



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

Оглавление	1
1 Введение	2
2 Постановка задачи	4
2.1 Неформальная постановка задачи	4
2.2 Формальное определение DOM Clobbering	6
2.3 Постановка задачи	6
3 Анализ задачи	7
3.1 Исследование возможных направлений атаки	7
Вызов ошибки в программе	7
Обход критических состояний	7
Выполнение произвольного кода	8
3.2 Требования к решению	8
3.3 Методика решения задачи	8
3.4 Доказательство корректности методики	9

Введение

В период зарождения всемирной паутины веб-страницы представляли собой статические документы, сверстанные на языке HTML. Но, с течением времени, страницы становились все более и более динамическими. Современные 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, AJAX и т. д.)

Активное использование языка JavaScript привело к появлению новой парадигмы создания 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 она 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 её недостаточной изученности. Однако, высокая степень опасности, которую таят в себе недостатки данного типа, указывает на необходимость разработки автоматизированного средства их поиска.

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; // указывает на <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 различными 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 ``, `<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 пару типичных примеров недостатка DOM Clobbering.

Пример 2:

```

1 <form name="querySelector">
2 <script>
3   var element = document.querySelector(""); // ошибка: document.
   querySelector указывает на <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:

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

```
1 <a href="plugins/preview/preview.html#<svg onload=alert(1)>" id="_cke_htmlToLoad" target="_blank">
2   Click me!
3 </a>
4
5 файл /plugins/preview/preview.html:
6 <script>
7   ...
8   document.write(window.opener._cke_htmlToLoad);
9   ...
10 </script>
```

Данный пример взят из работы 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 отработывал 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. Формальное определение DOM Clobbering

Дадим формальное 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. Постановка задачи

Основная задача данной работы - сформулировать методику и разработать инструментальное средство для определения возможности (и последствий) подмены объектов DOM для заданной веб-страницы.

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 $\langle i, Vars, Values \rangle$, где i - точка программы, $Vars$ - множество переменных, доступных в i , $Values$ - множество двоек $\langle var, value \rangle$, 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 $\langle i, Vars, Values \rangle$, в котором программа может оказаться.

Например, состояние $\langle 4, is_in_black_list, \langle is_in_black_list, true \rangle \rangle$ является критическим, так как для того, чтобы программа перешла в него, необходимо присутствие пользователя в черном списке.

С другой стороны, состояние $\langle 4, is_in_black_list, \langle is_in_black_list, HTMLForm \rangle \rangle$ 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 нарушителем удастся перевести программу в не критическое 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 сформулированы следующие требования к алгоритму анализа веб-приложений.

1. Алгоритм должен уметь детектировать возможность возникновения ошибок из-за переименования HTML элемента и места таких ошибок.
2. Алгоритм должен уметь детектировать возможность и пути обхода критических состояний.
3. Алгоритм должен уметь детектировать возможность выполнения произвольного кода.

3.3. Методика решения задачи

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

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

- Оператор О использует значение А.
- Оператор О зависит по управлению от оператора, который зависит от А.

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

- Оператор, вычисляющий А, зависит от В.
- Оператор, вычисляющий А, зависит от переменной, которая зависит от В.

Приведем пару примеров:

```
1 a = "a";
2 b = a + "b"; // b зависит от a
3 c = b + "c"; // c зависит от b, а значит и от a
4 if (a === "a") {
5     d = "d"; // d зависит от a
6 }
7 else {
8     e = "e"; // e зависит от a
9 }
10 func(a); // оператор вызова функции зависит от 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, зависящих по данным от рассматриваемого HTML элемента.
3. *Этап обработки результатов анализа* - по полученному списку зависящих переменных и операторов вычислить значение некоторых предикатов.
4. *Этап вывода* - результатом является список всех зависящих переменных и вычисленные значения предикатов.

3.4. Доказательство 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 функций) 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.