Г. М. Носеевич, А. А. Петухов

ПОИСК ВХОДНЫХ ТОЧЕК ДЛЯ ВЕБ-ПРИЛОЖЕНИЙ С ДИНАМИЧЕСКИМ ПОЛЬЗОВАТЕЛЬСКИМ ИНТЕРФЕЙСОМ

Введение

Первым шагом при исследовании веб-приложений методом «черного ящика» является сбор информации. На этом этапе необходимо определить **входные точки** — HTTP-запросы (URL, имена параметров и их возможные значения), при помощи которых происходит доступ к данным и функциям веб-приложения. В дальнейшем собранная информация может использоваться в различных целях: для поиска уязвимостей, связанных с некорректной обработкой входных данных[1], для проведения автоматизированного поиска уязвимостей авторизации [2] и т. д. При этом от полноты построенного множества точек входа зависит полнота производимого анализа.

При работе с веб-приложением пользователи активируют его входные точки (т. е. совершают HTTP-запросы), используя веб-интерфейс приложения (кликая по ссылкам, отправляя веб-формы и т. п.). Таким образом, пользовательский интерфейс веб-приложения содержит всю необходимую информацию о его возможных входных точках. Следовательно, путем построения множества всевозможных HTTP-запросов, которые могут быть совершены в процессе взаимодействия пользователя с каждой из страниц веб-приложения, можно получить полное множество его входных точек.

В классических веб-приложениях пользователи взаимодействуют с веб-страницами при помощи перехода по ссылкам и отправки веб-форм, причем после загрузки веб-страницы набор ссылок и форм, которые пользователь может активировать, остается неизменным до перехода на другую страницу. Такой пользовательский интерфейс называется *статическим*. Для веб-приложений со статическим интерфейсом решение описанной задачи сводится к разбору HTML-документа и выделению в нем ссылок и веб-форм.

Во многих современных веб-приложениях пользовательский интерфейс построен с использованием клиентского javascript и является динамическим. Использование javascript-сценариев позволяет произвольным образом реагировать на события, происходящие на странице. В связи с этим любое действие или последовательность действий пользователя может привести к выполнению javascript-кода, который, в свою очередь, может отправлять HTTP-запросы и получать и обрабатывать HTTP-ответы. При этом установка обработчиков событий может производиться как в момент начальной загрузки страницы, так и в процессе работы с ней, а содержимое выполняемых запросов может вырабатываться на ходу, для чего предусмотрен интерфейс XMLHttpRequest. При этом выделение ссылок и форм в HTML-коде страницы не позволяет получить сколько-нибудь полное множество возможных HTTP-запросов.

В результате задача обнаружения точек входа сводится к анализу программ на языке javascript и в общем случае не может быть полностью решена: статический анализ в общем случае неразрешим, а динамический анализ невозможно провести для всех возможных путей выполнения программы. В связи с этим востребованным становится приближенный метод построения множества возможных HTTP-запросов, позволяющий проводить анализ страниц широкого класса современных веб-приложений.

В данной статье предлагается алгоритм построения множества возможных запросов для страниц веб-приложений с динамическим пользовательским интерфейсом на основе комбинации статического и динамического анализа кода страницы. Построенный метод учитывает особенности самого

[1] Такой подход, получивший название «фаззинг» [1], часто используется для поиска уязвимостей, связанных с некорректной обработкой пользовательского ввода (в том числе SQL-injection и XSS).

распространенного на сегодняшний день каркаса для построения динамического пользовательского интерфейса веб-приложений — jQuery [3].

1. Особенности веб-приложений с динамическим интерфейсом

В классических веб-приложениях совершение HTTP-запроса при взаимодействии пользователя с веб-приложениями может произойти в двух случаях: при активации ссылки (HTML anchor) и при отсылке HTML-формы. Совершаемые при этом HTTP-запросы определяются для ссылок атрибутом href, а для форм — атрибутом action и значениями полей формы. В этом случае искомое множество входных точек на некоторой странице может быть построено путем выделения на ней всех ссылок и форм.

В современных веб-приложениях для придания страницам интерактивности и улучшения пользовательского интерфейса активно используется клиентский javascript. При этом javascript-сценарии взаимодействуют с браузером и веб-страницей, в которой они загружены, с использованием интерфейсов Browser Object Model (BOM) и Document Object Model (DOM). Интерфейс DOM позволяет устанавливать для элементов веб-страницы функции-обработчики событий, что позволяет определять реакцию страницы на те или иные действия пользователя. Например, используя обработчик onclick, можно установить реакцию на событие щелчка мыши.

Функция-обработчик является произвольной javascript-функцией, что позволяет осуществлять сколь угодно сложную реакцию на действия пользователя. Кроме того, интерфейс XMLHttpRequest позволяет совершать из javascript-обработчиков произвольные HTTP-запросы и динамически встраивать их результаты в веб-страницу[2]. Это приводит к возможности построения асинхронных веб-интерфейсов, в которых пользователь может продолжить просмотр веб-страницы, пока происходят отправка HTTP-запроса и получение ответа на него. Данная технология, получившая название AJAX (Asynchronous JAvascript and XML [4]), является одним из ключевых технических средств для построения современных веб-приложений.

Таким образом, на страницах веб-приложений с динамическим интерфейсом произвольное действие или даже произвольная последовательность действий пользователя может привести к совершению HTTP-запроса, причем содержимое этого запроса также вырабатывается динамически. Кроме того, действия, совершаемые обработчиками, могут зависеть от состояния глобального контекста выполнения: текущих значений глобальных переменных, содержимого DOM-дерева и т. д. Другие обработчики, в свою очередь, могут модифицировать глобальный контекст.

Рассмотрим механизм диспетчеризации событий, используемый в DOM (более подробно см. [5]). Все элементы веб-страницы организованы в виде дерева (DOM Tree), корнем которого является элемент Document. Последовательность элементов вдоль пути от корня дерева к узлу, в котором произошло событие, называется *путем распространения события* (event propagation path).

При возникновении события в некотором узле DOM-дерева (данный узел будем называть *целевым*) порождается объект события, который проходит через три *фазы обработки* (см. рис. 1):

1. В процессе **фазы захвата** объект передается **перехватывающим** обработчикам соответствующего события для узлов, входящих в путь распространения, начиная от корня.

2. Во время **целевой фазы** вызываются обработчики для целевого узла.

3. Во время **фазы всплытия** вызываются обычные (не перехватывающие) обработчики для узлов вдоль пути распространения, от целевого элемента к корню. Данная фаза выполняется не для всех типов событий.

[2] При этом действует так называемое «правило ограничения домена» — Same Origin Policy, которое предотвращает для сценариев, загруженных с некоторого домена, доступ к данным, находящимся на других доменах.

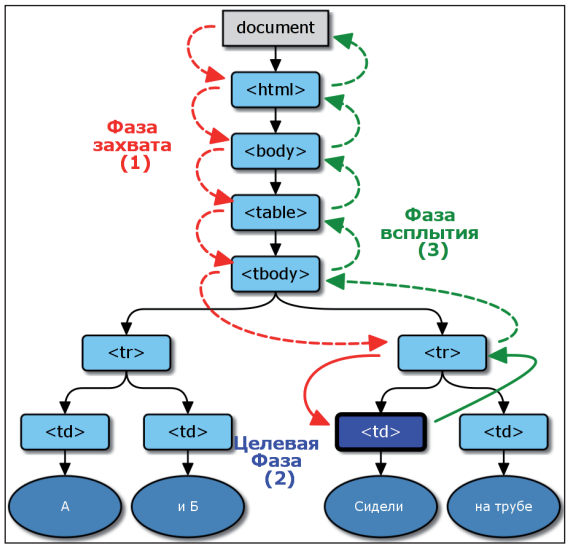

Рис. 1. Процесс диспетчеризации события в HTML-документе

Заметим, что любой из обработчиков может немедленно завершить обработку события, вызвав метод Event.stopPropagation для объекта события.

У некоторых типов событий может быть определено *действие по умолчанию*. В частности, для события клика по ссылке действием по умолчанию является переход на новую веб-страницу. При этом для некоторых типов событий существует возможность предотвратить совершение действия по умолчанию при помощи метода Event.preventDefault.

Установка обработчиков событий происходит с использованием DOM-интерфейса EventTarget.addEventListener, которому в качестве параметров передается тип обрабатываемого события, функция-обработчик, а также параметр, который определяет, является ли устанавливаемый обработчик перехватывающим.

Как можно видеть, в веб-приложениях, которые активно используют клиентский javascript, перечень ссылок и форм, во-первых, не исчерпывает множество возможных HTTP-запросов, а во-вторых, отдельным ссылкам могут соответствовать запросы, которые в реальности не могут быть выполнены из-за того, что обработчик события предотвращает действие по умолчанию.

Как было описано выше, задача построения для заданной страницы множества возможных HTTP-запросов в общем случае сводится к анализу сценариев на языке javascript. В связи с этим проведение точного автоматического анализа невозможно, и задача сводится к поиску эвристического алгоритма, который позволяет с приемлемой точностью строить искомое множество для широкого класса веб-страниц. Возможны два подхода к анализу веб-страниц — статический и динамический. В рамках первого подхода текст HTML-страницы и включенного в нее javascript-кода подвергается анализу без исполнения сценариев, тогда как при динамическом анализе происходит полная загрузка страницы и исполнение сценариев, в процессе которого анализируется контекст выполнения (значения полей объектов и т. п.).

2. Обзор инструментов для статического анализа javascript

Целями обзора являются:

- исследование функциональных возможностей статических анализаторов javascript применительно к анализу веб-страниц, содержащих смесь HTML и javascript-кода;
- определение возможности применения статического анализа к задаче построения множества HTTP-запросов, которые могут быть совершены в процессе взаимодействия пользователя с данной веб-страницей.

В качестве критериев обзора были выделены следующие характеристики инструментов статического анализа:

- общие функциональные возможности: построение синтаксического дерева (AST) и графа потока управления (CFG), анализ типов, анализ указателей, построение срезов;
- специальные возможности по анализу javascript-кода веб-страниц: поддержка интерфейса Document Object Model и возможность анализа смешанного HTML+javascript кода;
- наличие инструмента в свободном доступе, возможность интеграции с ним.

2.1. Инструменты для статического анализа javascript

Средства анализа качества кода: JSLint, JSure, javascript Lint

Данные средства направлены на оценку качества кода на языке javascript. Оценка качества происходит путем проверки кода на соответствие набору простых правил, например:

- запрет на разрыв строки посередине оператора в тех случаях, когда это может привести к двусмысленности;
- запрет на использование «опасных» возможностей языка javascript (eval, new Function() и т. п.);
- обнаружение бесполезного и недоступного кода (в простейших случаях).

Данные средства решают узкоспециализированную задачу оценки качества кода и непригодны в как средства статического анализа веб-страниц.

JSAnalyse и Jint

Средство JSAnalyse предназначено для поиска статических зависимостей между файлами с исходными текстами на языке javascript. Инструмент реализован на языке C# на основе Jint, интерпретатора языка javascript для платформы .NET. Для поиска зависимостей между файлами для каждого из них строится список декларируемых имен, а также производится поиск вызовов функций с именами, которые не задекларированы в том же файле. В случае, если имя незадекларированной функции в одном файле совпадает с именем, задекларированным в другом файле, делается вывод о наличии зависимости.

Из общих функциональных возможностей данные средства поддерживают только построение синтаксического дерева. Анализ HTML-страниц с javascript-кодом невозможен.

JSHydra

Данное средство является анализатором javascript-кода, построенным на основе javascript-интерпретатора SpiderMonkey, который также используется в веб-браузере Firefox. Средство поддерживает только построение синтаксического дерева и произведение трансформаций над ним. Анализ HTML-страниц с javascript-кодом невозможен.

T.J. Watson Libraries for Analysis (WALA)

WALA — это средство статического анализа кода на языках Java, javascript, PHP, ABAP, а также Java-байткода и .NET-байткода. WALA поддерживает построение синтаксического дерева, графа потока управления, содержит реализацию нескольких алгоритмов анализа типов, анализа указателей, построения срезов и т. п.

Все реализованные в WALA алгоритмы статического анализа оперируют над промежуточным представлением (Intermediate representation, IR) программ. При анализе байткода происходит непосредственная трансляция из байткода в IR, а при анализе исходных кодов — предварительная генерация не зависящего от языка абстрактного синтаксического дерева (Common AST, CAst). Такой подход позволяет использовать одни и те же алгоритмы для различных языков. С точки зрения особенностей браузерного javascript WALA имеет поддержку Document Object Model и позволяет анализировать HTML-страницы с javascript путем их предварительной конвертации в чистый javascript.

К сожалению, проведенное экспериментальное исследование возможностей текущей версии WALA для анализа кода сложных веб-страниц показало недостаточную производительность

средства. Так, попытка построения графа вызовов для веб-страницы, содержащей библиотеку jQuery, не завершилась в течение нескольких часов.

2.2. Вывод

Проведенный обзор показывает, что на данный момент единственная функциональная возможность статических анализаторов javascript-кода, которую можно использовать при анализе веб-страниц реальных веб-приложений, — это построение и анализ синтаксического дерева.

3. Исследование возможности применения динамического анализа

Как показал обзор статических анализаторов javascript-кода, чисто статический подход для анализа сложных страниц оказывается неприменим. В связи с этим необходимо рассмотреть задачу в терминах динамического анализа. С точки зрения динамического анализа для решения поставленной задачи необходимо решить, как минимум, следующие подзадачи:

1. определить обработчики, установленные на странице после ее окончательной загрузки;
2. выяснить, какие действия пользователей приведут к вызову этих обработчиков и потенциально к совершению HTTP-запросов;
3. определить содержимое возникающих HTTP-запросов;
4. учесть возможность динамического изменения веб-страницы, зависимость результатов обработки от глобального контекста и возможность изменения глобального контекста в процессе обработки события.

Использование динамического подхода для анализа javascript-кода веб-страниц изучено слабо. Существующие работы, в основном, рассматривают применение динамического taint-анализа для предотвращения атаки межсайтового выполнения сценариев (см., например: [6]). Также динамический анализ используется для выделения характеристик javascript-кода в работе [7], рассматривающей задачу обнаружения вредоносных веб-страниц. Существующие подходы, таким образом, оказываются не применимыми для поставленной задачи.

Будем предполагать, что загрузка веб-страниц и исполнение javascript-сценариев происходят в контролируемой среде (в качестве такой среды можно использовать библиотеку HtmlUnit, аналогично работе [7]). Такой подход позволяет определять, какие обработчики установлены для рассматриваемой страницы в текущий момент времени.

Для определения факта совершения HTTP-запроса и получения его содержимого будет применен метод перехвата совершаемых HTTP-запросов (такая возможность реализована в используемой среде HtmlUnit).

В качестве базового алгоритма построения искомого множества возможных HTTP-запросов рассмотрим перебор всех элементов страницы, имеющих обработчики. Для всех таких элементов необходимо инициировать соответствующие события и определять, произошел ли HTTP-запрос и каково его содержимое. Этот алгоритм гарантирует, по крайней мере, однократное исполнение всех установленных на странице обработчиков.

У такого алгоритма, однако, есть несколько недостатков. С одной стороны, порядок посещения элементов может оказать влияние на результат, так как при выполнении обработчиков может произойти удаление некоторых элементов, обработчики для которых, таким образом, никогда не будут вызваны.

С другой стороны, выполнение каждого обработчика происходит только по одному из возможных путей, в связи с чем некоторые из возможных HTTP-запросов могут быть пропущены. В частности, в обработчике может происходить ветвление в зависимости от первоначальной цели события[3], что позволяет в одном и том же обработчике по-разному обрабатывать

[3] Напомним, что возникновение события в некотором элементе DOM-дерева может привести к исполнению обработчиков не только для данного элемента, но и для других узлов на пути от элемента к корню дерева. При этом объект события, передаваемый обработчику, содержит информацию о первоначальной цели — элементе, для которого было инициировано событие.

события, произошедшие в разных дочерних узлах. Такой подход называется *делегированной* обработкой.

Несмотря на некоторую неочевидность, эта техника применяется во множестве веб-страниц либо непосредственно, либо с использованием библиотек. Это связано с тем, что такой подход позволяет устанавливать обработчики даже для тех элементов, которые еще не были добавлены на веб-страницу, что существенно ускоряет общее время загрузки страницы: уже загруженные javascript-сценарии могут исполняться параллельно с загрузкой элементов страницы (например, изображений).

С учетом данной особенности однократное исполнение каждого обработчика не является достаточным, и в общем случае требуется, как минимум, для каждого установленного обработчика перебирать все возможные целевые элементы. При этом возрастает количество элементов веб-страницы, которые надо протестировать (характерным использованием делегированной обработки является установка обработчиков на корневой элемент страницы, что приводит к необходимости перебирать все элементы страницы для каждого такого обработчика), что значительно увеличивает время анализа.

Для преодоления описанной проблемы было решено учесть особенности библиотеки jQuery [3], что позволило уменьшить количество тестируемых элементов веб-страницы.

Библиотека jQuery содержит интерфейс jQuery.on, реализующий делегацию обработки событий. В данный метод помимо функции-обработчика передается строка-*селектор*[4], характеризующая элементы, для которых должен вызываться обработчик.

При этом вместо пользовательского обработчика на страницу устанавливается функция, которая осуществляет вызов необходимых обработчиков, проверяя соответствие реальной цели события набору селекторов. Сами пользовательские обработчики и соответствующие им селекторы хранятся в служебных структурах данных jQuery.

Описанная особенность позволяет исключить из перебора те элементы веб-страницы, которым не соответствует ни один из селекторов обработчиков jQuery. Для этого необходимо во время исполнения получить список селекторов и подвергать проверке только те элементы страницы, которые соответствуют хотя бы одному из селекторов.

Во многих веб-страницах, однако, обработчики событий устанавливаются как с использованием jQuery, так и напрямую. Описанную выше оптимизацию можно применить только для обработчиков, установленных с использованием jQuery, а в остальных случаях необходимо по-прежнему осуществлять полный перебор потомков.

Таким образом, необходимо иметь возможность определить, является ли данный обработчик стандартным обработчиком, установленным библиотекой jQuery.

Для этого можно использовать возможность на этапе выполнения получить исходный код любой javascript-функции при помощи метода toString. При сравнении функции-обработчика с шаблоном нельзя просто проверять тексты на совпадение, так как повсеместно используемые минификаторы javascript-кода[5] могут генерировать различающийся текст для одних и тех же исходных данных. Поэтому для полученного фрагмента исходного кода строится синтаксическое дерево, которое сравнивается с шаблонным синтаксическим деревом для обработчика jQuery. При получении совпадения с точностью до имен переменных делается вывод о том, что анализируемый обработчик является jQuery-обработчиком.

Для того чтобы изменения в странице, совершенные при выполнении одного обработчика, не влияли на последующее исполнение другого обработчика, инициация каждого события

[4] В jQuery применяется библиотека селекторов sizzlejs, позволяющая производить выборку из DOM-дерева с использованием синтаксиса, сочетающего CSS и XPath.

[5] Javascript-минификаторы предназначены для уменьшения объема javascript-кода путем удаления пробельных символов, переименования переменных и т. п.

производится в рамках независимой копии исходной страницы. Если в результате инициации события происходит добавление на страницу новых элементов, для них также проводится анализ обработчиков.

4. Предлагаемый алгоритм

На основании проведенного анализа был предложен следующий алгоритм получения множества HTTP-запросов, которые могут быть совершены при взаимодействии пользователя с данной веб-страницей (**целевое множество**):

1. Загрузить веб-страницу и исполнить включенные в нее javascript-сценарии.

2. **Составить множество тестируемых пар** (элемент, тип события)**:**

a. для каждого узла, имеющего установленный обработчик события, получить исходный текст обработчика;

b. построить синтаксическое дерево для текста обработчика и сравнить его с шаблонным обработчиком jQuery;

c. в случае совпадения получить список селекторов для пары (рассматриваемый узел, рассматриваемый тип события) из служебных структур данных jQuery и добавить во **множество тестируемых пар** те элементы-потомки рассматриваемого узла, которые соответствуют хотя бы одному из селекторов;

d. в случае отсутствия совпадения добавить во **множество тестируемых пар** узел и все его потомки.

3. Для каждой пары *(элемент, тип события)* из построенного множества:

a. сделать копию текущей веб-страницы с сохранением javascript-контекста;

b. инициировать событие для соответствующего элемента на копии страницы;

c. в случае, если событие привело к совершению запроса, **добавить тройку** *(элемент, тип события, запрос)* **в целевое множество**;

d. в случае, если событие привело к добавлению новых элементов в DOM-дерево и/или привело к установке новых обработчиков, **обновить множество тестируемых пар**.

5. Оценка полноты предложенного алгоритма

Описанный алгоритм был реализован на основе библиотеки HtmlUnit, которая, в свою очередь, применяет библиотеку Rhino для работы с javascript. Полученная реализация была протестирована на страницах распространенных веб-приложений, использующих библиотеку jQuery[6]. Было произведено сравнение полученных результатов с результатами наивного алгоритма, основанного на переборе ссылок и веб-форм. Результаты тестирования показаны в таблицах 1 и 2.

Таблица 1. Результаты работы предложенного алгоритма

	Twitter	**Google Code Search**	**Youtube**	**Reddit**
Обнаружено троек *(элемент, событие, запрос)*	49	59	680	251
Обнаружено запросов	48	43	110	178

Таблица 2. Результаты работы наивного алгоритма

	Twitter	**Google Code Search**	**Youtube**	**Reddit**
Обнаружено троек *(элемент, событие, запрос)*	45	34	174	247
Обнаружено запросов	44	34	102	175

[6] Данные страницы были выбраны исходя из перечня популярных веб-приложений, использующих jQuery [8].

Каждый из предлагаемых алгоритмов получает на вход HTML-код веб-страницы и выдает набор троек *(элемент, событие, запрос)* в качестве результата. Каждая такая тройка соответствует обнаружению HTTP-запроса, который происходит при инициации определенного события для некоторого элемента страницы. При этом произведенные HTTP-запросы для некоторых троек могут совпадать, поэтому количество обнаруженных **попарно не совпадающих** запросов показано отдельно.

Результаты эксперимента позволяют сделать следующие выводы:

- Предложенный алгоритм строит более полное множество возможных HTTP-запросов, чем наивный алгоритм, основанный на переборе ссылок и веб-форм.
- Интерфейс веб-приложений часто обладает избыточностью, часто различные действия приводят к одному и тому же HTTP-запросу. В таких случаях предложенный алгоритм обнаруживает большее количество действий для одного и того же запроса.
- Прирост в количестве обнаруженных HTTP-запросов по сравнению с наивным методом невелик (единицы процентов), однако предлагаемый алгоритм в любом случае обнаруживает все запросы, найденные с помощью наивного алгоритма.

Заключение

В данной статье предлагается метод обнаружения входных точек на страницах веб-приложения с динамическим интерфейсом. Проведенные эксперименты показывают, что предложенный метод, основанный на комбинации статического и динамического анализа кода страниц, превосходит по полноте существующий наивный метод, основанный на выделении ссылок и веб-форм.

СПИСОК ЛИТЕРАТУРЫ:

1. *Sutton M., Greene A., Amini P.* Fuzzing: brute force vulnerability discovery. Addison-Wesley Professional, 2007.
2. *Noseevich G., Petukhov A.* Detecting Insufficient Access Control in Web Applications // 1st SysSec Workshop Proceedings. Amsterdam, 2011. P. 11–18.
3. *Resig J.* jQuery: The write less, do more, javascript library. URL: http://jquery.org (дата обращения: 04.04.2013).
4. *Garrett J.* Ajax: A new approach to web applications. URL: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications (дата обращения: 04.04.2013).
5. Document Object Model (DOM) Level 3 Events Specification. URL: http://www.w3.org/TR/DOM-Level-3-Events/ (дата обращения: 04.04.2013).
6. *Vogt P., Nentwich F., Jovanovic N., Kirda E., Kruegel C., Vigna G.* Cross-site scripting prevention with dynamic data tainting and static analysis // Proceeding of the Network and Distributed System Security Symposium (NDSS), San Diego, 2007.
7. *Cova M., Kruegel C., Vigna G.* Detection and analysis of drive-by-download attacks and malicious JavaScript code // Proceedings of the 19th International Conference on World Wide Web, Raleigh, 2010.
8. Sites Using jQuery. URL: http://docs.jquery.com/Sites_Using_jQuery (дата обращения: 30.01.2013).