Поиск уязвимостей нарушения целостности динамическим анализом программ на языке Javascript

Просочкина Д. А., студент кафедры безопасных информационных технологий Университета ИТМО

Аннотация

В данной статье рассмотрен новый метод поиска уязвимостей нарушения целостности, в частности уязвимости межсайтового скриптинга, основанного на объектной модели документа, в программах на языке JavaScript. Метод основан на автоматизации алгоритма действий, выполнямых аудитором при проведении тестирования на проникновение.

Введение

На настоящий момент JavaScript является одним из наиболее используемых языков программирования в области разработки веб-приложений. На каком бы языке не была написана серверная часть приложения, за клиентскую чаще всего отвечает именно JavaScript.

Одной из самых распространенных уязвимостей в программах на языке JavaScript является межсайтовый скриптинг (Cross-site scripting, XSS)[1]. Данная уязвимость находится на третьем месте в рейтинге уязвимостей от OWASP (Open Web Application Security Project) за 2015 год [2]. Межсайтовый скриптинг делится на 3 вида:

- отраженный (reflected);
- хранимый (stored);
- основанный на объектной модели документа (DOM-based, Documet Object Model).

В последнее время фокус злоумышленников смещается с серверной части приложения на клиентскую. Здесь им помогает достичь своих целей сравнительно новый тип XSS (по отношению к отраженным и хранимым) – межсайтовый скриптинг, основанный на объектной модели документа, который является уязвимостью нарушения целостности. DOM-based XSS не требует взаимодействия с сервером: атака может быть реализована

посредством изменения окружения DOM в браузере жертвы. Сама по себе страница (т.е. ответ на HTTP-запрос) не изменяется, однако скрипт, содержащийся на клиентской стороне выполняется по-другому из-за изменения DOM.

Таким образом, у разработчиков веб-приложений возникает необходимость в автоматизированном средстве поиска таких уязвимостей.

Разрабатываемый метод

Основой для создании динамического анализатора, обнаруживающего уязвимости DOM-based XSS, стала модель поведения аудитора, проводящего тестирование на проникновение. При тестировании на наличие DOM-based XSS аудитор сначала определяет все возможные точки входа в веб-приложение, а затем подает на эти входы специально сформированные данные – XSS-вектора. Чаще всего эти вектора, кроме всего прочего, содержат функцию alert(), вызывающую всплывающее окно, по которому аудитор может однозначно определить наличие уязвимости. Внедрить JavaScript-код через DOM-based XSS можно двумя способами:

- внедрение с помощью HTML-тега, например, “`<script>alert()</script>`”;
- непосредственное внедрение инструкций на языке JavaScript, например, “`alert();`”.

Динамический анализ делится на три этапа:

- инструментирование;
- выполнение программы:
- анализ полученных результатов.

При инструментировании исходного кода необходимо построить абстрактное синтаксическое дерево (АСД) кода. Для этих целей был использован парсер Esprima[3]. Для последующего анализа необходимо модифицировать полученное абстрактное синтаксическое дерево, что было произведено с помощью библиотеки falafel[4].

Исследователями были определенны методы языка JavaScript, через которые непроверенные данные могут попасть в веб-приложение. Такие методы получили название source (источник). В большинстве своем это свойства объектов document и location. Поэтому первыйм шагом инструментирования стало определение наличия источников и их замена на XSS-вектор. Фрагмент кода, выполняющего эту задачу представлен в Листинге1.

```
code = falafel(code, function(node){
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){};
    }
  }
});
```

Листинг 1. Поиск точек входа в приложение при объявлении переменной, передача на эти входы XSS-вектора.

Чтобы закрыть уязвимость, разработчику мало определить ее наличие, ему также необходимо знать точное место в коде, которое приводит к появлению дефекта. Для этого нужно выявить маршрут из функций и методов приложения, через которые проходит XSS-вектор. Представленный в Листинге 2 код проверяет есть ли среди аргументов вызываемой функции/метода XSS-вектор. Если есть, то имя функции/метода записывается в маршрут.

```
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){}');
              }
      }
      }
  }
```

Листинг 2. Определение функций и методов, в которые попадает XSS-вектор.

Таким образом, результатом работы анализатора будут маршруты прохождения данных по функциям/методам в процессе выполнения приложения.

Возьмем в качестве простейшего тестового примера программу представленную в Листинге 3.

```
var usrInput = document.referrer;
function modString(str1){
    str1 = str1 + 'smth';
}
modString(usrInput);
document.write(usrInput);
```

Листинг 3. Простейший пример для демонстрации работы анализаторв.

В результате инструментирования приведенного выше кода и его последующего выполнения в браузере, пользователь увидит всплывающее окно и получит следующие данные, выведенные в консоль:

```
modString document.write
```

Эти данные могут существенно сократить время поиска причины возникновения уязвимости.

Однако, одного XSS-вектора не достаточно, чтобы с уверенностью сказать уязвимо приложение или нет. Разработчики могут реализовать фильтрацию данных в своих приложениях, чтобы помешать злоумышленнику совершить атаку. В определенных случаях такие фильтры можно обойти. Например, если фильтр не допускает использования тега `<script>…</script>`, то осуществить атаку можно с помощью следующего вектора: <img src=’none.jpg’ onerror=’alert();’>. Кроме того, сущесвуют способы обойти фильтрацию таких специальных символов, как кавычки (одинарные и двойные) и угловые скобки. Из сказаного выше становится понятно, что JavaScript-код может быть внедрен в приложение несколькими способами. Поэтому для эффективной работы анализатора необходимо собрать некоторую базу различных XSS-векторов, которые затем будут последовательно подоваться на вход приложению. Формирование метода обработки результатов работы анализатора для нескольких XSS-векторов является дальнейшей целью исследования.

Заключение

В данной статье был представлен метод поиска уязвимостей нарушения целостности. Метод позволяет автоматизированно передать на вход программе данные (XSS-вектор), необходимы для атаки, после чего выводит информацию о функциях и методах, через которые прошел вектор в процессе выполнения программы. Этот маршрут может существенно сократить время локализации уязвимости.

Литература

1. Client-side JavaScript security vulnerabilities
   http://www.slideshare.net/orysegal/clientside-javascript-vulnerabilities
2. Top-10 vulnerabilities by OWASP
   http://www.ibm.com/developerworks/library/se-owasptop10/se-owasptop10-pdf.pdf
3. ECMAScript parsing infrastructer for multipurpose analysis
   http://esprima.org/index.html
4. Falafel – tool for AST modification
   https://github.com/substack/node-falafel

5. Jalangi - A dynamic analysis framework for JavaScript http://www.eecs.berkeley.edu/~gongliang13/jalangi_ff/
6. 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.
7. A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. Web Application Security Consortium, 2005.
8. С.П. Вартанов, А.Ю. Герасимов. Днамический 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.
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. 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.
11. 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.
12. T. Ball. The concept of dynamic analysis. In Software EngineeringESEC/FSE99, pages 216–234. Springer, 1999.
13. 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.