VEX: Vetting Browser Extensions For Security Vulnerabilities

Sruthi Bandhakavi Samuel T. King P. Madhusudan Marianne Winslett
University of Illinois at Urbana Champaign
{sbandha2,kingst,madhu,winslett}@*illinois.edu*

Abstract

The browser has become the *de facto* platform for everyday computation. 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 does not scale well and is subject to human error.

In this paper, we present VEX, a framework for highlighting potential security vulnerabilities in browser extensions by applying static information-flow analysis to the JavaScript code used to implement extensions. We describe several patterns of flows as well as unsafe programming practices that may 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 analyze thousands of browser extensions, and VEX finds six exploitable vulnerabilities, three of which were previously unknown. VEX also finds hundreds of examples of bad programming practices that may lead to security vulnerabilities. We show that compared to current Mozilla extension review tools, VEX greatly reduces the human burden for manually vetting extensions when looking for key types of dangerous flows.

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 and interacting with browser-level events and data. Some 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 JavaScript execution [20], or GREASEMONKEY that provides a full-blown programming environment for scripting browser behavior [6]. These are just a few of the thousands of extensions currently available for Firefox, the second most popular browser today[1].

Extensions written with benign intent can have subtle vulnerabilities that expose the user to a disastrous attack from the web, often just by viewing a web page. Firefox extensions run with full browser privileges, so attackers can potentially 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, tens of extension vulnerabilities have been discovered in the last few years, and capable attacks against buggy extensions have already been demonstrated [23].

To help reduce the attack surface for extensions, Mozilla provides a set of security primitives to extension developers. However, these security primitives are discretionary, and can be difficult to understand and use correctly. For example, Firefox provides an `evalInSandbox (text, sandbox)` function that returns the result of evaluating the `text` string under the restricted privileges associated with the environment `sandbox`. Using `evalInSandbox` correctly requires developers to test the result of a call to `evalInSandbox` with the non-traditional "===" rather than "==", as the "==" operation may invoke unsafe code as a side effect (See `http://developer.mozilla.org/En/Components.utils.evalInSandbox` for details).

Current approaches from the research community propose dynamic techniques for improving the security of extensions. The SABRE system tracks tainted JavaScript

[1] Firefox now surpasses Internet Explorer in W3schools traffic (`www.w3schools.com/browsers/browsers_stats.asp`), arguably due to the popularity of Firefox extensions.

objects to prevent extensions from accessing sensitive information unsafely [9]. Although SABRE can prevent potentially malicious flows from both exploited extensions and from malicious extensions, SABRE adds overhead to all JavaScript execution within the browser, adding 6.1x overhead for the SunSpider benchmark and 2.36x overhead for the V8 JavaScript benchmark. Furthermore, SABRE's dynamic nature pushes security violation notification to users who are unable to determine if a particular flow is malicious or benign. The Google Chrome Extension System revisits the overall extension API to make it easier for the browser to enforce least privilege and strong isolation on extensions [4]. Their system works by partitioning the full set of extension functionality into different protection domains, and sand-boxing extensions to prevent them from obtaining more privileges than needed. Although this system is likely to limit the damage from some extension attacks, it does little to prevent the vulnerabilities themselves.

In this paper, we propose VEX, a system for finding vulnerabilities in browser extensions using static information-flow analysis. Many vulnerabilities translate to certain types of *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 analyze for these flows statically using high-precision static analysis that is both path-sensitive and context-sensitive, to minimize the number of false positive suspect flows. VEX uses precise summaries to analyze code, and 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). Because VEX uses static analysis, we avoid the runtime overhead induced by dynamic approaches.

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 30 releases in 1/1/06–11/20/09, and NOSCRIPT had 38 releases just in 1/1/09–11/20/09. 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.

We show that VEX can catch several known vulnerabilities, such as a vulnerability in the FIZZLE extension [8], and also find new problems, including exploitable vulnerabilities in BEATNIK and WIKIPEDIA TOOLBAR. In particular, VEX reported a previously unknown vulnerability in WIKIPEDIA TOOLBAR that could lead to an attack, and that resulted in the report CVE-2009-4127. We reported this vulnerability to the WIKIPEDIA TOOLBAR developers, who fixed the extension. We also show that VEX can help to find the use of unsafe programming practices, such as misuse of `evalInSandbox`, that can result from subtle information flows.

The remainder of the paper is organized as follows. Section 2 describes the threat model and the assumptions under which we analyze the browser extensions. Section 3 provides background material on the architecture of Firefox and the nature of certain key undesirable information flows in its extensions. Section 4 describes our static analysis and the various design choices we made to build VEX. Section 5 lists and describes our results. Section 6 surveys related work, and Section 7 concludes the paper.

2 Threat model, assumptions, and usage model

In this paper, 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 extensibility 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. 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 consider some web sites as trusted. For example, if an extension gleans information from Facebook, we assume that the Facebook code 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.

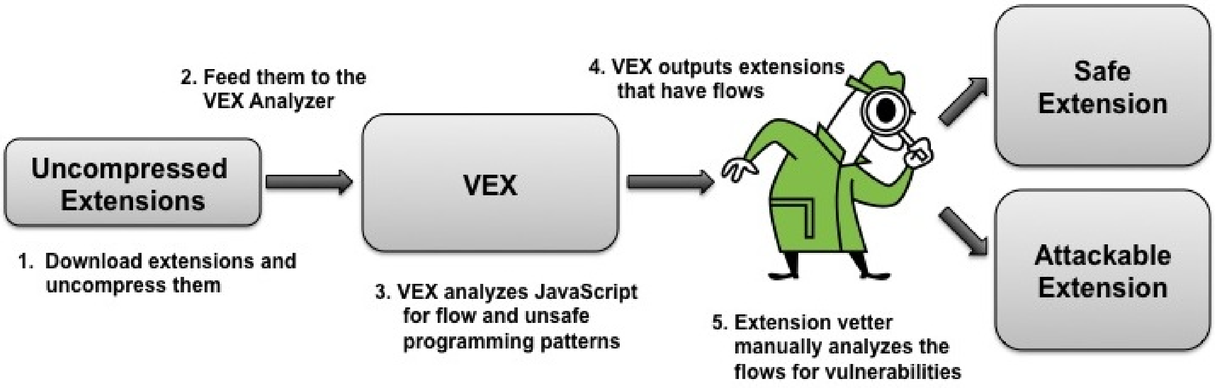
2. Feed them to the VEX Analyzer
4. VEX outputs extensions that have flows
Safe Extension
Uncompressed Extensions
VEX
Attackable Extension
1. Download extensions and uncompress them
3. VEX analyzes JavaScript for flow and unsafe programming patterns
5. Extension vetter manually analyzes the flows for vulnerabilities

Figure 1: The overall analysis process of VEX.

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.

3 Background

3.1 Mozilla privilege levels

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, i.e., everything surrounding the content pane. Page privileges are more restrictive than chrome privileges. For example, a page loaded from site x cannot access content from sites other than x. General Firefox code runs with full chrome privileges, which give it access to all browser states and events, OS resources like the file system and network, and all web pages. Firefox provides the extensions with full chrome privileges by exposing a special API called the XPCOM Components to extension JavaScript, thereby allowing the extensions to have access to all the resources Firefox can access.

Extensions can often access objects that run with page privileges and interact with page content, as well as objects that run with full chrome privileges. Extensions can also include user interface components via a *chrome document*, which also runs with full chrome privileges. For example, the object `window` refers to the chrome window and the object `window.content` refers to the content window. To access the `document` object referring to the content (i.e., the user page), the extension has to access the `document` property of the content window, i.e., `window.content.document`.

To make this extension architecture practical, Firefox has APIs for extension code to communicate across protection domains. 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].

3.2 Points of attack

Here we discuss key vulnerable points for code injection and privilege escalation attacks against non-malicious extensions: `eval`, `evalInSandbox`, `innerHTML`, and `wrappedJSObject`. We focus on these Firefox features because they are key points of interaction between objects with page and chrome privileges, respectively, and this interaction is a key source of security vulnerabilities, as noted above. Though other avenues of attack are possible, we do not consider them here.

eval: 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 serialize 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 web pages, the attacker will be able to inject JavaScript code that will run with full chrome privileges.

InnerHTML: Each HTML element for a page has an innerHTML property that defines the text that occurs between that element's opening and closing tag. Extensions can change the innerHTML property to alter existing document object model (DOM) elements, or to add new DOM elements. When an extension modifies the innerHTML property, the browser re-parses and processes the new data. Thus, passing specially crafted unsanitized strings (e.g., <img> tags with script in their onload attribute) into innerHTML modifications can lead to code injection attacks.

EvalInSandbox: One way Firefox facilitates communication across protection domains is through the evalInSandbox method. This method enables extensions to execute JavaScript in the extension's context with restricted privileges, thus enabling extensions to process untrusted data from web pages safely. The sandbox object is an empty JavaScript object created with restricted privileges. For example, the call s = Sandbox("http://www.w3.org/") creates a sandbox s where code can execute with page privileges, as though it came from the domain www.w3.org. One can add properties to this object by calling the evalInSandbox function, and any attempts to access global scope objects from within evalInSandbox, including privileged chrome objects, are denied. evalInSandbox complicates extension programming because objects returned from the method call execute in the extension with full chrome privileges. Since methods associated with the object could have been modified within the sandbox, they should not be called in the chrome context. For example, "==" should not be used on these objects as its evaluation calls the tostring or valueOf method, which could have been modified; instead the non-traditional "===" operator needs to be used.

wrappedJSObject: JavaScript objects can be dynamically modified. That means that any web page can modify the properties of the document object. For example, a web page can reassign the getElementById method to return a malicious script. To prevent this script from being executed by the extension when it calls window.content.document.getElementById, Firefox automatically wraps the object so that the window.content.document accesses only use the original document object, not the modified one. However, Firefox also provides the wrappedJSObject method, which lets the extension access the modified version, even when automatic wrapping is turned on; calling wrappedJSObject on a content document is potentially dangerous.

3.3 Suspicious flow patterns

In this section we discuss the five source to sink flows that might be vulnerable. Specifically, we track flows from Resource Description Framework (RDF) data (e.g., bookmarks) to innerHTML, content document data to eval, content document data to innerHTML, evalInSandbox return objects used improperly by code running with chrome privileges, or wrappedJSObject return object used improperly by code running with chrome privileges. These flows do not always result in a vulnerability, and they are by no means an exhaustive list of all possible extension security bugs, but they are the patterns we use in our tool.

RDF is a model for describing hierarchical relationships between browser resources [33]. Extension developers can store persistent extension data in an RDF file, or access browser resources, such as bookmarks, stored in RDF format. 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 in the RDF file. Extensions that use RDF data need to sanitize it properly if they use it directly in an innerHTML statement that modifies an element in a chrome document.

Content document data flowing to eval or innerHTML can sometimes be exploited. This flow can result in script execution with chrome privileges if specially crafted content from the window.content.document object is passed to eval or innerHTML or an element in the chrome document.

For evalInSandbox and wrappedJSObject, problems can only result if the return values of these constructs are executed with chrome privileges. For evalInSandbox this means comparing return values using == or != from code running with chrome privileges. For wrappedJSObject, this means making method calls on returned objects from code running with chrome privileges.

Such flow patterns may occur in only a few of the extensions that use these constructs. According to the Mozilla extension review web page, reviewers have an open-source automatic tool to help with reviews (see https://addons.mozilla.org/

`en-US/firefox/pages/validation`), but this tool just greps for strings that indicate dangerous patterns. Afterward, the reviewer must go through the code of each suspect extension to understand the flows and determine which constitute vulnerabilities and which are benign. As this task is difficult, painful, and error-prone, we designed the VEX tool to help extension reviewers vet the flows in extensions automatically, greatly reducing the number of extensions that need manual review.

4 Static information flow analysis

We develop a general explicit information flow static analysis tool VEX for JavaScript that computes flows between any source and sink, including the flows described in Section 3.3. While we could develop analysis techniques for a particular source and sink, we prefer a more general technique that will perform the analysis once, and from the results, allow us to search for any source-to-sink flow. This allows VEX to be run in a single pass over thousands of extensions, rather than using separate passes for each target pattern.

To support fine-grained information-flow analysis, VEX tracks the precise dependencies of flows from variables to objects created 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 (false positives are non-existent flows reported by the tool).

Statically analyzing JavaScript extensions for flows is a non-trivial task. JavaScript extensions have a large number of objects and functions. 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. The objects are also dynamic, 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. The challenge is to accurately keep track of such objects, properties, and the corresponding flows to them.

Our analysis keeps track of an *abstract heap* (AH) that is not *a priori* bounded, and keeps track of the precise heap nodes and field relations and corresponding flows, but ignores the exact primitive values in the heap (like integers). However, we bound *the number of iterations* in computing the least fixed-point, and hence the abstract heap gets bounded implicitly.

The abstract heap transformations for any statement closely mimic a big-step operational semantics for JavaScript, except that primitive values are forgotten, and hence conditionals are not evaluated; we refer the reader to work on operational semantics of JavaScript [27, 18].

Apart from tracking heap structures, the abstract heap also records *explicit-flow* dependencies to heap nodes, and the rules for updating flows naturally depend on the program's semantics. Also, as we talk about in more detail below, there are some aspects of the heap (such as prototype fields) that are not currently supported in our tool. The static analysis itself is flow-sensitive and context-sensitive, and the context-sensitivity is handled using classical function-summary based methods.

The above choices, namely the choice of abstract heaps, and the context-sensitive flow-sensitive analysis, are *design choices* we have made, based on our experiments with extensions for over a year, and were motivated to reduce false positives. However, we have not tried all variants of these choices, and it is possible that other choices (for example, choosing to bound abstract heaps by merging objects created at a program site), may also work well on extensions. However, we do know that *context-sensitivity* is important (in several extensions we manually examined) and further *flow-sensitivity* seems important if the tool is extended to consider sanitization routines as flow-stoppers.

The rest of this section is structured as follows. First we explain our analysis using abstract heaps for a core subset of JavaScript, which does not have statements like `eval`, associative array accesses, calls to Firefox APIs, etc. Subsequently, we describe how we handle the aspects not covered in the core.

4.1 Analysis of a core subset of JavaScript

Core JavaScript: A core subset of JavaScript is given in Figure 2; this core reflects the aspects of JavaScript described above, but omits certain features (such as `eval`) which we will describe later.

Abstract Heaps: Our analysis keeps track of a *one abstract heap* at each program point. This *abstract heap* tracks JavaScript objects and functions and the relationships between them in the form of a graph. Each node in the graph is a heap location generated by the program. Two different nodes, n_1 and n_2 are connected by an edge labeled f, if node n_1's property f *may refer* to n_2. To keep track of the actual information flows between different program variables, we also keep track of all the program variables that flow into the nodes in abstract heap. Let $PVar$ be the set of all the program variables in the JavaScript program.

More precisely, an abstract heap σ is a tuple (ns, n,d, fr, dm, tm), where:

- ns is a set of heap locations,

EXPRESSIONS ::=

|	c	(CONSTANT)
|	x	(VARIABLE)
|	$x.f$	(FIELD ACCESS)
|	$x.prot$	(PROTO ACCESS)
|	$eop\ e$	(BINARY OP)
|	`this`	(THIS)
|	$\{f_1 : e_1, \ldots, f_n : e_n\}$	(OBJECT LITERAL)
|	`function` $(p_1, \ldots, p_n)\{S\}$	(FUNCTION DEF)
|	$f(a_1, \ldots, a_n)$	(FUNCTION CALL)
|	`new` $f(a_1, \ldots, a_n)$	(NEW)

STATEMENTS ::=

|	`skip`	(SKIP)
|	$S_1; S_2$	(SEQ)
|	`var` x	(VARIABLE DECL.)
|	$x := e$	(ASSIGN 1)
|	$x.f := e$	(ASSIGN 2)
|	`if` e `then` S_1 `else` S_2	(CONDITIONAL)
|	`while` e `do` S `od`	(WHILE)
|	`return` e	(RETURN)

Figure 2: Core JavaScript syntax.

- $n \in (ns \cup \{\bot\})$ represents the *current node*, and is either a node in the heap or the symbol $\bot$,
- $d \subseteq PVar$ represents the subset of program variables that flow in to the current node n,
- $fr \subseteq ns \times PVar \times (ns \cup \{\bot\})$ encodes the pointers representing properties (fields). A triple $(n_1, f, n_2) \in fr$ means that the property f of the object n_1 *may be* located at n_2.
- $dm \subseteq ns \times PVar$ is a relation that denotes a *dependency map*. A pair $(n_1, x) \in dm$ denotes that the program variable x flows into the node n_1.
- $tm : ns \times ns$ is a "this-map" relation, which is actually the relation of a function. A pair $(n_1, n_2) \in tm$ means that the scope of n_1 is n_2.

Notation: The relation tm will always be a function; we define formally the function $tm : ns \to ns$ as $tm(n) = n'$, where $(n, n') \in tm$. Let $dm : ns \to 2^{PVar}$ be the function that corresponds to the relation dm, $dm(n) = \{x | (n, x) \in dm\}$, i.e. the set of all the program variables that flow into the node n.

The Analysis: We now describe our analysis for the core subset of JavaScript. VEX handles functions and objects by creating a node for every object or function and their properties. Relationships between various nodes are accurately generated and tracked in the analysis. JavaScript uses prototype-based inheritance; however, our analysis does not track prototypes. Instead, a new property insertion into the prototype field of an object is treated as if the property is being inserted into the object itself. We found that this is sufficient in case of JavaScript extensions as the inheritance chain is not deep in most cases. VEX keeps track of the accurate scope information using the this-map.

Our analysis consists of a set of rules for generating abstract heaps at program points, and is defined by essentially capturing the effect of statements on the abstract heap. These rules follow a big-step operational semantics adapted to work on the abstracted heap.

The big step operational semantics on abstract heaps is defined as a relation , $(Prog, \sigma) \Downarrow \sigma'$, where `Prog` is an program expression or statement and σ and σ' are abstract heaps. Such a relation intuitively means that σ' is the heap obtained from the complete evaluation of `Prog` starting from the heap σ. This resulting heap, in every iteration, will be *merged* with the current heap after the program, conservatively taking the union of dependencies.

We now define this relation for expressions and statements.

Notation: For any abstract heap σ, let $\sigma = (ns_\sigma, n_\sigma, d_\sigma, fr_\sigma, dm_\sigma, tm_\sigma)$. In other words, n_σ refers to the second component of σ, etc. The function $fresh()$ creates a new heap location. A special node n_G represents the global heap, which consists of the objects like `Object`, `Array`, etc.

Evaluating expressions:
Figure 3 gives the rules for evaluating expressions in the program.

Rule (CONSTANT) evaluates to a $\bot$ node with empty dependencies. Rule (THIS) extracts the scope of the current node. The next five rules describe the variable and field access expressions.

In case of a variable access, the existence property x is checked in the current scope (represented by n_σ(rule (VAR))), and returned if it exists. If it is not in the current scope, then the global node (rule (GLOBAL VAR)) is checked for property x. If it exists, then it is returned with dependencies. If the location for a particular variable is found in neither the current scope nor the global scope, using rule (UNINITIALIZED VAR) we create a new node n_{new} and add it to the global scope. Similar rules apply for field accesses in rules (FIELD ACCESS) and (UNINIT FLD).

For binary operators(rule (BINARY OP)), we return the union of dependencies of both the expressions. When an object literal expression((OBJ. LIT.)) is encountered, a *summary* is computed by recursively creating heap locations for each of its properties and then creating the

$$\frac{\cdot}{(c,\sigma)\Downarrow(ns_\sigma,\bot,\emptyset,fr_\sigma,dm_\sigma,tm_\sigma)}\ (\textsc{Constant})\qquad \frac{\cdot}{(\texttt{this}\ ,\sigma)\Downarrow(ns_\sigma,tm_\sigma(n_\sigma),dm_\sigma(tm_\sigma(n_\sigma)),fr_\sigma,dm_\sigma,tm_\sigma)}\ (\textsc{this})$$

$$\frac{(n_\sigma,x,n_x)\in fr_\sigma}{(x,\sigma)\Downarrow(ns_\sigma,n_x,dm_\sigma(n_x),fr_\sigma,dm_\sigma,tm_\sigma)}\ (\textsc{Var})\qquad \frac{\nexists n'_x.(n_\sigma,x,n'_x)\in fr_\sigma\quad (n_G,x,n_x)\in fr_\sigma}{(x,\sigma)\Downarrow(ns_\sigma,n_x,dm_\sigma(n_x),fr_\sigma,dm_\sigma,tm_\sigma)}\ (\textsc{Global Var})$$

$$\frac{\nexists n'_x.(n_\sigma,x,n'_x)\in fr_\sigma\quad \nexists n''_x.(n_G,x,n''_x)\in fr_\sigma\quad n_G\neq n_\sigma}{(x,\sigma)\Downarrow(ns_\sigma\cup\{n_{new}\},n_{new},\emptyset,fr_\sigma\cup\{(n_G,x,n_{new})\},dm_\sigma,tm_\sigma\cup\{(n_{new},n_G)\})}\ (\textsc{Unitialized Var})\ where,\ n_{new}=fresh()$$

$$\frac{(x,\sigma)\Downarrow\sigma'\quad (n_{\sigma'},f,n_f)\in fr_{\sigma'}}{(x.f,\sigma)\Downarrow(ns_{\sigma'},n_f,d_{\sigma'}\cup dm_{\sigma'}(n_f),fr_{\sigma'},dm_{\sigma'},tm_{\sigma'})}\ (\textsc{Field Access})\qquad \frac{(x,\sigma)\Downarrow\sigma'}{(x.prot,\sigma)\Downarrow\sigma'}\ (\textsc{Prot Access})$$

$$\frac{(x,\sigma)\Downarrow\sigma'\quad \nexists n_f.(n_{\sigma'},f,n_f)\in fr_{\sigma'}}{(x.f,\sigma)\Downarrow(ns_{\sigma'}\cup\{n_{new}\},n_{new},d_{\sigma'},fr_{\sigma'}\cup\{(n_{\sigma'},f,n_{new})\},dm_{\sigma'},tm_{\sigma'}\cup\{(n_{new},n_{\sigma'})\})}\ (\textsc{Uninit Fld})\ where, n_{new}=fresh()$$

$$\frac{(e_1,\sigma)\Downarrow\sigma_1\quad (e_2,\sigma)\Downarrow\sigma_2}{(e_1\,op\ e_2,\sigma)\Downarrow(ns_{\sigma_1}\cup ns_{\sigma_2},\bot,d_{\sigma_1}\cup d_{\sigma_2},fr_{\sigma_1}\cup fr_{\sigma_2},dm_{\sigma_1}\cup dm_{\sigma_2},tm_{\sigma_1}\cup tm_{\sigma_2})}\ (\textsc{Binary Op})$$

$$\frac{(e_1,\sigma)\Downarrow\sigma_1\quad\ldots\quad(e_n,\sigma)\Downarrow\sigma_n}{(\{f_1:e_1,\ldots,f_n:e_n\},\sigma)\Downarrow\sigma'}\ (\textsc{Obj. Lit.})\quad where,\ \begin{array}{ll} n_{\sigma'}=fresh()=n_{new} & d_{\sigma'}=\bigcup_{i=1}^{n}d_{\sigma_i}\\ ns_{\sigma'}=ns_\sigma\cup\{n_{new}\}\cup(\bigcup_{i=1}^{n}ns_{\sigma_i}) & fr_{\sigma'}=fr_\sigma\cup(\bigcup_{i=1}^{n}(n_{new},f_i,n_{\sigma_i}))\\ dm_{\sigma'}=dm_\sigma\cup(\bigcup_{i=1}^{n}dm_{\sigma_i}) & tm_{\sigma'}=tm_\sigma\cup(\bigcup_{i=1}^{n}(n_{\sigma_i},n_{new}))\end{array}$$

$$\frac{(S,\sigma'')\Downarrow\sigma'}{(\texttt{function}\ (p_1,\ldots,p_n)\{S\},\sigma)\Downarrow\sigma'}\ (\textsc{fun-def})\quad where,\ \begin{array}{l} ns_{\sigma''}=ns_\sigma\cup\{n^0_{new}\}\cup(\bigcup_{i=1}^{n}n^{p_i}_{new})\quad n_{\sigma''}=fresh()=n^0_{new}\quad d_{\sigma''}=\emptyset\\ n^{\texttt{\_\_RET}}_{new}=fresh()\quad \forall i\in\{1,\ldots,n\}.n^{p_i}_{new}=fresh()\\ fr_{\sigma''}=fr_\sigma\cup\{(n^0_{new},\texttt{\_\_RET},n^{\texttt{\_\_RET}}_{new})\}\cup(\bigcup_{i=1}^{n}\{(n^0_{new},i,n^{p_i}_{new})\})\\ dm_{\sigma''}=dm_\sigma\cup(\bigcup_{i=1}^{n}\{(\texttt{\_\_RET},i),(n^{p_i}_{new},i)\})\\ tm_{\sigma''}=tm_\sigma\cup\{(n^0_{new},n_\sigma)\}\cup\{(n^{\texttt{\_\_RET}}_{new},n^0_{new})\}\cup(\bigcup_{i=1}^{n}(n^{p_i}_{new},n^0_{new}))\end{array}$$

$$\frac{(f,\sigma)\Downarrow\sigma''\quad(n_{\sigma''},\texttt{\_\_RET},n')\in fr_\sigma\quad(e_1,\sigma)\Downarrow\sigma_1\quad\ldots\quad(e_n,\sigma)\Downarrow\sigma_n}{(f(e_1,\ldots,e_n),\sigma)\Downarrow(ns_\sigma,\bot,d',fr_\sigma,dm_\sigma,tm_\sigma)}\ (\textsc{fun-call1})\quad where, d'=\bigcup_{i=1}^{n}(\exists(n',i)\in dm_\sigma.d_{\sigma_i})$$

$$\frac{(f,\sigma)\Downarrow\sigma''\quad n_{\sigma''}=\bot\quad(e_1,\sigma)\Downarrow\sigma_1\quad\ldots\quad(e_n,\sigma)\Downarrow\sigma_n}{(f(e_1,\ldots,e_n),\sigma)\Downarrow(ns_\sigma,\bot,\bigcup_{i=1}^{n}d_{\sigma_i},fr_\sigma,dm_\sigma,tm_\sigma))}\ (\textsc{fun-call2})$$

Figure 3: Semantics for all core expressions except new.

graph where the new object node is linked to the properties with the labeled edges.

A function definition((FUN-DEF)) is treated in a similar fashion as the object literal, except that new summary locations are created for each of the function arguments and also for the return variable (i.e. $n^{\texttt{\_\_RET}}_{new}$). The function body is evaluated with respect to the new heap. The result of the evaluation is the new heap with the function summary attached to the node $n^{\texttt{\_\_RET}}_{new}$. A function call(rule (FUN-CALL1)) uses this summary to compute the node and dependencies of the return value. The return value of the function can be obtained by evaluating each of the function argument expressions, and replacing the appropriate nodes in the function summary with the values returned. If the function is not defined, then the dependencies of the return values are the union of dependencies of the individual function parameters(rule (FUN-CALL2)). A constructor expression (containing new) is similar to a function call, where if the object being instantiated is retrieved from the local or the global scope, then a copy of the graph starting with this object is created and returned.

Evaluating statements:
The statement semantics are given in Figure 4. A variable declaration(VAR. DECL.) creates a new node in the current scope. If the heap node for that variable already exists, it is replaced by this new node. The assignment statement (rules (ASSIGN1) and (ASSIGN2)) evaluates the left hand side and the right hand side expressions, replaces the node on the left hand side with the node on the right hand side. Note that conditionals in if-then-else and while statements are, of course, not evaluated as our heaps are symbolic. The while state-

$$\frac{\cdot}{(\texttt{skip}, \sigma) \Downarrow \sigma}\ (\textsc{Skip}) \qquad \frac{(C_1, \sigma) \Downarrow \sigma' \quad (C_2, \sigma') \Downarrow \sigma''}{(C_1; C_2, \sigma) \Downarrow \sigma''}\ (\textsc{Seq})$$

$$\frac{(n_\sigma, x, n_x) \in fr_\sigma}{(\texttt{var}\ x, \sigma) \Downarrow (ns, n_\sigma, d_\sigma, fr, dm_\sigma, tm)}\ (\textsc{Var.Decl.}) \quad \begin{array}{l} where, ns = (ns_\sigma \cup \{n_{new}\}) \setminus \{n_x\} \\ fr = (fr_\sigma \setminus \{(n_\sigma, x, n_x)\}) \cup \{(n_\sigma, x, n_{new})\} \\ tm = tm_\sigma \cup \{(n_{new}, n_\sigma)\} \end{array}$$

$$\frac{(e, \sigma) \Downarrow \sigma'' \quad (x, \sigma) \Downarrow \sigma_x}{(x := e, \sigma) \Downarrow (ns, n_\sigma, d_\sigma, fr, dm, tm)}\ (\textsc{Assign1}) \quad \begin{array}{l} where, ns = ns_{\sigma_x} \cup ns_{\sigma''} \\ fr = (fr_{\sigma''} \setminus \{(n_\sigma, x, n_{\sigma_x})\}) \cup \{(n_\sigma, x, n_{\sigma''})\} \\ dm = dm_{\sigma''} \\ tm = tm_\sigma \cup \{(n_{\sigma''}, tm_{\sigma_x}(n_{\sigma_x}))\} \end{array}$$

$$\frac{(e, \sigma) \Downarrow \sigma'' \quad (x, \sigma) \Downarrow \sigma_x \quad (x.f, \sigma) \Downarrow \sigma_f}{(x.f := e, \sigma) \Downarrow \sigma'}\ (\textsc{Assign2}) \quad \begin{array}{l} where, n_{\sigma'} = n_\sigma \quad d_{\sigma'} = d_\sigma \\ fr_{\sigma'} = (fr_{\sigma''} \setminus \{(n_{\sigma_x}, f, n_{\sigma_f})\}) \cup \{(n_{\sigma_x}, f, n_{\sigma''})\} \\ dm_{\sigma'} = dm_{\sigma''} \cup \{(n_{\sigma''}, y) | y \in d_{\sigma_x}\} \cup \{(n_{\sigma_x}, y) | y \in d_{\sigma''}\} \\ tm_{\sigma'} = tm_\sigma \cup \{(n_{\sigma''}, n_{\sigma_x})\} \end{array}$$

$$\frac{(S_1, \sigma) \Downarrow \sigma_1 \quad (S_2, \sigma) \Downarrow \sigma_2}{(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma) \Downarrow (ns_{\sigma_1} \cup ns_{\sigma_2}, n_\sigma, d_\sigma, fr_{\sigma_1} \cup fr_{\sigma_2}, dm_{\sigma_1} \cup dm_{\sigma_2}, tm_{\sigma_1} \cup tm_{\sigma_2})}\ (\textsc{Cond})$$

$$\frac{(S_1, \sigma) \Downarrow \sigma'}{(\texttt{while}\ e\ \texttt{do}\ \ S_1\ \texttt{od}\ , \sigma) \Downarrow \sigma'}\ (\textsc{While1}) \qquad \frac{(S_1, \sigma) \Downarrow \sigma' \quad (\texttt{while}\ e\ \texttt{do}\ \ S_1\ \texttt{od}\ , \sigma') \Downarrow \sigma''}{(\texttt{while}\ e\ \texttt{do}\ \ S_1\ \texttt{od}\ , \sigma) \Downarrow \sigma''}\ (\textsc{While2})$$

$$\frac{(e, \sigma) \Downarrow \sigma'}{(\texttt{return}\ e, \sigma) \Downarrow (ns_{\sigma'}, n_\sigma, d_\sigma, fr_{\sigma'} \cup \{(n_\sigma, \texttt{\_\_RET}, n_{\sigma'})\}, dm_{\sigma'}, tm_{\sigma'})}\ (\textsc{Ret})$$

Figure 4: Statement semantics.

ment is interesting: we evaluate the while body till we reach a fixed point (or till we reach a fixed number of loop un-rollings) as depicted in (WHILE2). However, notice that the abstract heap is also allowed to immediately go across a while-loop (WHILE1). The semantics for the rest of the statements is standard.

Given the above rules for abstract heaps, we start analyzing the JavaScript program using an initial state consisting of a global heap, represented by node n_G. This global heap consists of summaries for a few built-in objects like Array. We evaluate the rules either till we converge on a least fixed-point, or till we reach a preset bound on the number of iterations.

4.2 Handling other features of JavaScript

Dynamic 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, we cannot simply ignore `eval` statements. Our approach has been to implement a static *constant-string analysis* for strings and subject the strings that are `eval`-ed to this analysis. Our static analysis engine inserts these constant strings into the code (as though it was static code), parses it, and computes the flows for them. Strings that are not statically known but subject to `eval` are essentially ignored, and this causes our tool to be unsound (see a later note on unsoundness). In most correct extensions, an `eval`-ed statement is dynamically chosen from a set of constant-strings or taken from trusted sources. Note that if there is a flow from an *untrusted* source to an `eval`, VEX will catch this flow, as it is a vulnerable flow pattern.

innerHTML: Modifications of the `innerHTML` of an HTML page by the extension makes the analysis considerably more complex. For instance, if a function `a()` calls function `b()` that calls function `c()`, and `c()` makes `innerHTML` modifications, it is hard to summarize this effect in the summary of `c()`, as the source of the flow is not locally available. We handle this by creating a *symbolic* representation of the source, computing summaries of `innerHTML` using this symbolic source, and allowing outside methods to instantiate the symbolic source to a concrete source in whichever context it becomes available.

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. VEX treats these cases in a conservative manner. Whenever a property is created in the node scope, its dependencies are added to the dependencies of the node as shown in the (ASSIGN 2) rule in the Figure 4. If we cannot evaluate the array index for any reason, it would be sufficient to retrieve the dependencies of the object.

Functions that take arbitrary number of arguments: Some functions in JavaScript can have variable numbers of arguments. For example, the $push$ method of the array can be called with any number of arguments and the arguments will be appended to the end of the array. To handle this, the summary of the $push$ method has a special field indicating that it can take variable number of arguments and when the method is called, we conservatively append the dependencies of all the arguments to the dependency set of the node representing the array object.

Browser's DOM API and XPCOM components: These objects are treated as uninitialized variables, fields and functions. The rules (UNINITIALIZED VAR), (UNINIT FLD) and (FUN-CALL2) can be applied to their accesses. When we need to keep track of the usage of certain components we introduce the component API function arguments into the dependency set. For example the RDF datasource is accessed using the following command:

```
rdf = Components.classes
      ["@mozilla.org/rdf/rdf-service;1"]
      .getService(Components.interfaces.nsIRDFService);
```

Our analysis introduces the string *"@mozilla.org/rdf/rdf-service;1"* and the variable *nsIRDFService* into the dependency set of the left hand side variable rdf.

4.3 Unsoundness and incompleteness

A static analysis tool like VEX is inherently conservative. First, if VEX reports a flow, there may be no such feasible flow in the program (i.e. VEX can have false positives). Though VEX *over-approximates* flows and tries to perform a sound analysis, there are several aspects of the analysis which, if implemented soundly, will make the tool throw too many infeasible flows, making it useless in practice.

Consider a program where there is an `eval` of a string that is dynamically created and not determinable statically. Since this string can be assigned any value, it could be any arbitrary program that can create flows between any of the variables in scope. A sound tool must *necessarily* summarize the `eval` as causing flows from all variables to all nodes, which would generate plenty of false positives and would essentially be useless. False negatives (i.e. miss detecting programs that have a flow), are also possible because of the fact that we have several uninitialized and unsummarized objects.

VEX has several sources of unsoundness and incompleteness: handling of `eval`, handling of prototypes, handling of higher-order functions, fixed number of unrolls of loops, handling `with`-scoping, handling exceptions, etc.

5 Evaluation

5.1 VEX implementation

The VEX tool checks for two kinds of flows: one from injectable sources to executable sinks to check for script-injection vulnerabilities, and the other, also modeled as flows, that checks for unsafe programming practices. VEX is implemented in Java (∼ 2000 LOC), and utilizes a JavaScript parser built using the ANTLR parser generator for the JavaScript 1.5 grammar provided by ANTLR [1]. ANTLR outputs Java-based Abstract Syntax Trees (AST) for JavaScript files, and VEX walks through the ASTs computing the flow sets from all interesting sources to all interesting sinks, in a single pass analysis, using the static analysis described in Section 4. For each sink object, VEX collects all the source objects that flow into it and checks for the occurrence of flow patterns. VEX reports these flows to the user along with the source and sink locations in the code.

Flow patterns checked: The current version of VEX checks for the following three flow patterns that capture flows from injectable sources to executable sinks:

- **Content Doc to Eval:** The source location is any point where the program accesses the API `window.content.document`, and the source object is the object that is returned from this call. The sink locations are `eval` statements and the sink objects are the objects being `eval`-ed.

- **Content Doc to innerHTML:** The source location and source objects for these flows are the same as above; the sink locations are the places where the extension writes directly into the DOM using `innerHTML` commands, and the sink objects are the objects being assigned by the `innerHTML` command. These DOM elements may be executable if they are in the chrome context.

- **RDF to innerHTML:** The source location and source objects are given by any retrieval of RDF objects (which are often injectable) and the sink locations and sink objects are `innerHTML` commands as above.

Furthermore, VEX searches for the following patterns that characterize two documented unsafe programming practices that could lead to security vulnerabilities:

- **evalInSandbox object to == or `!=`:** This flow is meant to detect an unsafe programming practice where an object retrieved by an `eval` in a sandbox is subject to an == or `!=` test (the recommended practice is that such objects must be tested with ===). The source location is hence any `evalInSandbox`-statement and the corresponding source objects are the objects *returned* by the `evalInSandbox` call. The sink locations are usages of == and `!=`, and the sink objects are the objects that are subject to these comparisons.

- **Method Call on wrappedJSObject:** Objects obtained using wrappedJSObject() commands are usually untrusted, and methods of such objects should not be called. The source locations are hence uses of wrappedJSObject() and source objects are the objects *returned* by them. Sink locations are methods calls and the sink objects are the objects whose methods are called.

The VEX tool can, of course, be adapted to other kinds of suspect flows – source and sink locations are straightforward, and the source and sink objects must be specified carefully as above.

5.2 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. In November 2009, we downloaded 699 of the most popular extensions. The two sets had 74 extensions in common, for a total of **2452 extensions.** Our suite includes *multiple versions* of some extensions, allowing cross-version comparisons. For instance, we found a vulnerability in a new version of BEATNIK (see Section 5.4), though its authors thought the vulnerabilities in the previous version were fixed.

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 4GB for the JVM.

5.3 Experimental results

Finding flows from injectible sources to executable sinks: Figure 5 summarizes the experimental results for flows that are from injectible sources to executable sinks (the first three flows outlined above). The first column is the number of extensions that syntactically has code that could indicate such a flow, identified using a `grep`-search. For the flow "Content-doc to Eval", the `grep` was for the string 'eval('; for "Content-doc to InnerHTML" flows, the `grep` was for the string 'innerHTML'; and for "RDF to InnerHTML" flows, the search was for both the strings "'innerHTML" and "@mozilla.org/rdf/rdf-service;1". As the table shows, this search finds hundreds of suspect extensions, far more than can be examined manually.

The third column indicates the number of extensions on which VEX reports an alert with corresponding flows. On an average, VEX took only 15.5 seconds per extension.

To look for potential attacks, we manually analyzed most of the extensions with suspect flows that VEX alerted us on, spending about two hours 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 six extensions, of which only three extensions were already known to be vulnerable. The attacks on Wikipedia Toolbar, Fizzle version 0.5.1 and Fizzle version 0.5.2 extensions are new, see Section 5.4 for more details.

The next 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 we were unable to convert to concrete attacks. Some flows were not exploitable as the input is *sanitized* correctly (either by the extension or the browser), preventing JavaScript injection, while others were not exploitable as the sinks do not turn out to be chrome executable contexts. These extensions are noted in the next two columns. Finally, VEX, being a conservative 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". A discussion on flows that do not lead to attacks is given in Section 5.5.

As noted in the last column, there were 13 extensions with VEX alerts that were too complex(or obscurely written) for us to manually analyze for an attack; we do not know whether attacks on these are possible or not.

Flow Pattern	grep Alerts	VEX Alerts	Attackable Extensions	Source is trusted website	Not Attackable			Unanalyzed
					Sanitized input	*Non-chrome sinks*	*Non-existent flows*	
Content Doc to `eval`	430	13	2*	1	0	3	5	2
Content Doc to `innerHTML`	534	46	0	14	6	6	9	11
RDF to `innerHTML`	60	4	4**	0	0	0	0	0

Attackable Extensions: * WIKIPEDIA TOOLBAR V-0.5.7, WIKIPEDIA TOOLBAR V-0.5.9 ,
** FIZZLE V-0.5, FIZZLE V-0.5.1, FIZZLE V-0.5.2 & BEATNIK V-1.2

Figure 5: Flows from injectible sources to executable sinks.

Unsafe Programming Practices	grep Alerts	VEX Alerts
`evalInSandbox` Object to == or !=	107	3
Method Call on `wrappedJSObject`	269	144

Figure 6: Results for unsafe programming practices.

Finding unsafe programming practices:
The results of the second set of experiments for flows that characterize the two unsafe programming practices of checking equality on objects evaluated in a sandbox and calling methods of unwrapped JS objects are shown in Figure 6.

The first column denotes the flow-pattern, the second column shows the number of extensions that had a grep pattern for the strings ‘evalInSandbox’ and ‘wrappedJSObject’, respectively. The third column shows the number of extensions that VEX alerts. Note that these flows correspond to unsafe programming practices declared by Mozilla for extension writers, and hence should be avoided. We analyzed 15 of the alerts and found that all of the flows we inspected were feasible and real, but we were unable to manually confirm the remainder because there were too many alerts to examine.

5.4 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 7 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 [13].

```
<script>
var root = Components.classes
["@mozilla.org/file/local;1"].createInstance
(Components.interfaces.nsILocalFile);
try {
root.initWithPath("/."); // for Linux or Mac
}catch (er){
root.initWithPath("\\\\."); // for Windows
}
var drivesEnum = root.directoryEntries,
drives = [];
while (drivesEnum.hasMoreElements()) {
drives.push(drivesEnum.getNext().
QueryInterface(Components.interfaces.
nsILocalFile).path);
}
alert (drives);
</script>
```

Figure 7: Attack script to display directories.

Let us now explain the various attacks we found on web extensions:

Wikipedia Toolbar, up to version 0.5.9
If a user visits a web page with the directory display attack script in its <head> tag, and clicks on one of the Wikipedia toolbar buttons (unwatch, purge, etc.), the script executes in the chrome context. The attack works because the extension has the code given in Figure 8 in its toolbar.js file.

```
script = window._content.document.
getElementsByTagName(``script")[0].innerHTML;
eval (script);
```

Figure 8: Wikipedia toolbar code.

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 on the wikipedia domain and evaluating the script only if this is true.

```
bookmarks.js:
 1. function Bookmarks(){
 2.  var bookmarks = new Array();
 3.  this.load = function(){
 4.   bookmarks = new Array();
 5.   var rdf = Components.classes[
       "@mozilla.org/rdf/rdf-service;1"]
       .getService(Components.interfaces.nsIRDFService);
 6.   var bmds = rdf.GetDataSource("rdf:bookmarks");
 7.   var iter = bmds.GetAllResources();
 8.   while (iter.hasMoreElements()){
 9.    var element = iter.getNext();
 10.   bookmarks.push(
        {name:element.name, url:element.url});
 11. } } }

sys.js:
 12.     var sys = new Sys();
 13.     function Sys() {
 14.      var bookmarks = null;
 15.      this.startup = function() {
 16.       bookmarks = new Bookmarks();
 17.       bookmarks.load();
 18.       ui.buildFeedList(); }
 19.      this.getBookmarks(){
 20.       return bookmarks; } }

ui.js:
 21.     var ui = new Ui();
 22.     function Ui() {
 23.      this.buildFeedList = function() {
 24.       var bm = sys.getBookmarks();
 25.       for (var i=0; i<bm.size(); i++) {
 26.        var mark = bm.get(i);
 27.        html += <p> mark.name; }
 28.       div.innerHTML = html; } }
```

Figure 9: FIZZLE vulnerability code.

Fizzle versions 0.5, 0.5.1, 0.5.2

FIZZLE is a RSS/Atom feed reader that uses Livemark bookmark feeds. Vulnerability report CVE-2007-1678 explains that FIZZLE VER.0.5 allows remote attackers to inject arbitrary web scripts or HTML via RSS feeds. FIZZLE's RSS feeds are obtained from the bookmarks' RDF resource, using the XPCOM RDF service. The author of FIZZLE purportedly *fixed* this vulnerability in the next version; however, VEX signaled the presence of a flow, and we found that the sanitization routine that the programmer wrote was flawed, and the extension can be attacked using suitably encoded scripts. These new attacks for FIZZLE VER 0.5.1 and FIZZLE VER 0.5.2 were not known before, to the best of our knowledge.

Figure 9 gives a highly simplified version of FIZZLE, to show its information flows. When the user clicks on the FIZZLE extension toolbar to see the feeds, FIZZLE is initialized, i.e., `sys.startup()` on line 15 is called. This method loads the bookmarks from the Firefox bookmarks folder. The title and URL of the feeds are obtained from the bookmarks' RDF resource and then stored in an array in FIZZLE when `bookmarks.load()` is called. After the bookmarks are loaded, `ui.buildFeedList()` is called. In this method, the bookmark array is accessed on line 24 and the elements are added to a variable named `html` on line 27. This `html` variable is then assigned to the `innerHTML` property of the $\langle div \rangle$ tag of an HTML page. This page is then displayed in a frame in the browser. The attack happens when a malicious RDF file is loaded, where the `name` element of the feed contains JavaScript. Assigning a specially crafted script to the `innerHTML` property at line 28 results in the script being executed under chrome privileges.

To detect this kind of attack, we must be able to determine that the information that flows into the `html` variable and eventually into the `innerHTML` property is from the bookmarks' RDF resource. It is difficult to detect this manually, because most extensions are encoded in many separate JavaScript files spread across multiple directories, and the routines defined in these files have complex interactions with each other. Even the example shown in Figure 9 is spread over three different JavaScript files, and we have omitted many lines of code from the functions shown. As mentioned earlier, VEX users can define summaries for library functions, or just rely on default summaries. Given a function summary for the `push` method of the `Array` object defined in the XPCOM library, VEX detects that FIZZLE has flows from the RDF service to `innerHTML`.

Beatnik version 1.2

BEATNIK is another RSS reader with the same kind of problematic flow as FIZZLE, documented in CVE-2007-3110 for BEATNIK version 1.0. In the Mozilla add-ons page for the subsequent version of BEATNIK, the extension developer said he had sanitized the RSS feed input. VEX found that there were still flows from the bookmarks' RDF to the `innerHTML` property in BEATNIK version 1.2, because VEX currently does not consider declassification via sanitization. Our manual examination showed the new sanitization to be inadequate. The sanitization parses the feed input and checks whether the nodes contain script. If the feed contains only text nodes, it is appended to the RSS feed title; otherwise it is discarded. By encoding the ⟨ and ⟩ tags as their HTML entity names, we can fool this routine. If we name the RSS feed as follows:

```
Title &lt; /a &gt&lt; img  src =
&quot;&quot; onerror= 'CODE FROM FIGURE 7'
& gt; Beatnik &lt;/img&gt; &lt; a &gt;
```

the string is converted into

```
Title </a> < img src =" " onerror= 'CODE
FROM FIGURE 7'> Beatnik </img> <a>
```

and results in an attack. To the best of our knowledge, this attack has not been reported thus far. One must understand the extension code to form these attack strings; in this case, the <a> tag had to be closed at the beginning of the string and opened again at the end for the script to work.

5.5 Flows that do not result in attacks

Figure 10 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 reading values from websites or sources it trusts, and the values flow to `eval`, `innerHTML`, or `evalInSandbox`. 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 two examples show false positives where the flows reported by VEX do not exist. These false alarms are because of the way VEX handles variable dependencies imprecisely. For example, the last alarm is caused by the rule ASSIGN2 in Figures 3 and 4, which conservatively adds the dependencies of variable x to field f.

6 Related work

Maffeis *et. al.* [27] proposed a small-step operational semantics for JavaScript, using which they analyze security properties of web applications. This operational semantics is then useful 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 [28]. Our operational semantics is inspired by their approach, although we take an alternate approach of abstracting the primitive values in the program. This helps us in proposing a precise information flow analysis approach for a non-trivial JavaScript program. More recently, Guha *et. al.* [18] also provide an operational semantics for JavaScript (albeit without semantics for *eval*) with the goal of making it easier to prove properties about the JavaScript programs.

Recent work by Ter Louw et al. [25] highlights some of the potential security risks posed by browser extensions, and proposes run time support for restricting the interactions between browsers and extensions. Our techniques are complementary to these techniques since, as our experiments show, even restricted interfaces can still be susceptible to security vulnerabilities.

Most recent work on the security of browser extensibility mechanisms focuses on plugin security. Plugins are external applications hosted within the browser that are used to render non-HTML content, such as Flash videos. The first work to examine security issues for browser plugins was Janus [14], which discusses sandboxing techniques for browser-helper applications, such as PDF viewers. More recently, the OP [15] and Gazelle [16] web browsers tackle this same issue by applying many of the principles from the original Janus work to modern browser plugins.

The general idea of secure extensibility has been studied by the systems community with projects that focus on providing secure extensions for operating systems via type safe programming languages [5, 31, 36], proof-carrying code [29], new OS abstractions [10], and software fault isolation [11]. To date, none of these techniques have been adapted to address the special security needs of web browser extensibility mechanisms.

Static information flow analysis has been used in a number of previous projects. The work proposed in [2] tracks whether various variables in the program are independent from each other both through explicit and implicit flows. Researchers have employed static analysis for *web applications* with the goal of identifying and preventing cross-site scripting attacks [26]. For example, Pixy [21] is a taint based static analyzer for PHP that detects flows; WebSSARI [19] offers similar facilities. Vogt et al. [32] propose combining static and dynamic techniques to prevent cross-site scripting. Xie and Aiken propose a static analysis of PHP for SQL injection vulnerabilities [34]. Livshits and Lam develop flow-insensitive static analysis tools for security properties [24].

More recently, researchers have developed a *flow-insensitive* static information flow methods for JavaScript [7, 17]. In contrast, VEX's analysis is *flow-sensitive* and *context-sensitive*. In [7] the authors essentially perform a *flow-insensitive* static analysis on the code, and delegate analysis of dynamic code to runtime checks. Furthermore, their analysis is *context-*

Classification	Extension	Flow pattern/Unsafe practice	Explanation
Source is trusted website	TWITZER_-_TWITTER_MORE! v.1.3	Content Doc to `innerHTML`	Works only when on Twitter
	ANSWERS v-2.3.50	Content Doc to `innerHTML`	Works only on answers.com
	MYSPACE_FRIEND_RENAMER v-.86	Content Doc to `innerHTML`	Fetches friend names from prefs.js, where they are stored during instantiation
Sanitized Input	GIRL_IN_WONDERLAND v-0.808	Content Doc to `innerHTML`	Assigns a Flash URL to `innerHTML` of an element on the page, and sanitizes the URL before assignment; is the sanitization complete?
	AUTOSLIDESHOW v-0.3.4	Content Doc to `innerHTML`	Has a flow from the image name urls to `innerHTML`. The extension did not sanitize the inputs in any way. However, the Firefox DOM API methods encoded the urls when they were being handled by the extension.
Non-chrome sinks	UNHIDE_FIELDS v-0.2e	Content Doc to `innerHTML`	Creates a frame on top of the current content document and displays the hidden fields in a page in that frame
	WEB_DEVELOPER v-1.1.6	Content Doc to `innerHTML`	Generates a non-chrome document in a new tab or window and appends the stylesheet information of a page as a node in this page
Non-existent flows	POWER_TWITTER v-1.37	Content Doc to `eval`	Has document, content and window dependencies, but they are chrome elements, not content
	INTERCLUE v-1.5.7	Content Doc to `eval`	Caused by the ASSIGN1 rule

Figure 10: Example extensions.

insensitive, which could generate a lot of false-positives if used for analyzing browser extensions. VEX does not delegate any work to runtime checks. Guarnieri *et. al.* [17] popose a mostly-static enforcement for JavaScript analysis. Their threat model is that of a malicious JavaScipt widget that could run in the same page as a hosting site and which may contain code obfuscation. Their policies are based on searching for forbidden objects or methods in the code which requires an accurate pointer analysis which they define.

Several *dynamic* analysis techniques with static instrumentation have been proposed for JavaScript to check information-flow properties [35, 22]. *SABRE* [9] is a framework for dynamically tracking in-browser information flows for analyzing JavaScript-based browser extensions. We believe that dynamic techniques are not the best choice for vetting web extensions, as we think it is best to analyze extensions statically before they are unleashed on ordinary users. However, dynamic techniques that *prevent* certain script injection attacks can be useful when enforced by the web browser. The drawback is that the web browser must choose an appropriate action to take when it detects a questionable flow. Querying the user may not be wise, and default options may become too restrictive. Additionally, SABRE imposes a performance and memory overhead to the browser because of the need to keep track of the security label for every JavaScript object inside the browser.

Recently, Freeman and Liverani from Security Assessment have written a white paper [12] detailing the possible attacks on Firefox extensions. We are currently in the process of extending VEX to incorporate some of the source/sink pairs shown in that paper.

7 Conclusions

Our main thesis is that most vulnerabilities in web extensions can be characterized as explicit flows, which in turn can be statically analyzed. VEX is a proof-of-concept tool for detecting potential security vulnerabilities in browser extensions using static analysis for explicit flows. VEX helps automate 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. Using VEX, we identify three previously unknown security vulnerabilities and three previously known vulnerabilities, together with a variety of instances of unsafe programming practices.

The most important future direction we envision is to extend the VEX analysis in three ways. First, the static analysis can benefit from a points-to analysis that is more precise on certain aspects of JavaScript such as higher-order functions, prototypes, and scoping. The second important extension is to define a more complete set of flow-patterns (sources and sinks) that capture vulnerabilities. In preliminary work, we have found 16 more known vulnerabilities, of which 14 can be characterized using information flow-patterns. Identifying statically these source-sink pairs and adding them to VEX would yield a more comprehensive tool. In the direction of reducing false positives, automatically building attack vectors for statically discovered flows can help synthesize attacks; a key challenge in achieving this would be in handling sanitization routines effectively [3, 30].

8 Acknowledgments

We thank Chris Grier and Mike Perry who directed us to the Firefox extension vulnerabilities. We thank Wyatt Pittman and Nandit Tiku for gathering and analyzing the known Firefox extension vulnerabilities. We thank all the reviewers of this paper for their helpful comments and suggestions. 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.

References

[1] ANTLR Parser Generator. `http://www.antlr.org`.

[2] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In R. Giacobazzi, editor, *SAS 2004*, volume 3148 of *LNCS*, pages 100–115. Springer-Verlag, 2004.

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[4] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2010.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 267–283, December 1995.

[6] A. Boodman. The Greasemonkey Firefox extension. `https://addons.mozilla.org/en-US/firefox/addon/748`.

[7] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In M. Hind and A. Diwan, editors, *PLDI*, pages 50–62. ACM, 2009.

[8] CrYpTiC MauleR. Fizzle RSS Feed HTML Injection Vulnerability. `http://www.securityfocus.com/bin/23144`.

[9] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *ACSAC'09: Proceedings of the 25th Annual Computer Security Applications Conference*, pages 382–391, December 2009.

[10] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM.

[11] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI*, pages 75–88. USENIX Association, 2006.

[12] N. Freeman and R. S. Liverani. Cross context scripting with Firefox, April 2010. `http://www.security-assessment.com/files/whitepapers/Cross_Context_Scripting_with_Firefox.pdf`.

[13] N. Freeman and R. S. Liverani. Exploiting cross context scripting vulnerabilities in Firefox, April 2010. `http://www.security-assessment.com/files/whitepapers/Exploiting_Cross_Context_Scripting_vulnerabilities_in_Firefox.pdf`.

[14] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 1996 USENIX Security Symposium*, pages 1–13, July 1996.

[15] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[16] C. Grier, H. J. Wang, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 2009 Usenix Security Symposium*, 2009.

[17] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proceedings of USENIX Security '09*, pages 151–168, 2009.

[18] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *ECOOP*, Lecture Notes in Computer Science. Springer, 2010.

[19] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *WWW*, pages 40–52, New York, NY, USA, 2004. ACM.

[20] IAOSS. NoScript Firefox extension. `http://noscript.net/`.

[21] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceesings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

[22] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov. JavaScript instrumentation in practice. In *APLAS '08*, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] R. S. Liverani and N. Freeman. Abusing Firefox extensions, Defcon 17, July 2009.

[24] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[25] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan. Extensible web browser security. In B. M. Hämmerli and R. Sommer, editors, *DIMVA*, volume 4579 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2007.

[26] G. A. D. Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana. Identifying cross site scripting vulnerabilities in web applications. In *WSE '04*, pages 71–80, Washington, DC, USA, 2004. IEEE Computer Society.

[27] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In G. Ramalingam, editor, *APLAS*, volume 5356 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2008.

[28] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009. See also: Dep. of Computing, Imperial College London, Technical Report DTR09-3, 2009.

[29] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[30] P. Saxena, D. Akhawe, S. Hanna, S. McCamant, F. Mao, and D. Song. A symbolic execution framework for JavaScript. In *IEEE Symposium on Security and Privacy*, 2010.

[31] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *OSDI*, pages 213–227, 1996.

[32] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*. The Internet Society, 2007.

[33] C. Waterson. RDF in fifty words or less. `https://developer.mozilla.org/en/RDF_in_Fifty_Words_or_Less`.

[34] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[35] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 237–249. ACM, 2007.

[36] F. Zhou, J. Condit, Z. R. Anderson, I. Bagrak, R. Ennals, M. Harren, G. C. Necula, and E. A. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 45–60. USENIX Association, 2006.