Московский Государственный Университет имени М.В. Ломоносова
Факультет Вычислительной Математики и Кибернетики
Кафедра Автоматизации Систем Вычислительных Комплексов

Чиботару Виктор Дорианович

Исследование способов эксплуатации недостатков веб-приложений, связанных с возможностью изменения области видимости переменных объектной модели веб-страницы

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Научный руководитель:
младший научный сотрудник
А.А. Петухов

Москва, 2016

Аннотация

В данной работе рассматривается метод автоматического анализа веб-приложений с целью выявления способов эксплуатации недостатков типа DOM Clobbering, связанных с возможностью изменения области видимости переменных объектной модели веб-страницы. Идея метода состоит в отслеживании и дальнейшем анализе потоков данных в коде веб-страницы, исполняемого на стороне клиента. В работе приводится обзор существующих методов анализа кода, написанного на языке JavaScript и предлагается использование метода тейнтирования (taint analysis). Предложенные идеи реализованы в виде дополнения к браузеру Mozilla Firefox. Для проверки корректности его работы было проведено тестирование как на синтетических наборах тестов, так и на реальных веб-приложениях.

Оглавление

1 Введение

В период зарождения всемирной паутины веб-страницы представляли собой статические документы, сверстанные на языке HMTL. Но, с течением времени, страницы становились все более и более динамическими. Сегодняшние приложения обычно содержат большое количество кода, исполняемого на стороне клиента (в веб-браузере). Стоит отметить, что несмотря на существование конкурентов, язык JavaScript используется в подавляющем большинстве веб-приложений для написания кода для клиентских частей. Об этом факте свидетельствует бурный рост популярности данного языка и технологий, которые его используют (например, NodeJS, AngularJS и т. д.)

Активное использование языка JavaScript привело к появлению новой парадигмы устройства веб-приложений, называемой Single Page Application (Одностраничное приложение). Работая с таким приложением, пользователь все время находится в рамках одной веб-страницы, в коде которой и реализована большая часть бизнес логики. Всё общение между клиентом и веб-сервером осуществляется с помощью AJAX запросов без необходимости перезагрузки или перехода на другие страницы.

Высокая популярность Single Page Application наглядно демонстрирует тенденцию к всё большему усложнению кода клиентской части. Однако, хорошо известно, что рост сложности системы часто ведёт к снижению её безопасности. В случае веб-приложений этот факт доказывается большим количеством недостатков на стороне клиента. Вот самые опасные и распространенные из них:

1. DOM Based XSS
2. DOM Redirection
3. Некорректное использование механизмов Same Origin Policy
4. DOM Clobbering

Самым распространенным из них является недостаток DOM Based XSS. В рейтинге недостатков веб-приложений OWASP (Open Web Application Security Project) Top Ten он занимает третье место. Он заключается в том, что код веб-страницы обрабатывает пользовательские данные и модифицирует её содержимое, позволяя злоумышленнику исполнять произвольные команды. Этот тип недостатков достаточно хорошо исследован и для него были разработаны эффективные автоматические средства обнаружения. В виду некоторых особенностей языка JavaScript (слабая типизация, возможность динамического исполнения кода) статические методы анализа кода оказываются непригодными для решения подобных проблем. Поэтому, разработанные решения опираются на такие методы, как динамический тейнт анализ и фаззинг.

Уязвимость, рассматриваемая в данной работе носит название DOM Clobbering (от англ. DOM - Document Object Model, Объектная Модель Документа и Clobber - перезаписывать). Суть DOM Clobbering заключается в возможности подмены объектов (переменных) веб-страницы с помощью изменения их области видимости. Как и DOM-based XSS, DOM Clobbering - это недостаток целиком на стороне клиента. Стоит отметить, что в реальных сайтах DOM Clobbering встречается редко,

что и является причиной его недостаточной изученности. Однако, высокая степень опасности, которую таят в себе недостатки данного типа, указывает на необходимость разработки метода и автоматизированного средства их поиска.

2 Постановка задачи

2.1. Неформальная постановка задачи

Рассмотрим на небольшом примере причины возникновения недостатка DOM Clobbering.

Пример 1:

```
1  <form name="form_name">
2  <script>
3    var form = document.form_name; // указываетна  <form name="
         form_name">
4  </script>
```

Для представления содержимого веб-страницы в виде объектов на языке JavaScript веб-браузеры используют интерфейс DOM (Объектная Модель Документа). В рамках DOM HTML-странице ставится в соответствие объект document, а окну веб-браузера - объект window. После загрузки и обработки страницы эти объекты заполняются различным свойствами (например, document.location - объект, содержащий информацию про расположение (URL) документа).

Так же дело обстоит и с тэгами страницы: после создания HTML элемента <x name='element_name' id='element_id'> он становится доступен по указателям document.element_name, window.element_name (верно для элементов одного из типов <img>, <form>, <embed>, <object> и <applet>) и window.element_id (верно для всех элементов), где element_name - это имя (атрибут name) созданного элемента, а element_id - это идентификатор элемента (атрибут id).

Аналогично в пространство имен документа попадают и объекты из JavaScript кода. Например, в результате загрузки страницы в пространство имен документа добавится выше описанное свойство document.location. То есть, свойства DOM могут заполняться как из контекста JavaScript кода, так и в результате обработки HTML элементов.

Такое поведение приводит к тому, что если в пространстве имен документа уже существовал объект document.x и HTML парсер встречает элемент <form name='x'>, то происходит конфликт имен и document.x начинает указывать на HTML форму. В этом заключается вся суть эксплуатации недостатка DOM Clobbering: злоумышленник подбирает имя HTML элемента таким образом, чтобы заменить какой-нибудь важный объект в объектной модели документа страницы.

Рассмотрим три типичных примеров недостатка DOM Clobbering.

Пример 2:

```
1  <form name="querySelector">
2  <script>
3    var element = document.querySelector("a"); // ошибка: document.
         querySelector указываетна  <form name“=”querySelector
4  </script>
```

В данном примере пользователь путем взаимодействия с приложением может изменять значение атрибута name (имя) у формы на странице и устанавливает его равным “querySelector”. Далее, в коде вызывается вызывается функция

document.querySelector, но, так как document.querySelector теперь указывает на HTML форму, которая не является функцией, при таком вызове произойдет ошибка. То есть, действуя подобным образом, злоумышленник может добиться нарушения работоспособности кода на стороне клиента.

Пример 3:

```
<form name="is_in_black_list">
<script>
  if (user_in_black_list()) {
    is_in_black_list = true;
  }
  if (is_in_black_list) {
    forbid_action();
  }
</script>
```

В данном примере пользователь путем взаимодействия с приложением может изменять значение атрибута name (имя) у формы на странице и устанавливает его равным “is_in_black_list". Далее вызывается функция user_in_black_list, суть которой заключается в проверке находится ли пользователь в черном списке приложения. Если это так, то глобальной переменной is_in_black_list присваивается значение true, иначе она остается неинициализированной. Далее, в зависимости от значения переменной is_in_black_list пользователю разрешается или запрещается выполнение какого-нибудь действия. Но, так как имя формы “is_in_black_list”, то во втором условном операторе is_in_black_list будет указывать на HTML форму, следовательно, пользователю будет отказано в выполнение действия. Заметим, что данный пример является несколько вырожденным, но он ясно дает понять, что в некоторых случаях злоумышленники могут обойти логику работы кода на стороне клиента с помощью недостатка DOM Clobbering и тем самым навредить пользователям.

Пример 4:

```
<a href="plugins/preview/preview.html#<svg onload=alert(1)>" id="
    _cke_htmlToLoad" target="_blank">
  Click me!
</a>файл

 /plugins/preview/preview.html:
<script>
  ...
  document.write(window.opener._cke_htmlToLoad);
  ...
</script>
```

Данный пример взят из кода реального веб-приложения. На одной из его страниц размещалась ссылка (элемент <a>) с идентификатором, равным “_cke_htmlToLoad”, указывающая на страницу “plugins/preview/preview.html#<svg onload=alert(1)>”. Далее, после того, как пользователь переходил по этой ссылке, на странице plugins/preview/preview.html отрабатывал код, записывающий строку window.opener._cke_htmlToLoad в конец веб-страницы. Но, так как window.opener указывал на ту страницу, с которой был осуществлен переход на текущую, а в ней _cke_htmlToLoad указывал на контролируемый элемент <a>, в конец документа записывалась строка “plugins/preview/preview.html#<svg onload=alert(1)>”. А дописывание в документ строки <svg onload=alert(1)> означало создание HTML

элемента типа <svg>, при завершении загрузки которого выполнялся код alert(1). Таким образом, злоумышленник получал возможность внедрять и исполнять произвольный код на стороне клиента, что может привести к плохим последствиям.

2.2. Определение DOM Clobbering

Дадим определение DOM Clobbering: DOM Clobbering - это недостаток клиентской стороны веб-приложения, заключающийся в возможности подмены объектов (переменных) веб-страницы с помощью изменения имен и/или идентификаторов некоторых HTML элементов на веб-странице.

2.3. Цель работы

Цель данной работы - сформулировать методику и разработать инструментальное средство для определения возможности (и последствий) подмены объектов DOM для заданной веб-страницы, а также провести исследование их применимости.

3 Анализ задачи

3.1. Исследование возможных направлений атаки

После исследования реальных примеров веб-страниц, обладающих недостатком DOM Clobbering, авторами были выявлены следующие возможные направления атаки.

Вызов ошибки в программе

Во-первых, злоумышленник может вызвать возникновение ошибки в коде клиентской стороны. Самый простой пример - это замена указателя на стандартную функцию, предоставляемую интерфейсом браузера, указателем на HTML элемент (пример 2 из главы Постановка задачи). Последствиями такой атаки является некорректная дальнейшая работа веб-приложения.

Обход критических состояний

Для описания второго направления, проанализируем пример 3 из главы Постановка задачи:

```
1  <form name="is_in_black_list">
2  <script>
3    if (user_in_black_list()) { // точка 1
4      is_in_black_list = true; // точка 2
5    }
6    if (is_in_black_list) { // точка 3
7      forbid_action(); // точка 4
8    }
9  </script>
```

Предположим, что функция user_in_black_list осуществляет посылку запроса на веб-сервер с целью выяснения находится ли пользователь в черном списке.

Введем пару определений:

- *Состояние программы в точке* i - это тройка $< i, Vars, Values >$, где i - точка программы, $Vars$ - множество переменных, доступных в i, $Values$ - множество двоек $< var, value >$, где $var \in Vars$, а $value$ - значение var в i.
- *Критическое состояние* - это такое состояние, в которое может перейти страница только при условии выполнения определенных требований на серверной стороне приложения.
- *Некритическое состояние* - это состояние, не являющееся критическим.
- *Допустимое для точки* i *состояние* - это состояние $< i, Vars, Values >$, в котором программа может оказаться.

Например, состояние $< 4, is\_in\_black\_list, < is\_in\_black\_list, true >>$ является критическим, так как для того, чтобы программа перешла в него, необходимо присутствие пользователя в черном списке.

С другой стороны, состояние $< 4, is\_in\_black\_list, < is\_in\_black\_list, HTMLForm >>$ также является допустимым для точки 4, но не является критическим.

Другими словами, для точки 4 существуют два допустимых состояния, одно из которых дает нарушителю возможность нанести вред пользователю, который не находится в черном списке. Очевидно, такая ситуация является угрозой безопасности приложения.

С помощью введенных выше определений второе направление атаки можно описать так: *С помощью недостатка DOM Clobbering нарушителям удается перевести программу в некритическое состояние в некоторой точке, для которой все прочие допустимые состояния являются критическими*

Выполнение произвольного кода

В-третьих, злоумышленники могут добиться исполнения произвольного кода в веб-браузере жертвы. Реальным примером такой атаки является пример 4 из главы Постановка задачи. Стоит отметить, что в данном случае методы эксплуатации идентичны методам эксплуатации недостатка DOM Based XSS. Данная атака является самой опасной из приведенных трех, так как она позволяет нарушителям осуществлять любые действия от имени жертвы.

3.2. Требования к решению

На основе проведенного исследования возможных направлений атаки были сформулированны следующие требования к к методике поиска недостатков и путей их эксплуатации:

1. Определение возможности возникновения ошибок из-за переименования HTML элемента и места таких ошибок.
2. Определение возможности и путей обхода критических состояний.
3. Определение возможности выполнения произвольного кода.

3.3. Методика решения задачи

Перед рассмотрением методики решения задачи, разработанной авторами, введем следующие определения:

Будем говорить, что **оператор O зависит от переменной A по данным**, если выполняется хотя бы одно из двух:

- Оператор O использует значение A.
- Оператор O зависит по управлению от оператора, который зависит от A.

Будем говорить, что **переменная A зависит от переменной B по данным**, если выполняется хотя бы одно из двух:

- Оператор, вычисляющий A, зависит от B.
- Оператор, вычисляющий A, зависит от переменной, которая зависит от B.

Приведем пару примеров:

```
1  a = "a";
2  b = a + "b"; // b зависитот a
3  c = b + "c"; // c зависитот b, азначитиот a
4  if (a === "a") {
5    d = "d"; // d зависитот a
6  }
7  else {
8    e = "e"; // e зависитот a
9  }
10 func(a); // операторвызовафункциизависитот a
```

Теперь всё готово для описания этапов методики:

1. *Этап подготовки* - для заданной веб-страницы с недостатком DOM Clobbering и заданного HTML элемента, имя которого можно изменять, выбирается очередное имя из заранее подготовленного списка имен.

2. *Этап анализа кода* - анализируется поток данных кода клиентской стороны приложения с целью построения списка всех переменных и операторов, зависимых по данным от рассматриваемого HTML элемента.

3. *Этап обработки результатов анализа* - по полученному списку зависимых переменных и операторов вычисляется значение некоторых предикатов.

4. *Этап вывода* - выводится полученный результат, то есть список всех зависимых переменных и вычисленные значения предикатов.

3.4. Обоснование корректности методики

Докажем, что полученная методика удовлетворяет всем требованиям, сформулированным выше.

Во-первых, рассмотрим пример с *атакой путем вызова ошибки*. Допустим, что в заранее подготовленном списке слов есть все имена HTML элемента, для которых в программе может произойти ошибка. Тогда, процесс исследования тривиален:

1. Выбрать следующее имя HTML элемента из списка имен.

2. Запустить программу и проверить произошла ли ошибка.

3. Если в списке имен еще остались имена, вернуться к шагу 1.

То есть, корректность решения зависит от полноты списка имен. Авторами были исследованы и выписаны все имена из стандартного пространства имен объекта document, которые уязвимы к недостатку DOM Clobbering. Эти имена будут присутствовать в DOM любой веб-страницы и представляют собой особую опасность. Для нестандартных имен в представленной ниже реализации существует функционал расширения встроенного списка.

Для проверки возможности проведения *атаки второго рода*, достаточно проверить зависит ли хотя бы одна переменная или хотя бы один оператор, соответствующие критическим участкам кода, от выбранного HTML элемента. Если такие зависимости обнаружены, то в данных участках кода будут возможны некритические состояния, что, как было показано выше, ведет к угрозе безопасности.

Как уже отмечалось выше эксплуатация недостатка DOM Clobbering *с выполнением произвольного кода* идентична эксплуатации недостатка DOM Based XSS. Для данного недостатка исследователями был получен следующий результат: проведение атак возможно если пользовательский ввод попадает в так называемые стоки (особые переменные и вызовы особых функции) без должной обработки. В терминах, введенных нами, это утверждение можно переформулировать следующим образом: *выполнение произвольного кода путем эксплуатации недостатка DOM Clobbering возможно в том случае, если хотя бы один сток зависит по данным от выбранного HTML элемента и данные попадают в этот сток без должной обработки*.

То есть, для ответа на вопрос о возможности проведения атак третьего вида, нужно ответить на двух других вопроса:

- Существует ли зависимый сток?
- Если да, то в каком виде в него попадают данные?

С помощью предложенной методики можно получить ответы на оба вопроса:

- Нужно проверить попал ли хотя бы один сток в список зависимых переменных и операторов.
- Нужно проверить какая обработка проводилась на пути от HTML элемента к стоку в графе потока данных.

На этом доказательство соответствия предложенной методики всем необходимым требованиям завершено.

4 Построение решения задачи

Опишем процесс построения решения задачи, основанного на предложенной выше методике.

4.1. Декомпозиция на подзадачи

Исходя из предложенной методики, можно очевидным образом разбить процесс построения решения на следующие подзадачи, соответствующие этапам методики:

- Задача выбора метода анализа кода на языке JavaScript для определения зависимостей по данным
- Задача создания среды для удобного проведения автоматического анализа кода

4.2. Выбор метода анализа кода

Итак, первая подзадача, которую нужно решить, - это задача выбора метода анализа кода (далее просто метода).

Перед методом стоит только одна задача: *проанализировать код заданной веб-страницы и построить для него ГПД (граф потока данных) и список переменных и операторов, которые зависят по данным от заданного HTML элемента.*

Основным критерием выбора метода является *корректность, то есть способность правильно строить ГПД и обнаруживать зависимости.* Все остальные свойства (такие как *скорость работы* или *простота реализации*) не играют столь важной роли, поэтому в данной работе не рассматриваются.

Существует две основные группы методов анализа JavaScript кода: статические и динамические. Рассмотрим их подробнее.

Статические методы

Под статическими методами анализа кода подразумевается методы, не предполагающие его исполнение. Такие методы работают только с исходным кодом приложения. Рассмотрим ряд особенностей языка JavaScript, затрудняющих применение статического анализа:

- Динамическая типизация
- Динамическое исполнения кода (функции eval, и т. п.)
- Вызов функций от произвольного числа аргументов
- Прототипное наследование

- Зависимость значения указателя this от контекста выполнения

Приведем пару примеров ситуаций, в которых методы статического анализа могут испытывать трудности:

Динамическая типизация:

```
var a = 1;
a = a + 'foo';
var b = typeof a; // b = 'string'
if (b === 'string') {
  console.log('a is string');
}
else if (b === 'number') {
  console.log('b is number');
}
```

Динамическое исполнение кода:

```
eval(atob("YWxlcnQoMSk7")); // эквивалентно alert(1);
```

Вызов функций от произвольного числа аргументов:

```
var f = function(a) {
  console.log('a is ' + a + ' ;arguments are: ' + arguments);

}
f(); // a is undefined ;arguments are: {}
f(1); // a is 1 ;arguments are: {'0': 1}
f(1, 2); // a is 1 ;arguments are: {'0': 1, '1': 2}
```

Прототипное наследование:

```
var Number = function() {
  this.num = 1;
}
var a = new Number();
var b = new Number();
a.num = 2;
delete Number.prototype.num;
console.log(a.num); // 1
console.log(b.num); // undefined
```

Зависимость значения указателя this от контекста выполнения:

```
var obj = {
  message: 'Hello World!'
  sayHello: function() {
    return this.message;
  }
}
var f = obj.sayHello;
obj.sayHello(); // 'Hello World!'
f(); // undefined
```

Описанные выше примеры иллюстрируют (хоть и в упрощенном виде) распространенные практики написания программ на языке JavaScript, которые часто встречаются в реальных веб-приложениях. Они наглядно показывают, что применяя только методы статического анализа достаточно трудно корректно восстановить потоки данных в коде. Следовательно, применение методов статического анализа нецелесообразно.

Динамические методы

В отличии от статических методов, динамические методы используют информацию, полученную во время работы программы (такую как, например, значения переменных или список вызванных функций). Из всего множества динамических методов наиболее подходящим для решаемой задачи является метод тейнтирования (также называемый тейнт-анализом, taint analysis). Как будет показано ниже, его основная идея близка к главной идее предложенной методики и позволяет эффективно обнаруживать потоки данных в коде.

Метод тейнтирования подразумевает выделение в коде двух особых типов объектов:

- *Истоки* - это объекты, через которые в программу попадает недоверенный пользовательский ввод.
- *Стоки* - это критически важные с точки зрения безопасности объекты, в которые не должен попасть необработанный пользовательский ввод.

Основная цель тейнт-анализа - выяснить попадает ли ввод из какого-либо истока в какой-либо сток в необработанном виде. Алгоритм работы метода состоит их трех этапов:

- *Пометка истоков (tainting)* - истоки помечаются особым образом.
- *Распространение пометок (propagation)* - при любом взаимодействии с помеченной переменной (присваивании, вызова функций, применение операторов, получение свойств, и т.д.) результат операции помечается.
- *Проверка* - проверяется существует ли помеченный сток.

Выводы

Из-за практической неприменимости методов статического анализа выбор авторов пал на методы динамического анализа, а именно на метод тейнтирования, так как он в идейном плане наиболее близок к предложенной методике.

4.3. Создание среды для анализа кода

Второй решаемой подзадачей является задача создания среды для анализа кода (далее просто среды).

Решаемые задачи

Здесь под средой подразумевается некоторое вспомогательное средство, которое должно решать следующие задачи:

1. Первичная обработка веб-страницы и (возможное) изменение некоторых ее свойств
2. Предоставление средству анализа кода интерфейса для доступа к содержимому веб-страницы
3. Интерпретация кода веб-страницы

Критерии оценки

Для оценки различных решений данной подзадачи были выбраны следующие критерии:

1. Простота разработки - среда должна требовать меньшее количество усилий на разработку, чем средство для анализа кода.
2. Переносимость - среда должна корректно работать на различных компьютерах с различными операционными системами.
3. Скорость - так как анализ кода является ключевым и достаточно долгим процессом, среда не должна тратить большое количество времени на работу.
4. Корректность - среда должна обрабатывать веб-страницы идентично тому, как это делают клиентские среды (веб-браузеры).

Авторами были проанализированы два способа создания такой среды, а именно:

- Модификация интерпретатора веб-браузера
- Создание дополнения для веб-браузера

Модификация интерпретатора

Данный подход заключается в модификации кода существующего интерпретатора программ на языке JavaScript. Примером является интерпретатор SpiderMonkey с открытым исходным кодом, входящий в состав веб-браузера Mozilla Firefox.

Проанализируем данный подход, основываясь на приведенных выше критериях:

1. Простота разработки - для модификации интерпретатора нужно сначала разобраться как он работает, а затем внести в него изменения. Авторам эта задача представляется сравнимой по сложности с задачей создания средства анализа кода.
2. Переносимость - разработанная таким образом среда будет обладать одним большим недостатком - при выпуске обновлений интерпретатора её нужно будет изменять соответственно.
3. Скорость - интерпретаторы, входящие в состав веб-браузеров являются хорошо оптимизированными программами, поэтому скорость работы среды будет высокой.
4. Корректность - разработанная таким образом среда будет работать абсолютно идентично веб-браузеру.

Создание дополнения для веб-браузера

Современные веб-браузеры предоставляют API (Application Programming Interface, Интерфейс Программирования Приложений), дающие возможность использования функционала браузера. На основе таких API создаются приложения для автоматизации работы с браузерами. Одним из таких приложений является SlimerJS, работающий с Mozilla Firefox.

SlimerJS предоставляет возможность решения всех задач, описанных в секции Решаемые Задачи. Проанализируем данный подход, основываясь на приведенных выше критериях:

1. Простота разработки - SlimerJS предоставляет простые и понятные интерфейсы для работы с браузером, освоение которых не занимает много времени.

2. Переносимость - работа SlimerJS зависит только от уставленного на компьютере браузера Mozilla Firefox, но не от операционной системы.

3. Скорость - использование SlimerJS будет затормаживать работу всей системы, но, как показали практические эксперименты, падение скорости является незначительным.

4. Корректность - разработанная таким образом среда будет работать абсолютно идентично веб-браузеру.

Выводы

Для наглядного сравнения двух возможных решений, сведем результаты анализа в следующую таблицу:

	Модификация интерпретатора	Создание дополнения для веб-браузера
Простота		
Переносимость		
Скорость		
Корректность		

Из-за большой сложности первого метода, предпочтение было отдано созданию среды в виде дополнения к веб-браузеру.

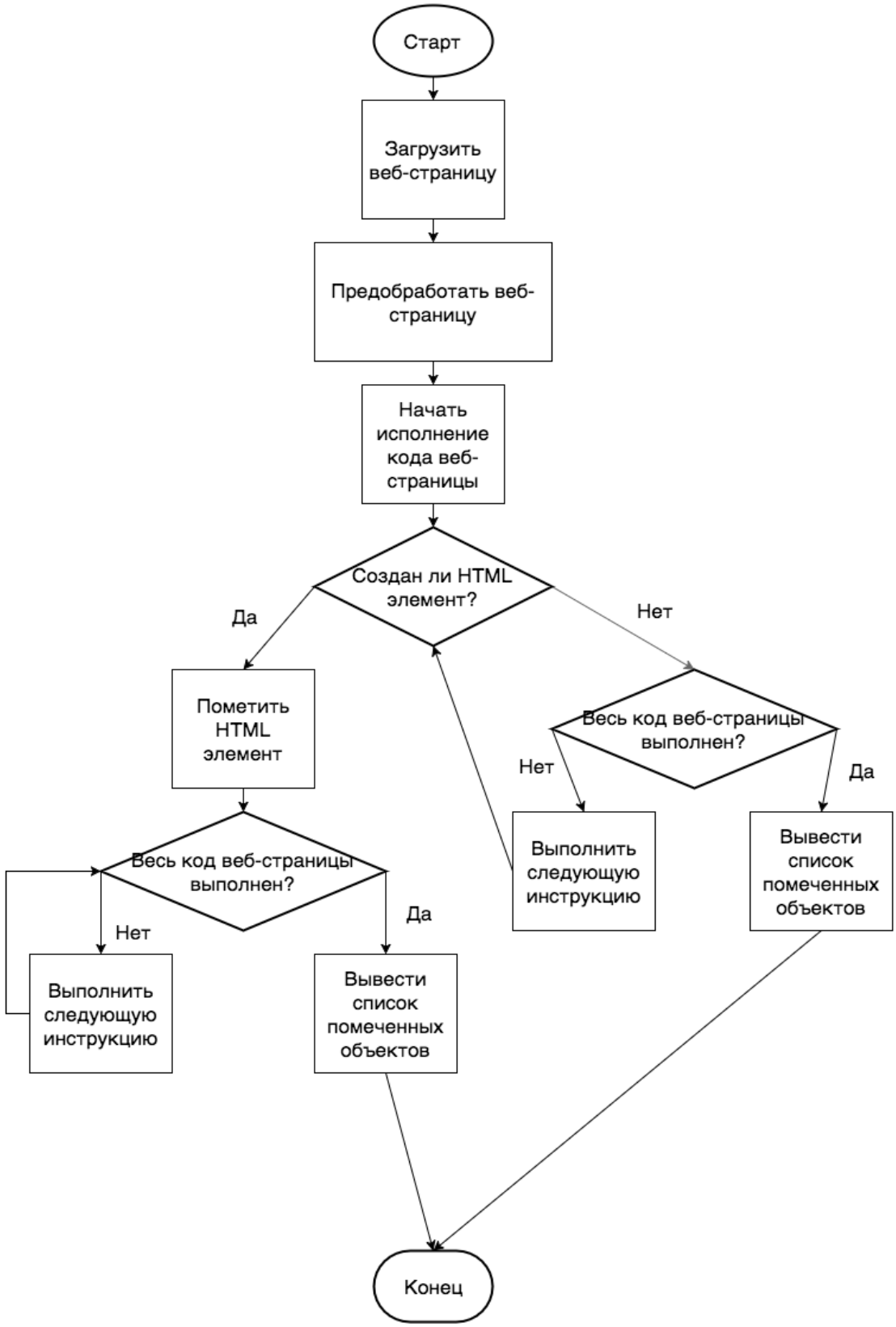
5 Описание практической части

В результате рассуждений, приведенных в предыдущей главе, были приняты следующие решения касательно программной реализации (далее реализации):

- Реализация будет представлять собой дополнение к браузеру Mozilla Firefox, написанное на языке JavaScript.
- В качестве промежуточного звена между браузером и реализацией будет выступать средство SlimerJS.
- Анализ кода будет осуществляться с применением метода тейнт-анализа.

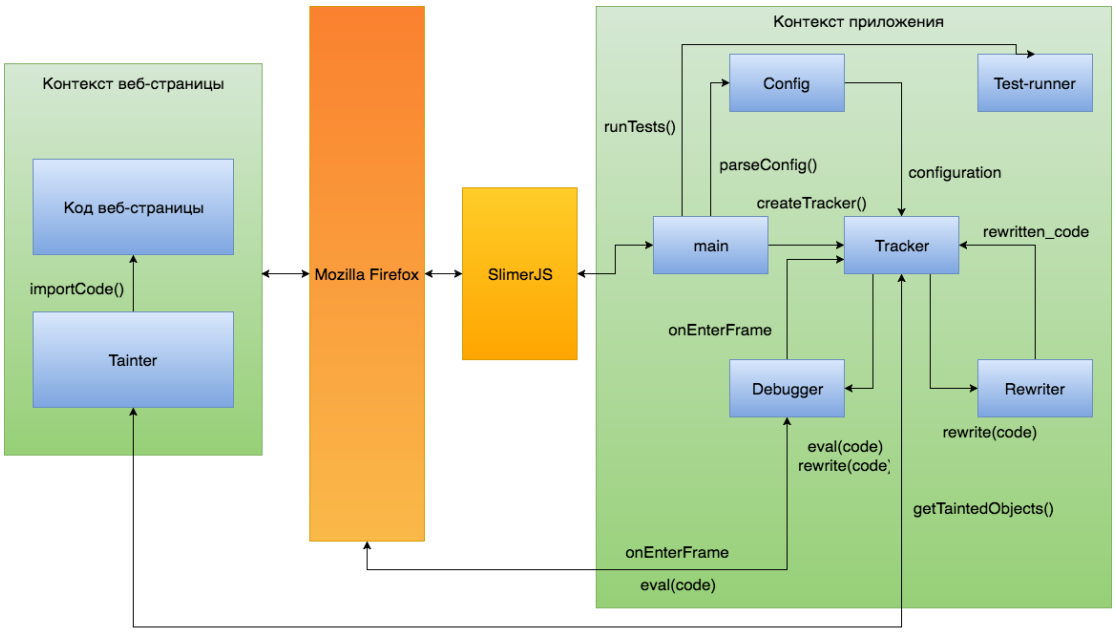
5.1. Алгоритм работы

Не вдаваясь в детали, алгоритм работы реализации можно проиллюстрировать следующей блок-схемой:


Старт
Загрузить веб-страницу
Предобработать веб-страницу
Начать исполнение кода веб-страницы
Создан ли HTML элемент?
Да
Нет
Пометить HTML элемент
Весь код веб-страницы выполнен?
Нет
Да
Выполнить следующую инструкцию
Вывести список помеченных объектов
Весь код веб-страницы выполнен?
Нет
Да
Выполнить следующую инструкцию
Вывести список помеченных объектов
Конец

5.2. Архитектура программной реализации

Архитектура программной реализации представлена на рисунке ниже. Краткое описание каждого из модулей будет дано далее.



В предлагаемой реализации существуют два контекста выполнения JavaScript кода: *контекст веб-страницы* и *контекст приложения*. Связующими звеньями между двумя контекстами являются веб-браузер и SlimerJS.

5.3. Контекст веб-страницы

Контекст веб-страницы содержит код веб-страницы (возможно измененный) и дополнительные модули, которые в него встраиваются. Весь его код исполняется в интерпретаторе браузера.

Модуль Tainter

В текущей реализации в код веб-страницы встраивается только один модуль, а именно модуль *Tainter*. Его задача заключается в нанесении меток на объекты и в отслеживании помеченных объектов. Для этого он предоставляет интерфейс, содержащий метод *taint(object)*, который и помечает объект. Авторами были созданы две реализации данного метода, которые будут подробно рассмотрены ниже в секции Реализация тейнтирования.

5.4. Веб-браузер

В предлагаемой реализации веб-браузер решает следующие задачи:

1. Загрузка и обработка веб-страницы.
2. Создание контекста веб-страницы.

3. Исполнение кода веб-страницы.

4. Предоставление средств отладки исполняемого кода контексту приложения (о средствах отладки будет подробнее рассказано в секции Модуль Debugger).

5.5. SlimerJS

SlimerJS является связующим звеном между веб-браузером и контекстом приложения. Он предоставляет удобный интерфейс для передачи запросов от контекста приложения (запрос на загрузку веб-страницы) к веб-браузеру и для информирования контекста приложения о событиях, произошедших в браузере (например, завершение загрузки веб-страницы).

5.6. Контекст приложения

Контекст приложения содержит модули, выполняющие основную работу по анализу кода. Кратко опишем работу главных из них.

Модуль Tracker

Модуль *Tracker* является главным координационным модулем всего приложения. Он отвечает за следующие действия:

1. Инициализация контекста веб-страницы.

2. Отслеживание создания HTML элемента.

3. Выполнение отладочного кода в контексте веб-страницы.

4. Вывод полученных результатов.

Модуль Debugger

Модуль *Debugger* использует интерфейс Debugger-API, предоставляемый браузером Mozilla Firefox. Его основная задача заключается в предоставлении средств отладки кода веб-страницы.

В предлагаемой реализации данный интерфейс используется для двух целей:

1. Модификация кода веб-страницы.

2. Выполнение отладочного кода в контексте веб-страницы.

Модуль Rewriter

Как будет показано ниже, для достижения некоторых целей приходится динамически переписывать код веб-страницы перед его выполнением. Для этого используется модуль *Rewriter*. О целях и стратегии переписывания операторов языка JavaScript будет рассказано далее.

5.7. Реализация тейнтирования

Как уже говорилось выше, в качестве метода анализа кода был выбран метод тейнтирования. В рамках рассматриваемой задачи единственным истоком является контролируемый HTML элемент. Стоками же являются все объекты, которые были помечены во время выполнения кода.

Рассмотрим список операций, которые распространяют пометки, то есть операций, результаты которых всегда помечаются:

1. Присваивание с помеченной правой частью:

```
1       var x = taint('foo')
```

2. Доступ к свойству помеченного объекта:

```
1       var tainted_obj = {'foo': 'bar'};
2       var x = tainted_obj.foo;
```

3. Вызов функций с помеченными аргументами:

```
1       var x = func(taint('foo'), 'bar');
```

4. Вызов помеченных функций от любых аргументов:

```
1       taint(func);
2       var x = func('foo', 'bar');
```

5. Результат выражений с участием помеченных объектов:

```
1       var x = taint('foo') + 'bar';
```

Авторами были написаны две реализации метода taint(), о которых и пойдет речь далее.

Тейнтирование с помощью proxy объектов

Первая реализация опиралась на использование так называемых прокси-объектов (proxy). Прокси - это объекты, являющиеся ёберткаминäд другими объектами (мы будем называть их внутренними). Задача прокси заключается в перехвате различных событий, связанных с внутренними объектами. В качестве реализации прокси-объектов авторами использовался глобальный объект Proxy из стандарта языка JavaScript ECMAScript 6.

Из всего множества событий, перехватываемых Proxy, авторам пригодились следующие:

- get: событие для операций *tainted.x* и *tainted['x']*
- set: событие для операций *tainted.x = y* и *tainted['x'] = y*
- apply: событие для операции *tainted(arguments)*

Суть метода *taint(object)* заключалась в замене объектов, которые нужно пометить, на прокси, обернутые вокруг них. Например, когда нужно было вернуть значение некоторого свойства помеченного объекта, возвращалось не само значение, а прокси. С помощью такого подхода легко решались проблемы, связанные с распространением пометок в следующих операциях:

- Присваивание с помеченной правой частью
- Доступ к свойству помеченного объекта
- Вызов функций с помеченными аргументами
- Вызов помеченных функций от любых аргументов

К сожалению, такой подход не даёт возможности перехватывать операции последнего типа (с участием помеченных объектов), поэтому эту задачу пришлось решать с помощью переписывания операторов.

В процессе разработки, авторы столкнулись и с другими проблемами, связанными с использованием Proxy:

- Proxy объект нельзя обернуть вокруг примитивных значений (string, number, boolean)
- Proxy объект обязан возвращать то же значение для неконфигурируемых и защищенных от записи свойств (non-writable non-configurable property)
- Некоторые объекты из DOM (например, *document.body*) нельзя заменить на Proxy.

Первые два ограничения не являются критическими и для них были найдены решения. Третье же показало, что для некоторых имен HTML элементов программная реализация не соответствует предложенной методике. Рассмотрим данную проблему на примере:

```
1    <form name='querySelector'> </form>
```

```
1    <script>
2      document.querySelector = new Proxy(document.querySelector);
3    </script>
```

```
1    <form name='body'> </form>
```

```
1    <script>
2      document.body = new Proxy(document.body);
3    </script>
```

Результат выполнения кода из первых двух листингов эквивалентен - функция *document.querySelector* заменится другим объектом. Код из третьего листинга также заменит объект *document.body*, а вот в работе кода из четвертого листинга произойдет ошибка.

Данная проблема ведет к тому, что предложенный алгоритм не будет работать для некоторых имен (например, document.body). Поэтому, авторами были принято решение о поиске другой реализации метода taint().

Тейнтирование на основе переписывания операторов

Второй возможной стратегией тейнтирования является нанесение пометок на сами объекты в виде некоторого специального свойства (*object.is_tainted = true*). Такой подход хорош тем, что он лишен проблем, связанных с использованием Proxy. Однако, он ставит перед разработчиками новую задачу, а именно, реализацию распространения пометок через операции.

На момент начала работ над вторым способом тейнтирования авторами уже был написан механизм переписывания кода, необходимый для замены стандартных операторов языка JavaScript. Было решено использовать этот механизм для переписывания операций на эквивалентные, но с возможностью распространения пометок.

Рассмотрим еще раз список операций, распространяющих пометки, и приведем примеры их переписываний:

1. Присваивание с помеченной правой частью:

```
var taint = function(object) {
  ...
  object.is_tainted = true;
  ...
}
var x = taint('foo');
```

2. Доступ к свойству помеченного объекта:

```
var tainted_obj = {'foo': 'bar'};
var x = __get__(tainted_obj, 'foo'); // вместо var x =
    tainted_obj.foo;
```

3. Вызов функций с помеченными аргументами:

```
var x = __call__(func, [taint('foo'), 'bar']); // вместо
    var x = func(taint('foo'), 'bar');
```

4. Вызов помеченных функций от любых аргументов:

```
taint(func);
var x = __call__(func, ['foo', 'bar']); // вместо var x =
     func('foo', 'bar');
```

5. Результат выражений с участием помеченных объектов:

```
var x = __add__(taint('foo'), 'bar'); // вместо var x =
    taint('foo') + 'bar';
```

С помощью данного подхода мы получили решение эквивалентное предложенной методике. Расплатой за это явилось увеличение времени выполнения и размера кода веб-страницы.

5.8. Реализация переписывания кода

Как было показано выше, для достижения поставленных целей необходимо добавление к реализации функционала, способного решать следующие задачи:

- Перегрузка операторов (+, -, &&, и т. п.)
- Перегрузка оператора typeof
- Создание геттеров (аксессоров) и сеттеров (мутаторов) для некоторых объектов
- Перехват вызовов функций

Одним из возможных решений является переписывание кода. Его суть заключается в замене кода, содержащегося в веб-странице, эквивалентным с точки зрения выполняемой программы кодом, который осуществляет некоторые дополнительные действия.

Приведем наглядный пример таких кодов:

```
var x = tainted('foo') + 'bar';
```

```
function __add__(left, right) {
  var result = left + right;
  if (is_tainted(left) || is_tainted(right)) {
    taint(result);
  }
  return result;
}
var x = tainted('foo') + 'bar';
```

Два приведенных кода является эквивалентными в том смысле, что в переменную *x* запишется результат конкатенации строк *'foo'* и *'bar'*, но в результате второго из них переменная *x* будет помечена.

Аппарат переписывания кода, реализованный авторами, работает следующим образом:

1. Построить по данному коду его представление в виде AST (Абстрактного Синтаксического Дерева).
2. Обойти дерево в глубину, преобразуя необходимые узлы.
3. По полученному AST сгенерировать новый код.

Рассмотрим работу приведенного алгоритма на примере:

```
var a = 'foo' + 'bar';
```

По данному коду будет построено следующее AST:

```
{
  "type": "Program",
  "body": [
      {
          "type": "VariableDeclaration",
          "declarations": [
              {
                  "type": "VariableDeclarator",
                  "id": {
                      "type": "Identifier",
                      "name": "x"
                  },
                  "init": {
                      "type": "BinaryExpression",
                      "operator": "+",
                      "left": {
                          "type": "Literal",
                          "value": "foo",
                          "raw": "'foo'"
                      },
                      "right": {
                          "type": "Literal",
                          "value": "bar",
```

```
                            "raw": "'bar'"
                        }
                    }
                }
            ],
            "kind": "var"
        }
    ],
    "sourceType": "script"
}
```

После преобразований будет получено следующее AST:

```
{
    "type": "Program",
    "body": [
        {
            "type": "VariableDeclaration",
            "declarations": [
                {
                    "type": "VariableDeclarator",
                    "id": {
                        "type": "Identifier",
                        "name": "x"
                    },
                    "init": {
                        "type": "CallExpression",
                        "callee": {
                            "type": "Identifier",
                            "name": "__add__"
                        },
                        "arguments": [
                            {
                                "type": "Literal",
                                "value": "foo",
                                "raw": "'foo'"
                            },
                            {
                                "type": "Literal",
                                "value": "bar",
                                "raw": "'bar'"
                            }
                        ]
                    }
                }
            ],
            "kind": "var"
        }
    ],
    "sourceType": "script"
}
```

По полученному AST будет сгенерирован следующий код:

```
var x = __add__('foo', 'bar');
```

Реализация аппарата использует следующие библиотеки:

- Построение AST - esprima
- Обход AST - estraverse
- Генерация кода - escodegen

5.9. Тестирование реализации

Тестирование реализации осуществлялось тремя способами:

Модульное тестирование

Модульное тестирование позволяет проверить корректность работы отдельных модулей программы. Оно также дает возможность убедиться в том, что код продолжает быть корректным после внесения в него изменений, что существенно сокращает количество времени, необходимого на разработку.

В предложенной реализации с помощью модульного тестирования проверялась работа следующих компонент:

- *Модуль Tainter* (проверялась корректность распространения пометок операциями)
- *Модуль Rewriter* (проверялась корректность переписывания кода)

Синтетические веб-страницы

Под синтетическими веб-страницами понимаются страницы, удовлетворяющие следующим критериям (например, набор тестовых страниц damnvulnerable.me):

- В них присутствует недостаток DOM Clobbering
- Их код достаточно мал по объему, чтобы его можно было проанализировать вручную
- Для них известны способы и результаты эксплуатации

В процессе тестирования проверялось соответствие результатов, полученных программой, и полученных вручную.

Реальные веб-сайты

Для проверки работы реализации в реальных условиях использовались веб-сайты, содержащие большое количество JavaScript-кода (например, twitter.com, reme.io, phpregex.appspot.com). В случае отсутствия в странице недостатка DOM Clobbering, создавалась её копия с внедренным недостатком. В процессе тестирования проверялось соответствие результатов, полученных программой, и полученных вручную.

6 Заключение

В рамках данной дипломной работы были достигнуты следующие результаты:

1. Проведено исследование способов эксплуатации недостатков веб-приложений, связанных с возможностью изменения области видимости переменных объектной модели веб-страницы, включающее в себя:
   - Определение причин возникновения недостатка
   - Определение способов эксплуатации недостатка
   - Определение возможных последствий эксплуатации
2. Разработана методика определения возможности и последствий подмены объектов DOM для заданной веб-страницы, основывающаяся на анализе зависимостей по данным.
3. Проведено исследование способов проведения автоматического анализа JavaScript-кода.
4. Проведено исследование применимости методов анализа JavaScript-кода.
5. Созданы две программные реализации предложенной методики:
   - С использованием прокси-объектов
   - С переписыванием кода
6. Проведено тестирование реализации.

Результаты работы программной реализации доказывают эффективность предложенной методики. Поставленная задача выполнена полностью.