Information Leakage Detection in JavaScript Browser Extensions

Jin Huang
Wright State University

Lu Liu
University of Colorado, Boulder

Jamie Weiss
Eastern Kentucky University

Abstract

Browser extensions are being used widely and commonly by many. However, they are not always secure and have a potential to contain vulnerabilities through leakage of sensitive user information. Extension examples include profile information autofills or shopping assistants which accesses a user's sensitive information such as account passwords or banking information. This access then leads to the risk of developers storing the information somewhere or sending it to miscellaneous network servers.

The system developed from this project checks browser extensions written in JavaScript for any possible vulnerabilities related to information leakage. It is designed with a mindset of creating an elegant way of analyzing extension files by storing the read file information in optimal data structures and targeting the behaviors that many vulnerabilities show. Sensitive information includes but is not limited to: forms, cookies, passwords, URLs, account information, various history, orders, and bookmarks.

Introduction

Online browser extensions are not always secure, and the system developed from this project is to serve the purpose of checking said browser extensions written in JavaScript for any possible vulnerabilities related to information leakage. The system is designed with a mindset of creating an elegant way of analyzing extension files by storing the read file information in optimal data structures and targeting the behaviors that many vulnerabilities show for economical performance. This interpreter system makes use of ESPRIMA to parse out ASTs from JavaScript files and then analyzes the outputted trees for source variables of sensitive information and sink locations that would expose the sensitive information to vulnerabilities. The ideal result is to have a final environment that contains a record of which variables have touched sensitive information and sink sources where said sensitive information is being leaked.

KEEP CITATIONS Citation of Einstein paper (?, ?). Citation of Freud book (Freud, 1900). Quick summary here, adding text here, CTRL + t

Background

When using online browsers, such as Google Chrome and Mozilla Firefox, certain capabilities can be added to personalize the userâĂŹs experience by allowing them access to tasks outside of their default browser. These capabilities are browser extensions, and are added to a userâĂŹs browser through the browserâĂŹs web store (Chrome Web Store or Addons for Firefox) According to Chromium, a project from Google dealing with open-source Chrome-based web browsing, in 2010, one-third of Google Chrome users had at least one browser extension installed to their browser. Most of these browser extensions are built using the JavaScript language, which allows them to work easily with the HTML of the browser to manage and manipulate data.

This ability to work with data through JavaScript presents the possibility of data leakage. This data leakage can occur as a result from moving data to a different location on the drive or even off of the host computer entirely, and can put sensitive information, such as account information, shopping and browsing history, and cookies at risk. This hazard of information leakage also opens the door for malicious entities purposefully making vulnerable browser extensions in order to obtain this sensitive information. According to Cisco, an electronics technology and security company, it is estimated that 85% of organizations are affected by malicious browser extensions.

Objectives

The goal of this project is to create a system that is able to detect vulnerabilities in JavaScript-based browser extensions. By analyzing the Abstract Syntax Trees of the source code, the system would determine whether or not the code in question contained sources and sinks that allow the code to manipulate and move sensitive information to an unwarranted location. In doing this, the system would allow for much safer use of browser extensions, both for the client and the web browsers. If bult efficiently, the system could even be used by the web browsers to examine any browser extensions attempting to be verified and put into the extension store.

Overall Framework

Rough overview of framework and system design

Parsing Abstract Syntax Trees

Abstract Syntax Trees (ASTs) were chosen as the primary data structure within this project to make use of the recursive capabilities that come with trees and their substructures. The way that the trees are structured is that each node contains some sort of type and will be nested within other nodes depending on the relationship. Each line of code within the extension files will have their own tree structure, providing us with multiple tree outputs to analyze.

For parsing out input JavaScript files into ASTs, the ESPRIMA parser was chosen for the amount of information provided and its compatibility with Node.js. The syntax tree format that it outputs comes with thorough documentation, and location features that report the locations of where certain nodes returned are found within the source code. For the trees that are outputted by ESPRIMA, the results were stored in .JSON files to help visually structure and further reference the trees from passed source codes.

The syntax tree format that ESPRIMA follows is a format that derives from Mozilla Parser API with additional formalization and expansion, named as the ESTreee specification. Each node is a basic JavaScript object with a string variable type. Different nodes have different descriptors set to type, and further additional variables are added depending on what structure type the node is. An appendix is provided that lists all of the syntax treeâĂŹs different type nodes and their containing variables, which aided in creating an interpreter with thorough and complete cases.

Building the Interpreter

Once the parser was found and its output from the source code understood, building the interpreter could begin. The idea for the interpreter was to âĂIJtraverseâĂİ the abstract syntax tree provided by the parser. That is, the interpreter would go from node to node down the tree, reaching the most basic parts of the program. This would be done using a switch case for each individual node, and recursively reaching the one below it until reaching the leaf nodes of said tree. This switch case would ideally contain every possible node type, and what the interpreter should do upon reaching that node type. With limited time, it has been agreed upon that only enough cases to be able to analyze a simple source code would be covered.

The cases of our interpreter are grouped into certain types of nodes, such as an operator, leaf, or function node. Operator nodes include cases that revolve around binary expressions, logical expression, and unary expressions. Leaf nodes include the very basic objects of boolean values, string literals, numbers, null objects, and variable identifiers. Function nodes include any built in functions within the JavaScript language, or a symbolic node to hold provided information for any user defined functions that the source code will future reference.

For an example of the mechanics of the interpreter, an AST will be passed through the evaluation function and will execute one of the switch statement cases depending on the type that it is. If the tree is an expression type, then it will pass the treeâĂŹs other nodes, such as a binary operator object type, through the evaluation function once more to analyze the tree until the leaf nodes are reached. With this interpretation of the source code, we are able to start looking at the content within the code to find possible vulnerabilities within function calls or assignment expressions.

Locating Sources Variables

Source variables vary widely and include multiple methods in identifying them. To find the most basic and general of sensitive information, several APIs and built in functions were focused on. There are functions to access a userâĂŹs cookies, bookmarks, forms, and more. There is also a function to obtain the values of user input, and these several functions were focused on in identifying the presence of sensitive information within the extension source code. Any variable declarations or assignment expressions were checked to see if the values that were set included sensitive keywords that would lead to a vulnerability.

Upon finding sensitive user information, the presence would be marked down in a class that keeps track of a variableâĂŹs details in regards to whether there is sensitive content or not. The class contains a dictionary where each key is a type of sensitive information, and the value that the key is set to is a boolean. A true value means that the variable that the dictionary belongs to has a value that contains sensitive information of that specific key type, such as user information, a password, URLs, or browsing history. With this tracking in place, the interpreter is able to mark down which variables have sensitive user information within it, and the type of sensitivity that the variable contains.

Locating Sink Functions

Identifying sink functions is similar to identifying source variables, where there are certain APIs or function calls that will either locally store or remotely send to a network a userâĂŹs sensitive information. The functions that we have focused on include APIs from browsersâĂŹ developer features (Mozilla, Chrome), building JavaScript features, XML server requests, and Ajax handling.

Vulnerability Analysis

The interpreter contains an environment that will hold all of the variables that are declared within the source code

given. Each variable that is tracked will also contain the previously mentioned dictionary that details what kind of sensitive information is contained within the variable if applicable.

To identify any vulnerabilities or leakages in the code, the interpreter currently has a trigger within the Call Expression case which is where function calls are handles. The case checks the passed object on whether it has the possibility of being a sink point and will then check the environment and any arguments that are passed through the function call. If the passed arguments contain variables that have been marked to have sensitive information, and alert will be outputted to report the finding of a possible vulnerability.

If given time for further development, it is a goal to implement an additional tracker to record the number of vulnerabilities found, along with the details of which function and sensitive variable, and to output the final analysis at the end of the program run. As of now, the interpreter only outputs a console alert to notify the user of vulnerability presence.

Evaluation

With the basic structure of the system built, testing could begin. The testing would be used to both check the accuracy of the system as well as to expand its capabilities in the future. Testing this system involved testing simple cases that would be found in an ordinary JavaScript source code. This test code would go through many adaptations until real scenarios of vulnerable source code could be tested. These real world source codes would allow more possibilities of sources and sinks to be found and added to the code to be monitored and marked as vulnerable. This method of using real world source code will be implemented further by pulling current source code from Google and MozillaâĂŹs web stores, in order to further expand and ensure the system is working as intended. Once the system can handle the majority of source code input given, work can begin on adding new features or goals for the system to add.

Tests Results

creating test codes and example vulnerabilities
running interpreter system on created examples

conclusion

system is able to identify a code's vulnerable sources and sinks
future expansion on further cases, sources, and sinks for more thorough usability
adding in real world extension source codes with the testing data
needing to continue expansion due to data driven model

References

Freud, S. (1900). *The interpretation of dreams*. Avon. (1965 ed.)