Vetting Browser Extensions for Security Vulnerabilities with VEX

Sruthi Bandhakavi  
Dept. of Computer Science  
Univ. of Illinois at Urbana  
Champaign  
sbandha2@illinois.edu

Nandit Tiku  
Dept. of Computer Science  
Univ. of Illinois at Urbana  
Champaign  
tiku1@illinois.edu

Wyatt Pittman  
Dept. of Computer Science  
Univ. of Illinois at Urbana  
Champaign  
wpittma2@illinois.edu

Samuel T. King  
Dept. of Computer Science  
Univ. of Illinois at Urbana  
Champaign  
kingst@illinois.edu

P. Madhusudan  
Dept. of Computer Science  
Univ. of Illinois at Urbana  
Champaign  
madhu@illinois.edu

Marianne Winslett  
Dept. of Computer Science  
Univ. of Illinois at Urbana  
Champaign  
winslett@illinois.edu

ABSTRACT

The browser has become the de facto platform for everyday computation and a popular target for attackers of computer systems. Among the many potential attacks that target or exploit browsers, vulnerabilities in browser extensions have received relatively little attention. Currently, extensions are vetted by manual inspection, which is time consuming and subject to human error. In this paper, we present VEX, a framework for applying static information flow analysis to JavaScript code to identify security vulnerabilities in browser extensions. We describe several patterns of flows that can lead to privilege escalations in Firefox extensions. VEX analyzes Firefox extensions for such flow patterns using high-precision, context-sensitive, flow-sensitive static analysis. We subject 2460 browser extensions to the analysis, and VEX finds 5 of the 18 previously known vulnerabilities and 7 previously unknown vulnerabilities.

1. INTRODUCTION

Driving the Internet revolution is the modern web browser, which has evolved from a relatively simple client application designed to display static data into a complex networked operating system tasked with managing many facets of a user’s on-line experience.

To help meet the varied needs of a broad user population, browser extensions expand the functionality of browsers by interposing on the seemingly simple task of filtering out ads based on a list of ad artifacts that interact with the browser in subtle and hard to understand ways. In some cases, extensions are simple and make only small changes to the appearance of web pages or the browser itself. Other extensions provide more sophisticated functionality, such as NoScript that provides fine-grained control over page JavaScript execution [8], or GreaseMonkey that provides a full-bloned programming environment for scripting browser behavior [3]. These are just a few of the thousands of extensions currently available for Firefox, the second most popular browser today.

Extensions written with benign intent can have subtle security-related bugs, called vulnerabilities, that expose users to devastating attacks from the web, often just by viewing a web page. Firefox extensions run with full browser privileges, so attackers can exploit extension weaknesses to take over the browser, steal cookies or protected passwords, compromise confidential information, or even hijack the host system, without revealing their actions to the user. Unfortunately, dozens of extension vulnerabilities have been discovered in the last few years, and capable attacks against buggy extensions have already been demonstrated [12].

In this paper, we propose VEX, a system for finding vulnerabilities in browser extensions using static information-flow analysis. Our key insight is that extension vulnerabilities often translate into explicit information flows from injectable sources to executable sinks. For extensions written with benign intent, most attacks involve the attacker injecting JavaScript into a data item that is subsequently executed by the extension under full browser privileges. We identify key flows of this nature that can lead to security vulnerabilities, and we check extensions for the presence of such flows using a high-precision static analysis that is both path-sensitive and context-sensitive, to minimize the number of false positive suspect flows. VEX has special features to handle the quirks of JavaScript (e.g., VEX does a constant string analysis for expressions that flow into the eval statement that execute dynamically generated code).

Determining whether extensions are malicious or harbor security vulnerabilities is a hard problem. Extensions are typically complex artifacts that interact with the browser in subtle and hard to understand ways. For example, the ADBLOCK PLUS extension performs the seemingly simple task of filtering out ads based on a list of ad servers. However, the ADBLOCK PLUS implementation consists of over 11K lines of JavaScript code. Similarly, the NoSCRIPT extension provides fine-grained control over which domains are allowed to execute JavaScript and basic cross-site scripting protection. The NoSCRIPT extension implementation consists of over 19K lines of JavaScript code. Also, ADBLOCK PLUS had 41 releases in 1/1/06–6/10/11, and NoSCRIPT had 48 releases just in 1/1/11–6/10/11. While Mozilla uses volunteers to vet each new extension and revision before posting it on their official list of approved Firefox extensions, examining an extension to find a vulnerability requires a detailed understanding of the code to reason about anything beyond the most basic type of information flow. Thus tools to help vet browser extensions can be very useful for improving the security of extensions.
2. THREAT MODEL, ASSUMPTIONS, AND USAGE MODEL

In this article, we focus on finding security vulnerabilities in buggy browser extensions. We do not try to identify malicious extensions, bugs in the browser itself, or bugs in other browser extensions, mechanisms, such as plug-ins. We assume that the developer is neither malicious nor trying to obfuscate extension functionality, but we assume the developer could write incorrect code that contains vulnerabilities.

We use two attack models. First, we consider attacks that originate from web sites, and we assume the attacker can send arbitrary HTML and JavaScript to the user's browser, modeling the usage model that assumes the user can navigate to any page on the internet. We focus on attacks where this untrusted data can lead to code injection or privilege escalation through buggy extensions. In the second attack model, we assume the same model as above, but we consider certain web sites as trusted. For example, if an extension gleaned information from the Facebook website, we assume that the Facebook data will not include arbitrary HTML and JavaScript, but only well formatted and trusted data.

According to the Mozilla developer site, Mozilla has a team of volunteers who help vet extensions manually. They run new and updated extensions isolated in a virtual machine to test the user experience. The editors also use a validation tool, which uses grep to look for key indicators of bugs. Many of the patterns they search for involve interactions between extensions and web pages, and they use their understanding of these patterns to help guide their inspection of the code. Our goal is to help automate this process, so that analysts can quickly hone in on particular snippets of code that are likely to contain security vulnerabilities. Figure 1 shows our overall work flow for using VEX: when extensions are subject to analysis by VEX, it reports precise code paths from untrusted sources to executable sinks in the extensions' code, which an expert must manually examine to check whether they can be used to mount an attack.

3. VEX INFORMATION FLOW PATTERNS

Firefox has two privilege levels: page for the web page displayed in the browser's content pane, and chrome for elements belonging to Firefox and its extensions. Page privileges are more restrictive than chrome privileges. For example, a page loaded from illinois.edu can only access content from illinois.edu. Firefox code and extensions run with full chrome privileges, which enables them to access all browser states and events, OS resources like the file system and network, and all web pages. Extensions also can include their own user interface components via a chrome document, which can run with full chrome privileges.

Firefox has APIs for extension code to communicate across protection domains and these interactions are one cause of extension security vulnerabilities. As the Mozilla developer site explains, “One of the most common security issues with extensions is execution of remote code in privileged context. A typical example is an RSS reader extension that would take the content of the RSS feed (HTML code), format it nicely and insert into the extension window. The issue that is commonly overlooked here is that the RSS feed could contain some malicious JavaScript code and it would then execute with the privileges of the extension—meaning that it would get full access to the browser (cookies, history etc) and to user’s files” [sic].

We characterize these cross-protection-domain interactions as information-flow patterns from JavaScript objects that include page content (untrusted sources) to JavaScript objects and methods that execute content with chrome privileges (executable sinks). In this section we discuss the sources and sinks that VEX tracks. Flows between these sources and sinks are sometimes benign, and represent an incomplete list of possible extension security bugs, but these are the patterns that VEX considers suspicious.

3.1 Untrusted sources

We now describe the untrusted JavaScript objects that extensions can access. Untrusted objects might contain foreign scripts that can lead to attacks if run with chrome privileges.

The JavaScript content-document object (window.content document) accesses the browser’s content page directly, and hence is an untrusted source. Also, the browser sets JavaScript pop-up nodes (document.popupNode) when the user clicks on document object model (DOM) elements. If this DOM element is part of the page content, then it includes untrusted page content.

One API that extensions use to access persistent state is the Resource Description Framework (RDF). RDF is a model for describing hierarchical relationships between browser resources [17] and is used by the browser to store persistent data, like bookmarks. Extension developers can store persistent extension data in an RDF file, or access browser resources stored in RDF format. However, RDF data can come from untrusted sources. For example, when a user stores a bookmark, Firefox records the un-sanitized title of the bookmarked page, which is controlled by the web page, in an RDF file. Extensions can also access un-sanitized bookmark URLs using the nsILivemarkService interface and the BookmarksUtils object.

Extensions access Firefox preferences through the nsIPrefService interface. Any extension can set values in the preferences, and extensions have unchecked access to all preference settings. Some extensions use this service to store untrusted strings obtained from web page content; hence using this service is also treated as an untrusted source.

In summary, the VEX treats the following as untrusted source objects: window.content document, document.popupNode, BookmarksUtils, and access to the new instances of the objects nsIRDFService, nsILivemarkService, and nsIPrefService.

3.2 Executable sinks

Now we describe the set of executable sinks, which are JavaScript objects and methods that provide a way to parse and execute JavaScript dynamically. VEX considers these executable sinks to be potentially dangerous when they execute untrusted JavaScript code with chrome privileges.
4. STATIC INFORMATION FLOW ANALYSIS

The eval function call interprets string data as JavaScript, which it executes dynamically. This flexible mechanism can be used to generate JavaScript code dynamically, for example to deserialize JavaScript Object Notation (JSON) objects. However, this flexibility can lead to code injection vulnerabilities in extensions. If extensions execute eval functions on un-sanitized strings that come from untrusted sources, an attacker can inject JavaScript code that runs with full chrome privileges.

Each HTML element in a page has an innerHTML property that defines the text that occurs between that element’s opening and closing tags. Extensions can change the innerHTML property to alter existing DOM elements, or to add new DOM elements, because the browser parses the modified text after JavaScript code modifies this property. Thus, passing specially crafted strings (e.g., <img> tags with JavaScript in their onload attribute) into innerHTML can lead to code injection attacks.

Extensions can add a new DOM element to a content page or chrome page by using the appendChild method. This method causes the browser to parse and process the data within the element, similar to the innerHTML property. Therefore, this feature can also be used to execute injected code.

In summary, the executable sinks that we consider in VEX are calls to the functions eval and appendChild, and assignments to innerHTML property.

4. STATIC INFORMATION FLOW ANALYSIS

The core component of VEX is a static analysis tool for detecting explicit information flows in browser extensions written in JavaScript. VEX computes flows between different sources and sinks, including all those described in Section 3. To support fine-grained information-flow analysis, VEX tracks the flows from source objects to the sinks encountered in the JavaScript extension, using a taint-based analysis. Motivated by the fact that every flow reported needs to be checked manually for attacks, which can take considerable human effort, we aim for an analysis that admits as few false positives as possible, where false positives are flows reported by VEX that cannot actually occur at run time.

Statically analyzing JavaScript extensions for flows is a non-trivial task. Object properties in JavaScript change dynamically, in the sense that new object properties can be created dynamically at run-time. Functions are objects in JavaScript, and hence can be created, redefined dynamically, and passed as parameters. In addition to the objects defined in the program, the extensions can also access the browser’s DOM API and the Firefox Extension API provided by XPCOM components, and the static analysis must handle them correctly. JavaScript browser extensions also have a large number of objects and functions that need to be tracked. The challenge is to accurately keep track of such objects, properties, and the corresponding flows to them.

The analysis engine in VEX is a static taint analysis to detect explicit flows, where taint propagation for JavaScript is achieved by adapting an operational semantics for JavaScript proposed by Maffeis et al. [14]. In the analysis, we replace concrete heaps by abstract heaps, where abstract heaps accurately track objects and their properties, but abstract the primitive values stored. An abstract heap can be seen as a directed graph, where every object and function in the JavaScript program is represented as a node, while the edges in the heap represent the field relationships between different objects. Additionally, every node in the abstract heap is associated with a taint value, which is used by VEX’s analysis to compute the information flows from the source objects to the sinks.

In the analysis, VEX handles only loop-free programs, and translates programs with loops to loop-free programs first by unrolling loops a bounded number of times (hence the analysis is not sound—see Section 4.3). The VEX abstract semantics computes and tracks the abstract heap on (loop-free) programs fairly precisely by mimicking the operational semantics for JavaScript. Unlike common abstraction domains used in the literature, at any point during the analysis, an abstract heap does not have a single node representing two objects; hence VEX is quite accurate in keeping track of the precise heap nodes and field relations and the corresponding flows, ignoring only the exact primitive values in the heap (like integers). Since programs are unrolled into loop-free code, the abstract heaps have a bounded size, leading to a terminating algorithm.

4.1 Abstract semantics of JavaScript

In this section, the abstract heap is described in detail, followed by a description of the data structures used for the static analysis. The high-level ideas behind VEX’s static analysis are also described.

Abstract heaps: We model the state of a JavaScript program using the notion of an abstract heap. Every object is stored on the heap. The heap is modeled as a set of (location, object) pairs. A location is an arbitrarily generated name created whenever a new object is created in the program. An object is a set of (property name, value) pairs. The property names could either be identifiers or strings. An abstract value could be a heap location (if the property points to another object), a function declaration, a security type, or a primitive value. Security types keep track of taints; a sink object’s security type acquires a taint associated with a source object, if there is an explicit flow from the source object to the sink object. A security type is modeled as a pair (taint value, source string); the taint value could either be Low or High and the taint source is a string identifying the source object of the taint. The primitive string values are preserved and propagated through string operations, whenever they evaluate to constant strings. All other primitive values are abstracted.

Figure 2 gives an example of a sample JavaScript heap computed using the VEX analysis. Every object and function in the JavaScript program is represented as a node in the heap, while the properties of the object are represented using edges in the graph. In the figure, the global object loc_Global has five properties ObjectProt, FunctionProt, Array, ArrayProt, and array_instance pointing to the nodes loc_ObjProt, loc_FunProt, loc_1, loc_ArrayProt, and loc_4 respectively. Every node in the heap is associated with a taint value, HIGH or LOW – HIGH representing the untrusted objects and LOW representing the trusted objects. High taints and low taints are represented by red and blue nodes, respectively, in the figures (all nodes in Figure 2 are LOW). Figure 3 shows the initial abstract heap representation of the window.content.document.object and the window.document.object; notice that one of the nodes loc_document, has a high taint.

The analysis: VEX analysis is based on a set of rules that transform abstract heaps according to each statement in the program, and it works by essentially over-approximating the effect of the statements on the abstract heap. These rules closely follow the small-step operational semantics proposed by Maffeis et al. [14], which covers the ECMA-262 standard for JavaScript. JavaScript core objects and functions are summarized to have only the essential functionality; an example summary is given in Section 4.2. Variables and functions that are not initialized in the current program execution or through summaries, are initialized to point to
placeholders dummy objects with high taints. The default taint of an object created in the extension is set to low unless the analysis explicitly sets the value to high or a variable is uninitialized. The loops in the program are unrolled a bounded number of times and function calls are inlined for a bounded unrolling of recursive calls, and every path of the resulting program is explored. Thus VEX function calls are inlined for a bounded unrolling of recursive calls, loops in the program are unrolled a bounded number of times and every object in the JavaScript heap has a special placeholder dummy objects with prototypes: JavaScript uses prototype-based inheritance [10]. Every object in the JavaScript heap has a special @Proto property, which is used to specify inheritance chains. Additionally, every function (that can be used as a constructor in new) has a prototype property. This prototype property is used to instantiate the @Proto property when a new object is created using the function constructor. An object inherits all the properties of its @Proto and of all the objects in the prototype’s @Proto chain.

Figure 2 illustrates how VEX handles prototype-based inheritance. The Array object in JavaScript is represented as the node @loc_1 in the figure. Since the Array object is a constructor, which can be used to create new instances of the array, it has a prototype field pointing to the object, ArrayProt, represented in the graph by the node @loc_ArrayProt. A new Array instance, array_instance, is created in the program using the statement: array_instance = new Array(). In Figure 2, @loc_4 represents the array instance object. The @Proto field of this object points to the object @loc_ArrayProt. Therefore, the push method is accessible to the array_instance object and can be called using the array_instance.push.

4.2 Handling other features of JavaScript

Function and Object Summaries: Natively supported functions and objects are replaced with stubs that summarize the effect on the heap and the taints when accessing them. VEX function and object summaries are hence simplified JavaScript objects and functions containing only the essential functionality of the objects. For example, a JavaScript Array object is defined in Figure 4 to be a function object with the @Proto of class, prototype, and @Proto properties initialized to the string “Function”, identifier ArrayProt, and identifier FunctionProt, respectively. The variables FunctionProt and ArrayProt point to the prototype objects, which contain the various functions like length and push.

```javascript
var Array = function () {
  this.@Class = "Function";
  this.prototype = ArrayProt;
  this.@Proto = FunctionProt; 
};
```

Browser’s DOM API and XPCOM components: VEX treats most of the browser’s DOM API and XPCOM components as uninitialized variables, fields, and functions. However, VEX provides explicit function summaries for the API components and objects that VEX needs to keep track of in order to trace the flows to and from the objects. VEX analysis sets the taint of the objects that represent insecure sources or those that are dependent on insecure sources to high.

Higher-order functions: VEX analysis accurately keeps track of the objects and implements function calls by inlining the function bodies according to the JavaScript semantics. Higher-order functions calls are also inlined. Additionally, VEX provides summaries for some higher order functions in the JavaScript API. For example, the setTimeout function in JavaScript takes a callback function as its first argument. This function is represented in VEX as a function in which the function body invokes the callback function in the first argument.

Dynamically generated code: The eval method in JavaScript allows execution of dynamically formed code, and is widely used in browser extensions. While an accurate analysis of the structure of dynamically created code is a research topic in itself, and quite out of the scope of this paper, the analysis cannot simply ignore eval statements. VEX analysis performs a constant-string analysis for strings and string operations. If the actual parameters to the eval statement evaluate to a constant string, VEX’s static analysis engine parses these constant strings and inserts them into the program flow just after the eval statement. This ensures that these newly parsed statements are included in the computation of the taint. In most correct extensions, an eval-ed statement is dynamically chosen from a set of constant-strings or taken from trusted sources, and hence evaluate to a constant string on the path explored (and tracked accurately by VEX). Parameters to eval, whose exact string values are not statically inferred by VEX along the path explored, are tested to check if they are tainted.

If there is a flow from an untrusted
source to an eval. VEX will report this flow, as it corresponds to a vulnerable flow pattern.

Object properties accessed in the form of associative arrays: In JavaScript, objects are treated as associative arrays. This means that any property of the object can be accessed using the array notation. Array indices could be constant strings, which are then evaluated to get the actual property being accessed; or they could be numbers, which indicate the property number that is being accessed; or they could be variables, that could be instantiated at run time. If VEX cannot evaluate the array index to a property name for any reason, the array access conservatively gets the taints of every property in the parent array object.

Functions that take arbitrary number of arguments: Some functions in JavaScript can have variable numbers of arguments. For example, the push method of an array can be called with any number of arguments and the arguments will be appended to the end of that array. To handle this in VEX, the object representing the push method has a special property indicating that it can take a variable number of arguments and when the method is called, VEX analysis conservatively appends the taints of all the arguments to the push method to the array object on which the method is called.

4.3 A note on soundness

Most static analysis tools, such as those used in compilers and those used in abstract interpretation, over-approximate the concrete semantics, and hence are sound. In the context of flow analysis, a sound tool never reports that a program has no flows when it has one. Soundness often entails a large number of false positives, i.e., flows that are reported by the tool but may not actually ever happen during execution.

VEX is not sound. We believe that a sound state-of-the-art analysis tool for JavaScript extensions would overwhelm and frustrate the tool’s users with a torrent of false positives. Thus to handle certain features of JavaScript without producing excessively many false positives, we chose not to make VEX sound. As a consequence, for example, a maliciously written extension could quite easily evade detection by VEX. On the other hand, a maliciously written extension can easily harm its users directly, without any input from untrusted web pages. This underlies the reason why our threat model assumes that the extension author is not malicious.

Instead of aiming for soundness, we concentrated on making VEX fairly accurate on paths in the program, without collapsing (merging) the nodes of the heap in any way. However, since VEX can only analyze a finite number of paths in the program (obtained by unrolling recursion a bounded number of times) in this accurate manner, the analysis VEX performs is inherently not sound.

False positives are also, of course, still possible in VEX, i.e., VEX may report flows that actually do not exist in the program. This stems from the fact that the analysis uses an abstraction. In particular, not having precise enough information for evaluating conditions, not precisely being able to determine the values of strings being subject to eval statements, etc. are common sources for false positives. Compared to classical heap analysis in programs that merges nodes in heaps, VEX performs a much more accurate analysis that reduces the number of false positives considerably. In experiments, we found that VEX produces very few false positives.

Overall, our choices were determined mainly by the complexity of JavaScript analysis and our aim at building a useful tool, which in turn led us to sacrifice soundness.

5. EVALUATION

VEX is implemented in Java (~ 7000 LOC), and utilizes a JavaScript parser built using the ANTLR parser generator for the Java 1.5 grammar provided by ANTLR [1]. ANTLR outputs Java-based Abstract Syntax Trees (AST) for JavaScript sources obtained from the pre-processing of the extension’s XUL and JavaScript files. The XUL files add different UI elements to the browser’s chrome. When any one of the user-interface elements is invoked and clicked, the corresponding event is triggered and the event-handler is called. We extract all such calls to the event-handlers from the XUL files and run them using VEX’s abstract operational semantics.

During the execution of the program using the abstract operational semantics outlined in Section 4, if the program reaches a vulnerable sink, it checks if the inputs or assignments to the sink are tainted. If they are tainted, VEX reports the occurrence of the flow along with the source objects and sink locations in the code. The source objects are the objects described in Section 3 and the sink locations are the points where the sinks described in Section 3 are encountered during the execution. The rest of this section summarizes our results.

The number of loop unrollings can be set as a parameter in the VEX analysis engine (in our experiments, a bound of just one was used). The VEX implementation has a number of optimizations to improve memory usage and speed. To save memory, abstract heaps are freed when backtracking in the depth-first search. But to save time, abstract heaps at join points are cached and compared when other paths hit these points, to avoid exploring paths unnecessarily.

5.1 Evaluation methodology

The extensions we analyzed were chosen as follows. First, in October 2008, we built a suite of extensions using a random sample of 1827 extensions from the Mozilla add-ons web site, by downloading the first extensions in alphabetical order for all subject categories. This extension suite had 2 extensions with known vulnerabilities. In November 2009, we downloaded 699 of the most popular extensions and 8 extensions with known vulnerabilities. The random sample and the popular extensions had 74 extensions in common, for a total of 2460 extensions. Our suite includes multiple versions of some extensions, allowing cross-version comparisons. For instance, we found a new version of the Fizzle (see [2]), to be vulnerable even though its authors tried to fix the vulnerabilities in the previous version.

We extracted the JavaScript files from these extensions and ran VEX on them, using a 2.4GHz 64 bit x86 processor with a maximum heap size of 16GB for the JVM.

To evaluate the effectiveness of VEX, we perform two kinds of experiments. First, we run VEX on the downloaded extensions and check if any of them have one of the malicious flow patterns. Second, we check if VEX can detect known extension vulnerabilities.

5.2 Experimental results

Finding flows from injectable sources to executable sinks: Figure 5 summarizes the experimental results for flows that are from injectable sources to executable sinks (flows for which the sinks are eval and innerHTML). Of the 2460 extensions analyzed by VEX, a grep showed that a total of 977 extensions had the occurrence of either the string “eval (“ or the string “innerHTML” or both.

The first column of Figure 5 indicates the exact source to sink flow pattern checked by VEX. The second column indicates the number of extensions on which VEX reports an alert with corresponding flows. On an average, VEX took 11.5 seconds per extension. It took about a week to analyze all the extensions with flows from untrusted sources to eval and innerHTML sinks.
To look for potential attacks, we manually analyzed the extensions with suspect flows found by Vex, spending about 20 minutes per extension on average. The next column reports the number of extensions on which we could engineer an attack based on the flows reported by Vex. We were able to attack nine extensions, of which only two extensions (FIZZLE VERSION 0.5 and BEATNIK V-1.0) were already known to be attackable. The rest of the attacks are new.

The next column shows the extensions where the source is provided either by the extension user or the extension developer or computed from the system parameters by the extension. The values are either stored in the preferences or in a local file. Since we trust the users and extension developers in our trust model, these extensions are considered to be non-vulnerable. However, if the preferences file or the local file system is corrupted in any way, these extensions can be attacked.

The fifth column shows the extensions where the source is code from a web site, and where an attack is possible provided the web site can be attacked. In other words, these extensions rely on a trusted web site assumption (e.g., that the code on the Facebook website is safe). We think that these are valid warnings that users of an extension (and Mozilla) should be aware of; trusted web sites can after all be compromised, and the code on these sites can be changed leading to an attack on all users of such an extension.

Not all flows lead to attacks—the next set of columns describe the alerts that were unable to convert to concrete attacks. Some extensions were not exploitable as the input is sanitized correctly (either by the extension or the browser), preventing JavaScript injection. Others extensions were not exploitable as the sinks were not in chrome executable contexts. These extensions are noted in the next two columns. Finally, Vex, being a static flow-analysis tool, does report alerts about flows that do not actually exist—there were very few of these, and are noted under the column “Non-existent flows”. Section 5.4 discusses the flows that do not lead to attacks.

**New vulnerabilities discovered:** The number of security vulnerabilities discovered are shown in column 3 in Figure 5, of which 7 are new. WIKIPEDIA TOOLBAR versions v-0.5.7 and v-0.5.9 have flows from window.content.document to eval, which leads to attacks. MOUSE GESTURES REDOX v-2.0.3 has flows from nsIPrefService to eval, which also led to an attack. BEATNIK V-1.2, FIZZLE V-0.5.1, and FIZZLE V-0.5.2 are also attackable, and have flows from nsIRDFService to innerHTML. KAIZOU V-0.5.8 has a flow from window.content.document to innerHTML which leads to attacks. Section 5.3 gives some details about the flows and the attacks in some of the vulnerable extensions. Details about FIZZLE (and BEATNIK) vulnerabilities can be found in the previous version of this article [2].

**Known vulnerabilities detected:** Apart from the new vulnerabilities found by Vex, there are several extensions that have been reported to be vulnerable in the past. In the course of our research, we found 18 unique extensions that were reported to be vulnerable in various databases like CVE, Secunia, etc. Of these 18, we did not find the source code for five extensions (GREASEMONKEY v≤0.3.5, WIZZ RSS v<3.1.0.0, SKYPE v<3.8.0.188, MOUSEOVERRDICTIONARY v<0.6.2, POW v<0.0.9), so we did not analyze them. Of the remaining 13 extensions, we found that 10 of them can potentially be found using explicit information flow analysis techniques, like Vex.

Currently, Vex can detect 5 of the above 10 known extensions that have flow-based vulnerabilities: FIZZLE v-0.5, BEATNIK V-1.0, COOLPREVIEWS v-2.7, 2.7.2, INFORSS v-<1.1.4.2, and SAGE v-< 1.3.9, <=1.4.3. COOLPREVIEWS has flows from document.popUpNode to appendChild. INFORSS has flows from nsIRDFService to appendChild. SAGE has flows from BookmarksUtil to an object accessing the local file system using the nsIFile interface.

The remaining 5 extensions have flow vulnerabilities but were not found by Vex for the following reasons. For FEEDSIDEBAR v-< 3.2, FIREBUG v-1.01, SCRIBEFIRE v-3.4.2, and UPDATE SCANNER v-3.0.3 the trigger of the flow was an event handler or a function call which was called outside the extension’s code base. In YOONO version ≤ 6.1.1 an un-sanitized JavaScript element like an image or link is rendered in the chrome context. However, it was difficult to find the source and sink objects from its source code.

Finally, there were three extension vulnerabilities (for which we had the source) that cannot be found by Vex because they are not flow vulnerabilities. These vulnerabilities include attacks on a file server (e.g., FIREFTP v<0.97.2, < 1.04), and directory traversal attacks (e.g., NAVIGATIONAL SOUNDS-VERSION-10.2, AJAX YAHOO MAIL VIAMATIC WEBSITE Review version-0.9) when a chrome package is “flat” rather than contained in a jar. In both the above cases, an attacker can escape from the extension’s directory and read files in a predictable location on the disk. Since such attacks are not related to chrome privilege escalations, and Vex does not handle them.

**5.3 Successful attacks**

**Attack scripts:** All our attack scenarios involve a user who has installed a vulnerable extension who visits a malicious page, and either automatically or through invoking the extension, triggers script written on the malicious page to execute in the chrome context. Figure 6 illustrates an attack payload that can be used in such attacks: this script displays the folders and files in the root directory.
The attack payloads could be much more dangerous, where the attacker could gain complete control of the affected computer using XPCOM API functions. More examples of such payloads are enumerated in the white-paper given in [7]. In this section, we illustrate a few attacks on extensions with previously unknown vulnerabilities.

The first line gets the first <script> element from the web page and executes it using eval. The extension developer assumes the user only clicks the buttons when a Wikipedia page is open, in which case <script> may not be malicious. But the user might be fooled by a malicious Wikipedia spoof page, or accidentally press the button on some other page. VEX led us to this previously unknown attack, which we reported to the developers, who acknowledged it, patched it, and released a new version. This resulted in a new CVE vulnerability (CVE-2009-41-27). The fix involved inserting a conditional in the program to check if the URL of the page is in Wikipedia’s domain and evaluating the script only if this is true.

**Kaizou v-0.5.8:** Kaizou is a web development extension that allows users to open the source of any web page in a separate window, modify the contents and render it again in the current window by pressing a button. However, this separate window has chrome privileges, and when the user saves the changes he made to the page source, the scripts in the page are executed with chrome privileges. A malicious web page can have an attack script, which could result in an attack when modified using KAIZOU.

**Mouse Gestures Redox v-2.0.3:** The MOUSE GESTURES REDOX extension allows users to create shortcuts for frequently used commands without using keyboard, menu or toolbars. The users can either create new gestures or download them from an online source. The new gestures are scripts, which are stored in the browser’s preferences file. When the gestures are enabled, they are retrieved from the prefs.js file and sent as arguments to the eval() function, thereby activating the gestures. If any of the gestures downloaded from the internet contain attack scripts, they would be executed in the chrome context when eval is called.

### 5.4 Flows that do not result in attacks

Figure 8 gives several examples of the suspect flows that we manually analyzed and for which either trusted sources were assumed by the extension or we could not find attacks.

The first set has extensions accessing values from web-sites or sources it trusts, and the values flow to eval or innerHTML. Of course, if the trusted sources are compromised, then the extensions may become vulnerable. The second set illustrates examples where the input was sanitized between the source and the sink. We do not know for sure that the sanitization is adequate, but we were unable to attack it. The third set of extensions had non-chrome sinks. The last set has two examples that show false positives where the flows reported by VEX do not exist in the code.

### 6. RELATED WORK

Maffeis et al. [14] proposed a small-step operational semantics for JavaScript, using which they analyze security properties of web applications. They also use their operational semantics for generating safe subsets of JavaScript and to manually prove that the so-called safe subsets of JavaScript are in fact vulnerable to certain attacks [15]. Our operational semantics follows their operational semantics, but works on an abstract heap. Guha et al. [10] propose an alternate operational semantics.

Ter Louw et al. [13] highlight some of the potential security risks posed by browser extensions, and propose run time support for restricting the interactions between browsers and extensions. Our analysis technique is complementary to their restrictions since even restricted interfaces can still be susceptible to security vulnerabilities.

More recently, researchers have developed static information flow analysis methods for JavaScript [4, 9]. In [4] the authors essentially perform a context-insensitive and flow-insensitive static analysis on the code, and delegate analysis of dynamic code to runtime checks. Guarnieri et al. [9] propose a mostly-static enforcement for JavaScript analysis, which is context-sensitive but flow-insensitive. In contrast, our analysis is both flow-sensitive and context-sensitive, thereby reducing the number of false positives.

Several dynamic analysis techniques with static instrumentation have been proposed for JavaScript to check information-flow properties [18, 11]. SABRE [5] is a framework for dynamically tracking in-browser information flows for analyzing JavaScript-based browser extensions. The taints are tracked by modifying the JavaScript interpreter. In contrast, Djeric et al. [6] dynamically track taints in both the browser’s native code and the script interpreter. Although dynamic techniques are useful in preventing certain types of script injection attacks if they are enforced by the web browser, they suffer from a few drawbacks. When a questionable flow is detected dynamically, the browser has to either choose an appropriate action (which might be overly restrictive) or ask the user to choose an action (which might lead to an attack if the user chooses a wrong option). Additionally, dynamic techniques impose a performance and memory overhead on the browser because of the need to keep track of the security label for every JavaScript object inside the browser. One of our main motivations was to facilitate a static analysis that scales to thousands of extensions, to circumvent
7. CONCLUSIONS

We have presented VEX, a tool for detecting potential security vulnerabilities in browser extensions using static analysis. VEX helps in automating the difficult manual process of analyzing browser extensions, by identifying and reasoning about subtle and potentially malicious flows. Experiments on thousands of extensions indicate that VEX is successful at identifying flows that indicate potential vulnerabilities and greatly reducing the number of flows that must be vetted manually. Using VEX, we identified 7 previously unknown security vulnerabilities and 5 known vulnerabilities, together with a variety of instances of unsafe programming practices.

An interesting future direction is to develop automatic ways to synthesize attacks that exploit flows reported by VEX. A technique based on constraint solving to generate attack inputs that satisfy the path constraints in the flow seems appropriate.

In the broader context, there is an increasing number of settings where small software teams (consisting of even one or two people) write software that is downloaded and used by hundreds of thousands of people. Browser extensions fall in this category, but several others have emerged, including mobile phone applications (for iPhone/Android/Windows) and Facebook applications. The teams writing these software do not always think about security carefully, leaving their users with potential privacy and integrity risks. We believe that precise static analysis tools, such as the one presented in this paper, combined with more precise and adaptable access control policies, can help address this security concern.

8. ACKNOWLEDGMENTS

We thank Chris Grier and Mike Perry who directed us to the Firefox extension vulnerabilities. This research was funded in part by NSF CAREER award #0747041, NSF grant CNS #0917229, NSF grant CNS #0831212, grant N0014-09-1-0743 from the Office of Naval Research, and AFOSR MURI grant FA9550-09-01-0539.

9. REFERENCES


<table>
<thead>
<tr>
<th>Classification</th>
<th>Extension</th>
<th>Flow pattern</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source is trusted web site</td>
<td>WORLD SMS V-2.2</td>
<td>Unknown var to eval</td>
<td>The source is a web site: <a href="http://worldsms.co.cc/json?get=info">http://worldsms.co.cc/json?get=info</a>, which is hardcoded into the extension code.</td>
</tr>
<tr>
<td></td>
<td>BROWSER BACKGROUNDS V-0.3.5</td>
<td>nsIRDFService to innerHTML</td>
<td>The user installs background images from the web site <a href="http://browserbackgrounds.com/">http://browserbackgrounds.com/</a>.</td>
</tr>
<tr>
<td>Sanitized input</td>
<td>ALPHANUMERATOR V-0.2</td>
<td>Content Doc to innerHTML</td>
<td>The input string is converted to numbers effectively sanitizing it.</td>
</tr>
<tr>
<td></td>
<td>VIEW SOURCE CHART V-2.7</td>
<td>Content Doc to innerHTML</td>
<td>The display page has a chrome url but is marked to be &quot;content-type.&quot;</td>
</tr>
<tr>
<td>Non-chrome sinks</td>
<td>PAGEDIFF V-1.3.0</td>
<td>Content Doc to innerHTML</td>
<td>The input string is converted to numbers effectively sanitizing it.</td>
</tr>
<tr>
<td>Non-existent flows</td>
<td>LINK_ALERT V-0.8.2.1</td>
<td>Unknown var to eval</td>
<td>eval’s argument is a packed and minified JavaScript string that VEX could not parse properly. VEX finds an unknown variable in this incorrectly parsed string.</td>
</tr>
<tr>
<td></td>
<td>SKIPSSCREEN V-0.1.09102009</td>
<td>Unknown var to innerHTML</td>
<td>During the execution, the extension will never follow the branch that has the sink, as the conditional variable is never initialized in the program.</td>
</tr>
</tbody>
</table>

Figure 8: Extensions that could not be attacked


