Seguridad en JavaScript a través de reescritura de código

Julieta Álvarez, Alexis Ferreyra, Marcelo González, Ricardo Medel, Martín Molina, Néstor Navarro, Emanuel Ravera

Departamento de Ingeniería en Sistemas de Información
Universidad Tecnológica Nacional - Facultad Regional Córdoba

Maestro Lopez esq. Av. Cruz Roja Argentina
Córdoba, Argentina
nestornav@gmail.com

Resumen

Debido al crecimiento de la complejidad de las aplicaciones tanto para dispositivos móviles como para web, la eficiencia y seguridad de los programas JavaScript ha captado recientemente la atención de investigadores y desarrolladores. Dentro de esta línea de I+D investigamos la reescritura de código en tiempo de compilación como herramienta para mejorar tanto la eficiencia como la seguridad de este tipo de programas. En este trabajo presentamos esta línea de I+D y en particular los resultados de utilizar una herramienta de introspección de código, desarrollada por nuestro grupo y llamada PumaScript, para atacar el problema de seguridad web conocido como mXSS (Mutation-based Cross-Site-Scripting). Este tipo de ataque se vale de la inyección de código JavaScript malicioso, aprovechando las vulnerabilidades proporcionadas por la propiedad *innerHTML* y la función *Eval*. PumaScript reescribe los programas JavaScript donde se encuentren este código vulnerable, utilizando una función segura con una semántica análoga.

Palabras clave: JavaScript, Inyección de Dependencia, Meta-programación, Reescritura de código

Contexto

Las tareas descriptas en este trabajo se enmarcan en el proyecto "*Desarrollo de infraestructura de meta-nivel sobre JavaScript para el desarrollo de software multi-plataforma*", el cual se lleva a cabo como parte de las actividades de investigación y desarrollo del Departamento de Ingeniería en Sistemas de Información de la Facultad Regional Córdoba, Universidad Tecnológica Nacional. El proyecto, comenzado el 1 de mayo de 2013, es dirigido por el Dr. Ricardo Medel y tiene como objetivo principal la generación de una infraestructura para la construcción de software multiplataforma sobre el lenguaje JavaScript. El proyecto es financiado por la Secretaría de Ciencia, Tecnología y Posgrado de la UTN bajo el

código UTN1701 (Disposición SCTyP 173/13).

Introducción

Aunque existen varios estándares para la programación de páginas web y dispositivos móviles, la multiplicación de plataformas incompatibles entre sí aún se presenta como un problema. En este proyecto proponemos utilizar técnicas de meta-programación a fin de superar estos inconvenientes.

A pesar de los constantes avances tecnológicos, las diferencias entre las distintas plataformas web y móviles siguen siendo un desafío diario para los desarrolladores de software. Tecnologías como JavaScript [1] y HTML se han posicionado como una posible solución al desarrollo ágil de aplicaciones multiplataforma. Sin embargo, a pesar del desarrollo de estándares para HTML, aún existen importantes diferencias entre las entidades que definen los estándares (W3C, ECMA, etc.) y los proveedores de los *runtimes* para la web (Opera, Google, Microsoft, Mozilla, etc.).

Por otra parte, los lenguajes dinámicos han incluido algunas capacidades de meta programación prácticamente desde el comienzo de la historia de la programación. Entre los más importantes podemos nombrar a Lisp [2], cuya función *eval* acepta como argumento una instancia de estructura de datos que representa directamente código ejecutable (*S-expression*). Otros lenguajes de programación más recientes, tales como Groovy [3] o Escala [4] también brindan mecanismos de reescritura del árbol de sintaxis abstracto (AST) en tiempo de compilación, llegando al punto de poder agregar nuevas fases en el proceso de compilación. Otro ejemplo es Meta D++ [5], que fue desarrollado por este grupo de investigación en un proyecto anterior y provee técnicas de reescritura de código a través del uso de clases especiales en tiempo de compilación que pueden escribir partes de AST de las funciones llamadoras. En cuanto a la introspección de código, muchos lenguajes proveen mecanismos simples, tales como la función *instanceOf* de JavaScript o la capacidad de preguntar si un objeto responde a un cierto mensaje, pero casi ninguno de ellos permite la introspección completa del programa.

Es por ello que, luego de observar la diversidad existente de implementaciones de JavaScript y HTML y la carencia de un *framework* que provea técnicas de meta-programación, se decidió proponer el desarrollo de una infraestructura de meta-programación para estos lenguajes, que permita a los desarrolladores reescribir su código JavaScript para las diferentes plataformas, a la que denominamos PumaScript [6].

Líneas de Investigación, Desarrollo e Innovación

Dentro de las principales líneas de investigación que se plantearon para el proyecto podemos caracterizar las de introspección, análisis de tipos y reescritura de código JavaScript para mejoras en la eficiencia y seguridad de las aplicaciones.

Como parte de este trabajo, dos estudiantes realizaron su Práctica Supervisada (PS), de 200 horas de trabajo en una empresa o proyecto de investigación, requeridas para obtener el título de Ingeniería en Sistemas de Información. El tema encarado por estos alumnos, guiados por un tutor, fue analizar la posibilidad de evitar ataques maliciosos de inyección de código que

aprovechen las vulnerabilidades ocasionadas por *innerHTML* y *Eval* a través de la reescritura de código JavaScript.

De las múltiples formas de realizar inyección de código, toma mayor relevancia la propiedad *innerHTML*, debido a su fácil implementación. Esta propiedad provoca que se reemplace el contenido del DOM (document object model) del sitio web por otro. Si dicha modificación es realizada de forma externa, es decir por un tercero sin el control del desarrollador, desencadena un ataque generando así un bache de seguridad en el sistema. Debido a la popularidad de este tipo de ataques [9] y los efectos adversos que causan, es de suma importancia plantear una solución a dicha amenaza.

PumaScript detecta la presencia de la propiedad *innerHTML* en el código JavaScript y la reescribe utilizando una forma análoga basada en funciones propias del DOM (como ser *createElement*, *appendChild*, etc) las que proporcionan una mayor seguridad. Gracias al potencial de nuestro framework, podemos recrear la misma lógica con la que el desarrollador implementó el sitio web, pero evitando incurrir en baches de seguridad proporcionados por la propiedad ya mencionada. Esto se puede observar con mayor detalle en el ejemplo que se plantea en la Figura 1, basado en un caso de uso de la propiedad *innerHTML*.

En este ejemplo *stringHtml* es una caja de texto en la cual se ha introducido código malicioso por un usuario, y *contenedorInstancia* un tag HTML del tipo *div*. Como se puede apreciar, cualquiera sea el valor de *códigoExterno*, éste será ingresado como código HTML al elemento html *contenedorInstancia*.

Continuando con el ejemplo, si se introdujera un usuario externo en la caja de texto:

<h3 onmouseover="xss()">Soy un titulo<h3>

Estaríamos en presencia de un hueco de seguridad importante, ya que le está permitiendo a un usuario externo inyectar código malicioso.

Eval es una función que recibe como parámetro un string. Si dicho parámetro representa una expresión, *Eval* la evaluará y si el argumento representa una o más instrucciones, ejecutará cada una de las instrucciones.

La evaluación de código en tiempo de ejecución siempre será lenta, ya que implica compilar y ejecutar en runtime, situación que escapa a cualquier posible optimización por parte del intérprete del browser. Sin tener en cuenta las cuestiones relacionadas a la velocidad, los problemas de seguridad varían mucho según la implementación que se realice con la evaluación de código, llegando a representar en algunas situaciones inconvenientes de seguridad realmente críticos.

Obviamente, los riesgos que puede presentar la evaluación de código en el cliente son muchos menores que en el servidor, pero aun así existen numerosos 'exploits' que pueden ser utilizados gracias a la evaluación de código [8]. Dentro de nuestro trabajo hemos hecho una transformación de tres de los casos más utilizados del uso de *Eval*, esto son Evaluar una cadena del tipo JSON, Acceder Dinámicamente a Propiedades y

```
var contenedorInstancia = document.getElementById("contenedor");
var códigoExterno = document.getElementById("stringHtml").value;
contenedorInstancia .innerHTML = códigoExterno;
```

Fig. 1. Caso de uso de la propiedad innerHTML

Casos por Defecto. Los mismos se detallan a continuación.

1. Evaluar una cadena del tipo JSON: En este caso se evalúa una cadena JSON para obtener un objeto puro. Ejemplo: *var objectJson = eval('({' + stringJSON + '})')*.

El problema de seguridad en este caso radica en que el stringJSON podría ser un *request* a algún servidor, lo que llevaría a un riesgo en la inyección de código malicioso en la variable. Para solucionar este problema se transformó esa llamada a la función *Eval* en una llamada a la función *parse* del objeto JSON. Esta función descarta el argumento si éste no es un string con formato JSON. La expresión del ejemplo anterior, entonces, quedaría de la siguiente forma: *var objectJson = JSON.parse('{' + stringJSON + '}')*.

2. Acceder dinámicamente a propiedades: Este caso es uno de los más reportados, probablemente por desconocimiento de las características básicas del lenguaje, ya que el uso de la sentencia *Eval* suele ser innecesaria. Ejemplo: *var objectProperty = eval('info.' + getProperty())*. La solución para este caso sería la reescritura hacia la forma: *var objectProperty = info[getProperty()]*.

3. Caso por Defecto: Este último caso se trata de una situación que no cumple con las características para considerarse entre los anteriores. Ejemplo: *eval('var x=' + userString + ';')*. Aquí se declara una variable *x* y quedaría declarada en forma global para todo el documento JavaScript. La transformación para estos tipos de casos por defecto sería agregar al argumento la sentencia *use strict*; que define que el código debe ser ejecutado en "modo estricto". En dicho modo no se puede usar variables no declaradas. Luego de aplicar PumaScript, la transformación quedaría: *eval('"use strict";' + ('var x=' + userString + ';'))*. Para este ejemplo, la transformación limitaría el ámbito de la variable *x* al contexto del Eval únicamente, por lo que la declaración de la variable no sería global.

Resultados y Objetivos

Los problemas de seguridad planteados a lo largo de este trabajo (propiedad *innerHTML* y la función *Eval*) pudieron ser resueltos dentro del marco de trabajo de la PS, tomando como base para la resolución el framework PumaScript desarrollado previamente por este grupo.

La solución implementada para la resolución a los problemas de seguridad descriptos, para el caso de la propiedad *innerHTML* sin importar en qué sección del código se encuentre la misma (dentro de una llamada a una función, una sentencia *for*, etc) se reemplazará mediante una llamada a la API extendida de PumaScript, reescribiendo ésta por una nueva función llamada ***safeHTML***. La función *safeHTML* utilizará las funciones del DOM para recrear la lógica de *innerHTML* y a su vez evitar los huecos de seguridad mencionados. Esto es posible gracias al uso de una "lista negra" dentro de *safeHTML*, donde se registran aquellas sintaxis inválidas, como ser tags HTML tales como *script*, y eventos del tipo *on,* que posibilitan la inserción de código malintencionado.

La meta-función implementada en PumaScript buscará todos los nodos de asignación, y al encontrarlos se quedará sólo con aquellos que usen la propiedad *innerHTML*. Luego de la identificación, la meta-función generará el nodo *CallExpression* necesario para llamar a la API *safeHTML* y así reescribir el código inseguro.

Por ejemplo, si el código original es:

contenedorInstancia.innerHTML = códigoExterno;

Luego de utilizar PumaScript obtendremos la siguiente salida:

safeHtml(contenedorInstancia, códigoExterno);

En cuanto a la función *Eval*, se necesitaba reescribir los casos mencionados en la sección anterior. Para ello se desarrolló una meta-función en PumaScript, identificando en primera instancia todas aquellas llamadas a la función *Eval* usando la función intrínseca *pumaFindByProperty* de PumaScript. Esto devuelve todos los nodos del AST (Árbol de Sintaxis Abstracta) del programa que cumplen con la condición de que el nombre del "callee" sea *Eval*. Posteriormente se identificaron y clasificaron aquellos casos que pertenecían a cada nodo encontrado, volviendo a llamar a la función *pumaFindByProperty* propia del framework. Una vez que se cuenta con esta clasificación, se realiza la reescritura reemplazando los nodos por su alternativa segura. Para esto, de acuerdo a cada caso, se crean nodos con la función de Puma *pumaAst* manteniendo nombres de variables del nodo original para luego hacer el cambio en el AST.

Formación de Recursos Humanos

El grupo está formado por un director formado, tres ingenieros recientemente recibidos en formación y un alumno.

Becas y Prácticas Supervisadas realizadas en el marco del proyecto:

Durante este proyecto se han realizado las siguientes actividades de formación de Recursos Humanos:

- Tres alumnos de la UTN-FRC realizaron sus Prácticas Supervisadas (PS, requisito para obtener el título de Ingeniero en Sistemas de Información).
- Cuatro ingenieros recientemente egresados recibieron Becas de Iniciación a la Investigación y Desarrollo – BINID, financiadas por la Secretaría de Ciencia, Tecnología y Posgrado de la UTN, dos de ellos en el año 2013 y otros dos en 2014.
- En el año 2014, un alumno recibió la Beca de Iniciación a la Investigación para Alumnos, con financiación del Rectorado de la UTN.

Referencias

[1] ECMA Standard, http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (visitado 13/03/14)

[2] J. McCarthy, "The implementation of Lisp", History of Lisp, Stanford University, 1979.

[3] Groovy Compile Time Meta programming - http://groovy.codehaus.org/Compile-time+Metaprogramming+-+AST+Transformations (visitado 13/03/14)

[4] Scala Language - http://www.scala-lang.org/ (visitado 13/03/14)

[5] LayerD - Introducción a Tiempo de Compilación - http://code.google.com/p/layerd/wiki/Introduction_to_Compile_Time (visitado 13/03/14)

[6] "PumaScript" en GitHub, https://github.com/emravera/puma (visitado 13/03/14)

[7] Esprima - Project Page, http://esprima.org/ (visitado 13/03/14)

[8] Escodegen - Project Page, https://github.com/Constellation/escodegen (visitado 13/03/14)

[9] A Measurement Study of Insecure JavaScript Practices on the Web, http://dl.acm.org/citation.cfm?id=2460386 (visitado el 28/02/15)