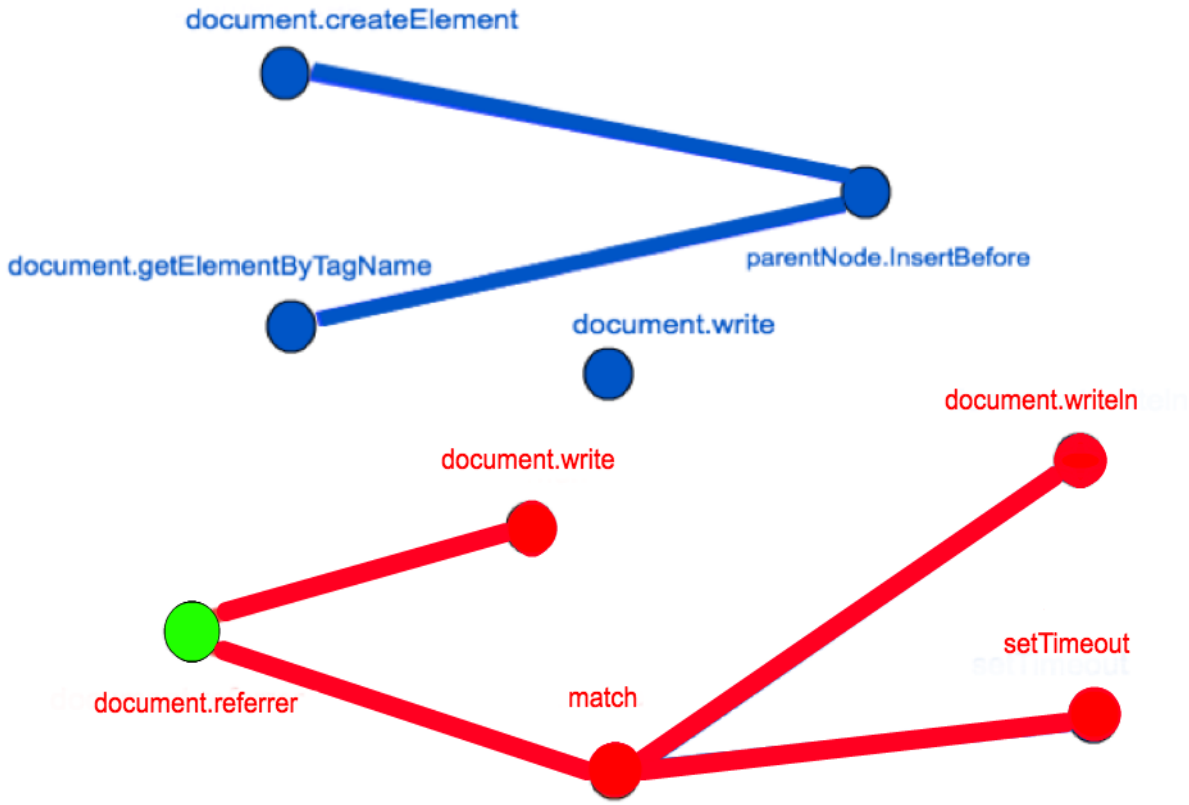

Рисунок 3 – Результаты анализа программы Real.js средством статического анализа ScanJS

Таким образом, было наглядно продемонстрированно, что разработанное средство сокращает время локализации уязвимости DOM-based XSS в исходном коде программы по сравнению с другими бесплатными средствами анализа (статический анализатор JSPrime использует тот же метод анализа, что и ScanJS).

Однако в программе может быть реализована фильтрация входных данных, преодолеть который может не каждый XSS-вектор. Поэтому дальнейшее совершенствование средства может включать сбор некоторой базы XSS-векторов, которые затем последовательно будут использоваться для анализа программ. Такой подход позволит не только определять наличие

более сложных уязвимостей, но и поможет выявить недостатки фильтрующих функций. Еще одним перспективным направлением дальнейшей разработки может быть увеличение точности локализации уязвимости до одного метода или небольшой группы методов.

ЗАКЛЮЧЕНИЕ

В рамках данной работы автором было разработано средство поиска уязвимостей нарушения целостности, в частности уязвимости межсайтового скриптинга, основанного на объектной модели документа, динамическим анализом в программах на языке JavaScript. Разработанное средство было использовано на реальных данных, были получены лучшие результаты по сравнению с другими аналогичными средствами. Ниже представлен детальный список результатов:

- Определены основные особенности уязвимости межсайтового скриптинга, основанного на объектной модели документа;

- Произведен обзор существующих методов поиска уязвимости DOM-based XSS, доступных каждому разработчику;

- Разработано новое средство поиска уязвимостей;

- Сформированы тесты для обеспечения высокого покрытия кода;

- С помощью разработанного средства найдена уязвимость в реальном проекте tiantiio.github.io с GitHub ;

- Результаты анализа разработанным средством сравнены с результатами анализа, проведенным анализатором ScanJS;

- Определены дальнейшие направления по совершенствованию разработанного средства.

В итоге, разработанное средство отвечает целям и задачам, поставленным в данной работе, а именно оно:

- определяет наличие уязвимости DOM-based XSS, которая относится к классу уязвимостей нарушения целостности;
- позволяет анализировать исходный код программы на любом этапе разработки;
- результаты анализа понятны не только специалистам по информационной безопасности, но и разработчикам, что является особенно актуальным ввиду большого количества разработчиков веб-приложений, не привязанных к конкретной компании.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. С.П. Вартанов, А.Ю. Герасимов. Динамический анализ программ с целью поиска ошибок и уязвимостей при помощи целенаправленной генерации входных данных. // Труды ИСП РАН том 26 вып. 1, 2014. С. 375-394.
2. Макконнелл С. Совершенный код = CODE COMPLETE : Мастер-класс / С. Макконнелл. - М. : Рус. Ред. ; СПб. : Питер, 2008, 2008. - 867 с. : ил. - Библиогр.: с. 842-862. - Предм. указ.: с. 863-867. - ISBN 978-5-7502-0064-1: 647-37
3. Д. Евдокимов. Инструментация – эволюция анализа [Электронный ресурс] // xakep.ru URL: https://xakep.ru/2013/09/11/61232/ Режим доступа: свободный. (дата обращения: 21.02.2016)
4. Рекомендации по стандартизации. «Информационные технологии. Основные термины и определения в области технической защиты информации». Р 50.1.053-2005.
5. G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. // In ACM SIGPLAN conference on Programming language design and implementation, pages 1{12. ACM, 2010.
6. M. Ishrat, M. Saxena, and M. Alamgir. Comparison of static and dynamic analysis for runtime monitoring. // International Journal of Computer Science & Communication Net-works, 2(5), 2012.
7. T. Ball. The concept of dynamic analysis. // In Software EngineeringESEC/FSE99, pages 216–234. Springer, 1999.
8. A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. // Web Application Security Consortium, 2005.

9. J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. // In 12th Annual Network and Distributed System Security Symposium, 2005

10. Koushik Sen and Swaroop Kalasapur and Tasneem G. Brutch and Simon Gibbs. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. // In Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013, 2013.

11. S. Wei and B. G. Ryder. Practical Blended Taint Analysis for JavaScript. // In International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013, pages 336–346. ACM, 2013.

12. Wögerer, W.: A survey of static program analysis techniques. // Technical report, Technische Universität Wien, 2005

13. O. Segal. Client-side JavaScript security vulnerabilities [Электронный ресурс] // slideshare.net URL: http://www.slideshare.net/orysegal/clientside-javascript-vulnerabilities Режим доступа: свободный. (дата обращения: 27.02.2016)

14. ECMAScript parsing infrastructer for multipurpose analysis [Электронный ресурс] // esprima.org URL: http://esprima.org/index.html Режим доступа: свободный. (дата обращения: 05.04.2016)

15. Falafel – tool for AST modification [Электронный ресурс] // GitHub URL: https://github.com/substack/node-falafel Режим доступа: свободный. (дата обращения: 08.04.2016)

16. Internet users [Электронный ресурс] // internetlivestats.com URL: http://www.internetlivestats.com/internet-users/ Режим доступа: свободный. (дата обращения: 11.03.2016)

17. Node.js®, JavaScript runtime built on Chrome's V8 JavaScript engine [Электронный ресурс] // nodejs.org URL: https://nodejs.org/en/ Режим доступа: свободный. (дата обращения: 23.03.2016)

18. Open Source JavaScript UI library, maintained by SAP [Электронный ресурс] // openui5.org URL: http://openui5.org/ Режим доступа: свободный. (дата обращения: 15.03.2016)

19. Project tiantiio.github.io [Электронный ресурс] // GitHub URL: https://github.com/tiantiio/tiantiio.github.io Режим доступа: свободный. (дата обращения: 30.04.2016)

20. Top Programming Languages Used in Web Development [Электронный ресурс] // Cleverism Magazine URL: https://www.cleverism.com/programming-languages-web-development/ Режим доступа: свободный. (дата обращения: 28.02.2016)

21. Top-10 vulnerabilities by OWASP [Электронный ресурс] // ibm.com URL: http://www.ibm.com/developerworks/library/se-owasptop10/se-owasptop10-pdf.pdf Режим доступа: свободный. (дата обращения: 28.02.2016)

ПРИЛОЖЕНИЯ

Приложение 1. Пример построения абстрактного синтаксического дерева с помощью парсера Esprima.


```
{
  "type": "Program",
  "body": [
    {
      "type": "VariableDeclaration",
      "declarations": [
        {
          "type": "VariableDeclarator",
          "id": {
            "type": "Identifier",
            "name": "a"
          },
          "init": {
            "type": "MemberExpression",
            "computed": false,
            "object": {
              "type": "Identifier",
              "name": "document"
            },
            "property": {
              "type": "Identifier",
              "name": "referrer"
            }
          }
        }
      ],
      "kind": "var"
    },
    {
      "type": "ExpressionStatement",
      "expression": {
        "type": "CallExpression",
        "callee": {
          "type": "MemberExpression",
          "computed": false,
          "object": {
```

```
                    "type": "Identifier",
                    "name": "document"
                },
                "property": {
                    "type": "Identifier",
                    "name": "write"
                }
            },
            "arguments": [
                {
                    "type": "Identifier",
                    "name": "a"
                }
            ]
        }
    }
],
"sourceType": "script"
}
```

Приложение 2. Исходный код сценария falafelPassVector.js.

```
        var fs = require('fs');
var falafel = require('falafel');

var filename = process.argv[2];

var vector = '<script>alert();<\\/script>';
var src1 =
['referrer','URL','documentURI','cookie','name','getElementById','href','search','path
name','hash'];

var code = fs.readFileSync(filename) + '';

code = falafel(code, function(node){

  //Pass XSS-vector to source

  if (node.type === 'ExpressionStatement'){
  if (node.expression.arguments){
  for (var k=0; k < node.expression.arguments.length; k++){
    try{
       if ((node.expression.arguments[k].object.name === 'document' ||
node.expression.arguments[k].object.name === 'window') &&
(src1.indexOf(node.expression.arguments[k].property.name)>-1)){
          node.expression.arguments[k].update('\'' + vector + '\'');
       }
    } catch(err){};
    try{
       if ((node.expression.arguments[k].object.object.name === 'document' ||
node.expression.arguments[k].object.object.name === 'window') &&
(node.expression.arguments[k].object.property.name === 'location') &&
(src1.indexOf(node.expression.arguments[k].property.name)>-1)){
          node.expression.arguments[k].update('\'' + vector + '\'');
       }
    } catch(err){};
    }
    }

  try{
```

```
    if ((node.expression.right.callee.object.name === 'document' ||
node.expression.right.callee.object.name === 'window') &&
(src1.indexOf(node.expression.right.callee.property.name)>-1)){
        node.expression.right.update('\" + vector + '\");
    }
    } catch(err){};
 try{
      if ((node.expression.right.object.name === 'document' ||
node.expression.right.object.name === 'window') &&
(src1.indexOf(node.expression.right.property.name)>-1)){
          node.expression.right.update('\" + vector + '\");
      }
      } catch(err){};
 try{
    if ((node.expression.right.object.object.name === 'document' ||
node.expression.right.object.object.name === 'window') &&
(node.expression.right.object.property.name === 'location') &&
(src1.indexOf(node.expression.right.property.name)>-1)){
        node.expression.right.update('\" + vector + '\");
    }
  } catch(err){};
 }


 if (node.type === 'VariableDeclaration') {
  for (var i=0; i<node.declarations.length; i++) {

    try {
       if ((node.declarations[i].init.object.name === 'document'||
node.declarations[i].init.object.name === 'window') &&
(src1.indexOf(node.declarations[i].init.property.name)>-1)){
           node.declarations[i].init.update('\" + vector + '\");
       }
    } catch(err) {}
    try{
       if ((node.declarations[i].init.object.object.name === 'document' ||
node.declarations[i].init.object.object.name === 'window') &&
(node.declarations[i].init.object.property.name === 'location') &&
(src1.indexOf(node.declarations[i].init.property.name)>-1)){
           node.declarations[i].init.update('\" + vector + '\");
       }
    } catch(err){};
```

```
    }
  }
});

fs.writeFile(process.argv[3], code, console.log('done!'));
```

Приложение 3. Исходный код сценария falafelInstrument.js.

```
        var fs = require('fs');
var falafel = require('falafel');

var filename = process.argv[2];

var vector = '<script>alert();<\/script>';
var sources = [];

var code = fs.readFileSync(filename) + '';

code = falafel(code, function(node){


  //Array of variables containing XSS-vector

  if (node.type === 'VariableDeclarator' && node.init.value === vector){
     sources.push(node.id.name);
  }

  //Instrument code

  if (node.type === 'Program'){
  node.update('<script>\nvar route = [];\n' + node.source() +
'\nconsole.log(route.join(\' \'));\n<\/script>')
  }
  for (var i=0; i < sources.length; i++){

    if (node.type === 'ExpressionStatement' && node.expression.arguments) {
       for (var j=0; j < node.expression.arguments.length; j++){
       if (node.expression.arguments[j].name === sources[i] ||
node.expression.arguments[j].value === vector){
          if (node.expression.type === 'CallExpression'){
             if (node.expression.callee.name) {
                node.update('try {\n' + node.source() + '\nroute.push(\" +
```

```
node.expression.callee.name + '\');' + '\n} catch(err){}');
            }
        }
        if (node.expression.callee.type === 'MemberExpression'){
            node.update('try {\n' + node.source() + '\nroute.push(\" +
node.expression.callee.object.name + '.' + node.expression.callee.property.name +
'\');' + '\n} catch(err){}');
            }
      }
      }
  }

  if (node.type === 'VariableDeclaration') {
    for (var k=0; k<node.declarations.length; k++) {
      if (node.declarations[k].init.arguments){
      for (var j=0; j < node.declarations[k].init.arguments.length; j++){

            if (node.declarations[k].init.arguments[j].name === sources[i] ||
node.declarations[k].init.arguments[j].value === vector){
                node.update('try {\n' + node.source() + '\nroute.push(\" +
node.declarations[k].init.callee.name + '\');' + '\n} catch(err){}');
            }
      }
      }
    }
    }
 }


});

fs.writeFile(process.argv[3], code, console.log('code instrumented!'));
```

Приложение 4. Исходный код программы Real.js

```
        var _hmt = _hmt || [];
(function() {
   var hm = document.createElement('script');
   hm.src = '//hm.baidu.com/hm.js?114678d958452b51a230e93038b58155';
   var s = document.getElementsByTagName('script')[0];
   s.parentNode.insertBefore(hm,s);
})();
var referrer = document.referrer;
document.write('refer');
document.write(referrer);
var rd = '(^.*localhost.*|.*localhost.*)';
if (referrer.match(rd)){
   document.writeln('等待5秒跳转');
   //window.location.href = document.referrer;
   setTimeout('javascript:location.href = referrer' ,5000);
}
```

Приложение 5. Инструментированный исходный код программы Real.js

```
<script>
var route = [];
var _hmt = _hmt || [];
(function() {
   var hm = document.createElement('script');
   hm.src = '//hm.baidu.com/hm.js?114678d958452b51a230e93038b58155';
   var s = document.getElementsByTagName('script')[0];
   s.parentNode.insertBefore(hm,s);
})();
var referrer = '<script>alert();<\/script>';
document.write('refer');
try {
document.write(referrer);
route.push('document.write');
} catch(err){}
var rd = '(^.*localhost.*|.*localhost.*)';
try {
if (referrer.match(rd)){
   document.writeln('等待5秒跳转');
   //window.location.href = document.referrer;
   setTimeout('javascript:location.href = referrer' ,5000);
}
route.push('referrer.match');
} catch(err){}
console.log(route.join(' '));
</script>
```