УДК 004.432.2

# СЛОЖНОСТИ ПРОВЕРКИ ИСХОДНОГО КОДА JAVASCRIPT НА ВРЕДОНОСНОСТЬ.

**Фрис А.В.**
**Научный руководитель — к.т.н. Кушнир В.П.**
*Сибирский федеральный университет*

Проверка произвольного исходного кода JavaScript на вредоносность чрезвычайно сложна. В то время как валидация структуры исходного кода – тривиальная задача, достаточно просто проверить, что код синтаксически корректен, то проверка содержимого уже совсем другое дело. Валидация содержимого блока кода означает, что мы должны убедиться, что код не выполняет вредоносных действий. Обычно вредоносный код JavaScript делает некоторые сочетания нижеследующего:

- доступ и манипулирование элементами DOM;
- отслеживание пользовательских событий;
- отслеживание событий браузера, таких как OnLoad и OnFocus;
- расширение или изменение встроенных объектов JavaScript;
- создание HTTP-подключений к другим доменам;
- создание HTTP-подключений к текущему домену.

К сожалению, данные вредоносные действия ничем не отличаются от таких же вполне легальных. Обычное дело для кода – манипулирование элементами DOM для создания эффектов DHTML. Он подписывается на различные события, чтобы реагировать на различные действия. Обычный код расширяет встроенные объекты языка по многим причинам. Предварительная загрузка изображений, веб-аналитика, подсчёт уникальных посетителей, интернет-реклама, скрытые фреймы являются вполне законными в обычных приложениях, где сценарии JavaScript посылают HTTP-запросы на всевозможные домены. Мы не можем окончательно определить, является ли сценарий вредоносным или нет, основываясь исключительно на том, какие действия он выполняет и какие возможности он использует. Вместо этого мы должны изучить контекст, в котором эти функции используются. Давайте предположим, что разработчик вручную анализирует исходный код и гарантирует, что он обращается только к соответствующим элементам DOM, не подписывается на определённые события и запрашивает статические изображения только от проверенных доменов. Может ли теперь разработчик отметить код как безопасный, зная, что он всё проверил? Ответ – нет. Возможно, что код делает больше, чем ему положено. JavaScript является весьма динамичным языком, который может значительно изменить себя во время исполнения. Практически все невстроенные функции могут быть переопределены. JavaScript позволяет даже так называемое динамическое исполнение кода, при котором исходный код хранится внутри строки и может быть передан интерпретатору для исполнения. JavaScript может генерировать этот код динамически или даже взять его из другого источника. Реальной опасностью с динамическим выполнением кода является то, что исходный код храниться в строке. Как эта строка собрана, остаётся полностью на совести разработчика. Злоумышленники почти всегда обфусцируют или зашифровывают строку для предотвращения обнаружения кем-либо дополнительной функциональности. А так как злоумышленник имеет практически неограниченное количество путей для шифрования и сокрытия вредоносного кода в строке, возможно, разработчики могли бы сосредоточиться на попытках обнаружить вызовы функций для динамического выполнения кода. Наиболее распространенным механизмом исполнения кода в строке является функ-

ция eval(). На первый взгляд кажется, что простое регулярное выражение, такое как /eval\s\(/ig должно помочь. К сожалению, это не так. Ниже приведены 13 примеров для вызова функции eval(), позволяющие обойти регулярные выражения для последовательности eval:

- eval.call(window, evalCode(1));
- eval['call'](window, evalCode(2));
- eval.apply(window, [evalCode(3)]);
- eval["apply"](window, [evalCode(4)]);
- window.eval.call(window, evalCode(5));
- window.eval['call'](window, evalCode(6));
- window['eval'].call(window, evalCode(7));
- window['eval']['call'](window, evalCode(8));
- window.eval.apply(window, [evalCode(9)]);
- window.eval['apply'](window, [evalCode(10)]);
- window['eval'].apply(window, [evalCode(11)]);
- window['eval']['apply'](window, [evalCode(12)]);
- var x = 'eval';
  var y = window;
  y[x]['ca' + String.fromCharCode(108, 108)](this, evalCode(13));

Последний способ вызова через массивы особенно мощный, поскольку позволяет злоумышленнику сослаться на функции eva(), call() или apply(), используя строки, а эти строки могут быть зашифрованы и обфусцированы различными путями. Последний пример также показывает, что невозможно создать регулярное выражение, чтобы обнаружить использование eval().

Более того, функция eval() не является единственным способом для выполнения кода JavaScript, хранящегося внутри строки. Это просто наиболее распространенный и хорошо известный способ динамического исполнения кода. Следующие примеры показывают еще несколько способов для выполнения динамически генерируемого кода:

- <img src="." onerror=evilCode />.
- var evilCode = "alert('evil');";
- window.location.replace("javascript:" + evilCode);
- setTimeout(evilCode, 10);
- setInterval(evilCode, 500);
- new Function(evilCode)();
- document.write("<script>" + evilCode + "</scr" + "ipt>");

Надеюсь, мы опровергли любое представление разработчиков об их способности обнаружить вредоносные фрагменты кода с помощью регулярных выражений. JavaScript имеет динамическую природу, его широкие возможности доступа к свойствам объекта с помощью строк, разнообразные средства вызова функций, а также множество методов исполнения кода внутри строки делают этот подход не действенным. Единственный верный способ, чтобы понять, что делает код на самом деле, нужно запустить этот код и посмотреть, что он делает. Только недавно исследователи по безопасности начали публично обсуждать действенные техники для анализа безопасности произвольного JavaScript кода.