Published in ACSAC'09: Proceedings of the 25th Annual Computer Security Applications Conference, Honolulu, Hawaii, December 2009

Analyzing Information Flow in JavaScript-based Browser Extensions

Mohan Dhawan and Vinod Ganapathy
Department of Computer Science, Rutgers University

Abstract

JavaScript-based browser extensions (JSEs) enhance the core functionality of web browsers by improving their look and feel, and are widely available for commodity browsers. To enable a rich set of functionalities, browsers typically execute JSEs with elevated privileges. For example, unlike JavaScript code in a web application, code in a JSE is not constrained by the same-origin policy. Malicious JSEs can misuse these privileges to compromise confidentiality and integrity, e.g., by stealing sensitive information, such as cookies and saved passwords, or executing arbitrary code on the host system. Even if a JSE is not overtly malicious, vulnerabilities in the JSE and the browser may allow a remote attacker to compromise browser security.

We present Sabre (Security Architecture for Browser Extensions), a system that uses in-browser information-flow tracking to analyze JSEs. Sabre associates a label with each in-memory JavaScript object in the browser, which determines whether the object contains sensitive information. Sabre propagates labels as objects are modified by the JSE and passed between browser subsystems. Sabre raises an alert if an object containing sensitive information is accessed in an unsafe way, e.g., if a JSE attempts to send the object over the network or write it to a file. We implemented Sabre by modifying the Firefox browser and evaluated it using both malicious JSEs as well as benign ones that contained exploitable vulnerabilities. Our experiments show that Sabre can precisely identify potential information flow violations by JSEs.

1. Introduction

Modern web browsers support an architecture that lets third-party extensions enhance the core functionality of the browser. Such extensions enhance the look and feel of the browser and help render rich web content, such as multimedia. Extensions are widely available for commodity browsers as plugins (*e.g.,* PDF readers, Flash players, ActiveX), browser helper objects (BHOs) and add-ons.

This paper concerns *JavaScript-based browser extensions* (JSEs). Such extensions are written primarily in JavaScript, and are widely available and immensely popular (as "add-ons") for Firefox [4] and related tools, such as Thunderbird. Notable examples of JSEs for Firefox include Greasemonkey [5], which allows user-defined scripts to customize how web pages are rendered, Firebug [3], a JavaScript development environment, and NoScript [8], a JSE that aims to improve security by blocking script execution from certain websites. Other browsers like Internet Explorer and Google Chrome also support extensions (*e.g.,* scriptable plugins and ActiveX controls) that contain or interact with JavaScript code.

However, recent attacks show that JSEs pose a threat to browser security. Two factors contribute to this threat:

(1) Inadequate sandboxing of JavaScript in a JSE. Unlike JavaScript code in a web application, which executes with restricted privileges [9], JavaScript code in a JSE executes with the privileges of the browser. JSEs are not constrained by the same-origin policy [38], and can freely access sensitive entities, such as the cookie store and browsing history. For instance, JavaScript in a JSE is allowed to send an `XMLHttpRequest` to *any* web domain. Even though JavaScript only provides restricted language-level constructs for I/O, browsers typically provide cross-domain interfaces that enable a JSE to perform I/O. For example, although JavaScript does not have language-level primitives to interact with the file system, JSEs in Firefox can access the file system via constructs provided by the XPCOM (cross-domain component object model) interface [7]. Importantly, *these features are necessary* to create expressive JSEs that support a rich set of functionalities. For example, JSEs that provide cookie/password management functionality rely critically on the ability to access the cookie/password stores.

However, JSEs from untrusted third parties may contain malicious functionality that exploits the privileges that the browser affords to JavaScript code in an extension. Examples of such JSEs exist in the wild. They are extremely easy to create and can avoid detection using stealth techniques [11, 13–15, 18, 41]. Indeed, we wrote several such JSEs during the course of this project.

(2) Browser and JSE vulnerabilities. Even if a JSE is not malicious, vulnerabilities in the browser and in JSEs may allow a malicious website to access and misuse the privileges of a JSE [12, 35, 39, 40, 45]. Vulnerabilities in older versions of Firefox/Greasemonkey allowed a remote attacker to access the file system on the host machine [35, 45]. Similarly, vulnerabilities in Firebug [12, 39] allowed remote attackers to execute arbitrary commands on the host machine using exploits akin to cross-site scripting. These attacks exploit subtle interactions between the browser and JSEs.

While there is much prior work on the security of untrusted browser extensions such as plugins and BHOs (which are distributed as binary executables) particularly in the context of spyware [22, 30, 31], there is relatively little work on analyzing the security of JSEs. Existing techniques to protect against an untrusted JSE rely on load-time verification of the integrity of the JSE, *e.g.,* by ensuring that scripts are digitally signed by a trustworthy source.

However, such verification is agnostic to the code in a JSE and cannot prevent attacks enabled by vulnerabilities in the browser or the JSE. Ter-Louw *et al.* [41] developed a runtime agent to detect malicious JSEs by monitoring XPCOM calls and ensuring that these calls conform to a user-defined security policy. Such a security policy may, for instance, prevent a JSE from accessing the network after it has accessed browsing history. Unfortunately, XPCOM-level monitoring of JSEs is too coarse-grained and can be overly restrictive. For example, one of their policies disallows XPCOM calls when SSL is in use, which may prevent some JSEs from functioning in a `https` browsing session. XPCOM-level monitoring can also miss attacks, *e.g.,* a JSE may disguise its malicious actions so that they appear benign to the monitor (in a manner akin to mimicry attacks [43]).

This paper presents *Sabre*, a system that uses *in-browser information-flow tracking* to analyze JSEs. Sabre associates each in-memory JavaScript object with a label that determines whether the object contains sensitive information. Sabre modifies this label when the corresponding object is modified by JavaScript code (contained both in JSEs and web applications). Sabre raises an alert if a JavaScript object containing sensitive data is accessed in an unsafe way, *e.g.,* if a JSE attempts to send a JavaScript object containing sensitive data over the network or write it to a file. In addition to detecting such confidentiality violations, Sabre also uses the same mechanism to detect integrity violations, *e.g.,* if a JSE attempts to execute a script received from an untrusted domain with elevated privileges.

Sabre differs from prior work [22] that uses information-flow tracking to analyze plugins and BHOs because it tracks information flow at the level of *JavaScript instructions* and does so *within the browser*. In contrast, prior work on plugin security tracks information flow at the *system level* by tracking information flow at the granularity of *machine instructions*. These differences allow Sabre to report better forensic information with JSEs because an analyst can explain flow of information at the granularity of JavaScript objects and instructions rather than at the granularity of memory words and machine instructions. For example, prior work [22] required the system-level information-flow tracker to have access to OS-aware views in order to attribute suspicious actions to specific plugins and BHOs. In contrast, Sabre can readily attribute suspicious actions to the JSEs that performed these actions. Finally, Sabre can be implemented by modifying the web browser and does not require the browser to execute in specialized information-flow tracking environments (*e.g.,* a modified system emulator).

Sabre employs techniques similar to prior work on JavaScript-level information flow tracking [16, 42]. However, it differs in two ways. First, Sabre precisely tracks information flows across different browser subsystems, including the DOM, local storage and the network. In contrast, prior work only tracked information flows within the JavaScript interpreter and provided rudimentary support for label propagation across the DOM [42]. Second, Sabre can *declassify or endorse information flows*. This support is critical for the analysis of JSEs because benign JSEs often contain flows from sensitive sources to low-sensitivity sinks. To our knowledge, prior work on JavaScript-level information flow has not needed such support.

To summarize, the main contributions of this paper are:

- **Sabre, an information flow tracker for JSEs.** We discuss the techniques used to implement information flow tracking in a web browser and the heuristics used to achieve precision (Section 3). Sabre handles explicit information flows, some forms of implicit flows, as well as cross-domain flows. We have implemented a prototype of Sabre in Firefox.
- **Evaluation on 24 JSEs.** We evaluated Sabre using malicious JSEs as well as benign ones that contained exploitable vulnerabilities. In these cases, Sabre precisely identified information flow violations. We also tested Sabre using benign JSEs. In these experiments, Sabre precisely identified potentially suspicious flows that we manually analyzed and whitelisted (Section 4).

We chose Firefox as our implementation and evaluation platform because of the popularity and wide availability of JSEs for Firefox. However, JSEs pose a security threat even in privilege-separated browser architectures (*e.g.,* [6, 27, 44]) for the same reasons as outlined earlier. The techniques described in this paper are therefore relevant and applicable to such browsers as well.

2. Background and Motivating Examples

Writing browser extensions in JavaScript offers a number of advantages that will ensure that JSEs remain relevant in future browsers as well. JavaScript has emerged as the *lingua franca* of the Web and is supported by all major browsers. It offers several primitives that are ideally suited for web browsing (*e.g.,* handlers for user-generated events, such as mouse clicks and keystrokes) and allow easy interaction with web applications (*e.g.,* primitives to access the DOM). JSEs can be written by developers with only a rudimentary knowledge of JavaScript and can readily be modified by others, which in turn allows for rapid prototyping. This is in contrast to plugins and BHOs, which are developed in low-level languages against browser-specific interfaces and are distributed as binary executables. Finally, because support for JavaScript is relatively stable as browsers evolve, JSEs can be readily ported across platforms and browser versions. Indeed, many of these benefits apply to extensions written in any scripting language, and have motivated several software systems to adopt such extension models, *e.g.,* AppleScript and Adobe Lightroom.

To allow easy access to browser resources and to support a rich set of functionalities, browsers execute JSEs with elevated privileges. However, doing so renders the browser susceptible to attacks via JSEs. Malicious JSEs may exploit elevated privileges to steal sensitive data or snoop on user activity. Worse, benign JSEs from trusted vendors may contain vulnerabilities that, in combination with browser vulnerabilities, may be exploited by remote attackers. The problem is exacerbated by the lack of good environments and tools, such as static bug finders, for code development in JavaScript. Moreover, because subtle bugs only manifest when a JSE is used with certain versions of the browser, comprehensive testing of JSEs for security vulnerabilities is

```
 1. <script type="text/javascript">
 2. window._GM_xmlhttpRequest = null;
 3. function trapGM(...) {
 4.   window._GM_xmlhttpRequest = window.GM_xmlhttpRequest;
 5.   ...
 6. }
 7. function checkGM() {
 8.   if (window._GM_xmlhttpRequest) {
 9.     window._GM_xmlhttpRequest(
10.       {method: 'GET', url: 'file:///c:/boot.ini',
11.        onload: function(Response) {
12.          document.formname.textfield.value
                           = Response.responseText;
13.       }});
14.   }
15. }
16. if (typeof window.addEventListener != 'undefined') {
17.    window.watch('GM_apis', trapGM);
18.    window.addEventListener('load', checkGM, true);
19. }
20. </script>
```

Figure 1. Example of malicious JavaScript code that exploits the Greasemonkey vulnerability to read the contents of `boot.ini` from disk (adapted from [35]).

often difficult.

The remainder of this section presents motivating examples that demonstrate how JSEs can compromise confidentiality and integrity. The first example shows how a remote attacker can exploit vulnerabilities in an otherwise benign JSE, while the second example presents a malicious JSE. In each case, we also describe how information-flow tracking, as implemented in Sabre, would have discovered the attack. We refer the reader to the companion technical report [21] for further examples.

Greasemonkey/Firefox Vulnerability. Greasemonkey is a popular JSE that allows user-defined scripts to make changes to web pages on the fly. For example, a user could register a script with Greasemonkey that would customize the background of web pages that he visits. Greasemonkey exports a set of APIs (prefixed with "GM") that user-defined scripts can be programmed against. These APIs execute with elevated privileges because user-defined scripts must have the ability to read and modify arbitrary web pages. For example, the `GM_xmlhttpRequest` API allows a user-defined script to execute an `XMLHttpRequest` to an arbitrary web domain, and is not constrained by the same-origin policy.

Unfortunately, a combination of vulnerabilities in older versions of Greasemonkey (CVE-2005-2455) and Firefox (CVE-2006-1734) allowed scripts on a web page to capture references to `GM` API functions (`GM_xmlhttpRequest` in particular) using the JavaScript `watch` function, as shown in Figure 1. When the page loads, the script uses this reference to issue a `GET` request to read the contents of the `boot.ini` file from the local file system. Although the script in Figure 1 simply modifies the DOM to store the contents of the `boot.ini` file, it could instead use a `POST` to transmit this data over the network to a remote attacker.

Information-flow tracking as implemented in Sabre detects this attack because sensitive user data (`boot.ini`) is accessed in unsafe ways. In particular, Sabre marks as sensitive all data that a JSE reads from a pre-defined set of sensitive sources, including the local file system. The call to `window._GM_xmlhttpRequest` (line 9 in Figure 1) executes JavaScript code from Greasemonkey to access the local file system. Consequently, `Response.responseText`, which this function returns, is also marked sensitive. In turn, the DOM node that stores this data is also marked as sensitive because of the assignment on line 12. Sabre raises an alert when the browser attempts to send contents of the DOM over the network, *e.g.,* when the user clicks a "submit" button.

```
function do_sniff() {
  var hesla = window.content.document.getElementsByTagName("input");
  data = "";
  for (var i = 0; i < hesla.length; i++) {
    if (hesla[i].value != "") {
      ...
      data += hesla[i].type + ":" + hesla[i].name
        + ":" + hesla[i].value + "\n";
      ...
    }
  }
  // the rest of the code sends 'data' via an email message.
}
```

Figure 2. A snippet of code from FFsniFF, a malicious JSE.

This example illustrates how a malicious website can exploit JSE/browser vulnerabilities to steal confidential user data. It also illustrates the need to precisely track security labels across browser subsystems. For instance, Sabre detects the above attack because it also modifies the browser's DOM subsystem to store labels with DOM nodes. Doing so allows Sabre to determine whether a sensitive DOM node is transmitted over the network. An approach that only tracks security labels associated with JavaScript objects (*e.g.,* [16, 42]) will be unable to precisely detect this attack.

A Malicious JSE. FFsniFF (Firefox Sniffer) [13] is a malicious JSE that, if installed, attempts to steal user data entered on HTML forms. When a user "submits" an HTML form, FFsniFF iterates through all non-empty `input` fields in the form, including password entries, and saves their values. It then constructs SMTP commands and transmits the saved form entries to the attacker (the attack requires the vulnerable host to run an SMTP server). FFsniFF also attempts to hide itself from the user by exploiting a vulnerability in the Firefox extension manager (CVE-2006-6585) to delete its entry from the add-ons list presented by Firefox.

Figure 2 presents a simplified snippet of code from FFsniFF and illustrates the ease with which malicious extensions can be written. Sabre detects FFsniFF because it considers all data received from form fields on a web page as sensitive. This sensitive data is propagated to both the array `hesla` and the variable `data` via a series of assignment statements. Sabre raises an alert when FFsniFF attempts to send the contents of the sensitive `data` variable along with SMTP commands over an output channel (a low-sensitivity sink) to the SMTP server running on the host machine.

3. Tracking Information Flow with Sabre

This section describes the design and implementation of Sabre. We had three goals:

(**1**) **Monitor all JavaScript execution.** Sabre must monitor all JavaScript code executed by the browser. This includes code in web applications, JSEs, as well as JavaScript code executed by the browser core, *e.g.,* code in browser menus

and XUL elements [10].
Monitoring all JavaScript code is important for two reasons. First, an attack may involve JavaScript code in multiple browser subsystems. For example, a malicious JSE may copy data into a XUL element, which may then be read and transmitted by JavaScript in a web application. In such cases, it is important to track the flow of sensitive data through the JSE to the XUL element and into the web application. Second, JSEs may often contain code, such as scripts in XUL overlays, that may be included into the browser core. Such code often interacts with JavaScript code in a web application. For example, an overlay may implement a handler that is invoked in response to an event raised by a web application. It is key to track information flows through code in overlays because overlays from untrusted JSEs may be malicious/vulnerable.

(**2**) **Ease action attribution.** When Sabre reports an information flow violation by a JSE, an analyst may need to determine whether the violation is because of an attack or whether the offending flow is part of the advertised behavior of the JSE. In the latter case, the analyst must whitelist the flow. For example, PwdHash [37] is a JSE that scans and modifies passwords entered on web pages. This behavior may be considered malicious if performed by an untrusted JSE. However, an analyst may choose to trust PwdHash and whitelist this flow. To do so, it is important to allow for easy action attribution, *i.e.,* an analyst must be able to quickly locate the JavaScript code that caused the violation and determine whether the offending flow must be whitelisted.

(**3**) **Track information flow across browser subsystems.** JavaScript code in a browser and its JSEs interacts heavily with other subsystems, such as the DOM and persistent storage, including cookies, saved passwords, and even the local file system. Sabre must precisely monitor information flows across these subsystems because attacks enabled by JSEs (*e.g.,* those illustrated in Section 2) often involve multiple browser subsystems.

We implemented Sabre by modifying SpiderMonkey, the JavaScript interpreter in Firefox, to track information flow. We modified SpiderMonkey's representation of JavaScript objects to include security labels. We also enhanced the interpretation of JavaScript bytecode instructions to modify labels, thereby propagating information flow. We also modified other browser subsystems, including the DOM subsystem (*e.g.,* HTML, XUL and SVG elements) and XPCOM, to store and propagate security labels, thereby allowing information flow tracking across browser subsystems. This approach allows us to satisfy our design goals. All JavaScript code is executed by the interpreter, thereby ensuring complete mediation even in the face of browser vulnerabilities, such as those discussed in Section 2. Moreover, associating security labels directly with JavaScript objects and tracking these labels within the interpreter and other browser subsystems makes our approach robust to obfuscated JavaScript code, *e.g.,* as may be found in drive-by-download websites that attempt to exploit browser and JSE vulnerabilities. Finally, the interpreter can readily identify the source of the JavaScript bytecode currently being interpreted, thereby allowing for easy action attribution.

Although Sabre's approach of using browser modifications to ensure JSE security is not as readily portable as, say, language restrictions [1, 2, 33], this approach also ensures compatibility with legacy JSEs. For example, Adsafe [1] would reject JSEs containing dynamic code generation constructs, such as `eval`; in contrast, Sabre allows arbitrary code in a JSE, but instead tracks information flow. An information-flow tracker based on JavaScript instrumentation will likely be portable across browsers; we plan to investigate such an approach in future work.

3.1. Security Labels

Sabre associates each in-memory JavaScript object with a pair of security labels. One label tracks the flow of sensitive information while the second tracks the flow of low-integrity information (to detect, respectively, violations of confidentiality and integrity). We restrict our discussion to tracking flows of sensitive information because confidentiality and integrity are largely symmetric.

Each security label stores three pieces of information: (i) a sensitivity level, which determines whether the object associated with the label stores sensitive information; (ii) a Boolean flag, which determines whether the object was modified by JavaScript code in a JSE; and (iii) the name(s) of the JSE(s) and web domains that have modified the object. The sensitivity level is used to determine possible information flow violations, *e.g.,* if data derived from a sensitive source is written to a low-sensitivity sink. However, Sabre raises an alert only if the object was modified by a JSE. In this case, Sabre reports the name(s) of the JSE(s) that have modified the object. For example, in Figure 1, the DOM node that stores the response from the `_GM_xmlhttpRequest` call is marked sensitive. Further, the data contained in the node is modified by executing code contained in Greasemonkey, via the return value from `_GM_xmlhttpRequest`. Consequently, Sabre raises an alert when the browser attempts to transmit the DOM node via HTTP, *e.g.,* when the user submits a form containing this node.

Sabre's policy of raising an alert only when an object is modified by a JSE is key to avoiding false positives. Recall that Sabre tracks the execution of *all* JavaScript code, including code in web applications and in the browser core. Although such tracking is necessary to detect attacks via compromised/malicious files in the browser core, *e.g.,* overlays from malicious JSEs, it can also report confidentiality violations when sensitive data is accessed in legal ways, such as when JavaScript in a web application accesses cookies. Such accesses are sandboxed using other mechanisms, *e.g.,* the same-origin policy. We therefore restrict Sabre to report an information-flow violation only when a sensitive object modified by JavaScript code in a JSE (or overlay code derived from JSEs) is written to a low-sensitivity sink.

Security labels in Sabre allow for fine-grained information flow tracking. Sabre associates a security label with each JavaScript object, including objects of base type (*e.g.,* `int`, `bool`), as well as with complex objects such as arrays and compound objects with properties. For complex JavaScript objects, Sabre associates additional labels, *e.g.,* each element of an array and each property of a compound object is associated with its own security label. In particular, an object `obj` and its property `obj.prop` each have their own security label.

Sabre stores security labels by directly modifying the interpreter's data structures that represent JavaScript objects. Doing so considerably eases the design of label propagation rules for a prototype-based language such as JavaScript. A JavaScript object inherits all the properties of its ancestor prototypes. Therefore an object's properties may not directly be associated with the object itself. For example, an object `obj` may access a property `obj.prop`, which in turn may result in a chain of lookups to locate the property `prop` in an ancestor prototype of `obj`. In this case, the sensitivity-level of `obj.prop` is the sensitivity of the value stored in `prop`. Sabre stores the label of the property `prop` with the in-memory representation of `prop`. Its label can therefore be accessed conveniently, even if an access to `prop` involves a chain of multiple prototype lookups to locate the property. Moreover, objects in JavaScript are passed by reference. Therefore, any operations that modify the object via a reference to it, such as those in a function to which the object is passed as a parameter, will also modify its label appropriately when the interpreter accesses the in-memory representation of that object.

JavaScript in a browser closely interacts with several browser subsystems. Notably, the browser provides the `document` and `window` interfaces via which JavaScript code can interact with the DOM, *e.g.,* a JSE can access and modify `window.location`. However, such browser objects are not stored and managed by the JavaScript interpreter. Rather, each access to a browser object results in a cross-domain call that gets/sets the value of the browser object. To store security labels for such objects, Sabre also modifies the browser's DOM subsystem to store security labels. Each DOM node has an associated security label. This label is accessed and transmitted by the browser to the JavaScript interpreter when the DOM node is accessed in a JSE.

In addition to the DOM, cross-domain interfaces such as XPCOM allow a JSE to interact with other browser subsystems, such as storage and networking. For example, the following snippet uses XPCOM's cookie manager.

```
var cookieMgr =
          Components.classes["@mozilla.org/cookiemanager;1"].
          getService(Components.interfaces.nsICookieManager);
var e = cookieMgr.enumerator;
```

In this case, the reference to `enumerator` is resolved via a cross-domain call to the cookie manager. Sabre must separately manage the security labels of `cookieMgr` and those of its properties because `cookieMgr` is not a JavaScript object. Sabre assigns a default security label to cross-domain objects (described in Section 3.2). It also ensures that properties that are resolved via cross-domain calls inherit the labels of their parent objects, *e.g.,* `cookieMgr.enumerator` inherits the label of `cookieMgr`.

3.2. Sources and Sinks

Sabre detects flows from sensitive sources to low-sensitivity sinks. We consider several sensitive sources (Figure 3 in [21]) which primarily deal with access to DOM elements, as well as sources enabled by cross-domain access, including those that allow access to persistent storage. Any data received over these interfaces is considered sensitive.

Low-sensitivity sinks accessible from the JavaScript interpreter include the file system and the network (Figure 4 in [21]). In addition to modifying the JavaScript interpreter to raise an alert when a sensitive object is written to a low-sensitivity sink, Sabre also modifies the browser's document interface to raise an alert when a DOM node that stores sensitive data derived from a JSE is sent over the network. For example, Sabre raises an alert when a `form` or a `script` element that contains sensitive data (*i.e.,* data derived from the cookie or password store) is transmitted over the network.

The browser itself may perform several operations that result in information flows from sensitive sources to low-sensitivity sinks. For example, the file system is listed both as a sensitive source and a low-sensitivity sink. This is because a JSE may potentially leak confidential data from a web application by storing this data on the file system, which may then be accessed by other JSEs or malware on the host machine. However, the browser routinely reads and writes to the file system, *e.g.,* bookmarks and user preferences are read from the file system when the browser starts and are written back to disk when the browser shuts down. To avoid raising an alert on such benign flows, Sabre reports an information-flow violation only if an object is written to by a JSE (as discussed in Section 3.1). Consequently, it does not report an alert on benign flows, such as the browser reading and writing user preferences. Even so, a benign JSE may contain instances of flows from sensitive sources to low-sensitivity sinks as part of its advertised behavior. Disallowing such flows may render the JSE dysfunctional. In Section 3.4, we discuss how Sabre handles such flows via whitelisting.

While sources and sinks listed above help detect confidentiality-violating information-flows, a similar set of low-integrity sources and high-integrity sinks can also be used to detect integrity violations. In this case, Sabre detects information-flows from low-integrity sources, *e.g.,* the network, to high-integrity sinks, *e.g.,* calls to `nsIProcess`, which can be used to start a process on the host system. These sources and sinks are largely similar to the ones described in [21]; we omit a detailed discussion for brevity.

3.3. Propagating Labels

Sabre modifies the interpreter to additionally propagate security labels. JavaScript instructions can roughly be categorized into assignments, function calls and control structures, such as conditionals and loops.

Explicit flows. Sabre handles assignments in the standard way by propagating the label of the RHS of an assignment to its LHS. If the RHS is a complex arithmetic/logic operation, the result is considered sensitive if any of the arguments is sensitive. Assignments to complex objects deserve special care because JavaScript supports dynamic creation of new object properties. For example, the assignment `obj.prop = 0` adds a new integer property `prop` to `obj` if it does not already exist. Recall that Sabre associates a separate label with `obj` and `obj.prop` (in contrast to [42]). In this case, the property `prop` inherits the label of `obj` when it is initially created, but the label may change because of further assignments to prop. An aggregate operation on the entire object (*e.g.,* a length operation on an array) will use the label of the object. In this case, the label of the object is calculated (lazily, when the object is used) to be the aggregate of the labels of its child properties, *i.e.,* an object is considered sensitive if any of its constituent properties stores sensitive information. Sabre handles arrays in a similar fashion by associating each array element associated with its own security label. However, the label of the entire array is the aggregate of its members; doing so is important to prevent unintentional information leaks [42].

Sabre handles function calls in a manner akin to prior work [42]. The execution of a function may happen within a *labeled scope* (described below), in which case the labels of variables modified in the function are combined with the label of the current scope. The scope of a function call such as obj.func() automatically inherits the label of the parent object obj. eval statements are handled similar to function calls; all variables modified by code within an eval inherit the label of the scope in which the eval statement executes.

Cross-domain function calls require special care. For example, consider the following call, which initializes a nsIScriptableInputStream object (sis) using a nsIInputStream object (is): sis.init(is). In this statement, sis is not a JavaScript object. The function call to init is therefore resolved via a cross-domain call. To handle cross-domain calls, we supplied Sabre with a set of *cross-domain function models* that specify how labels must be propagated across such calls. For example, in this case, the model specifies that the label of is must propagate to sis. We currently use 127 function models that specify how labels must be propagated for cross-domain calls.

Implicit flows. While the above statements are examples of explicit data dependencies, conditions (and closely related statements, such as loops and exceptions) induce implicit information flows. In particular, there is a control dependency between a conditional expression and the statements executed within the conditional. Thus, for instance, all statements in both the T and F blocks in the following statement must be considered sensitive, because document.cookie.length is a considered sensitive:

```
if (document.cookie.length > 0) then {T} else {F}
```

Sabre handles implicit flows using labeled scopes. Each conditional induces a scope for both its true and false branches. The scope of each branch inherits the label of its conditional; scopes also nest in the natural way. All objects modified within each branch inherit the label of the scope in which they are executed.

```
x = false; y = false;
if (document.cookie.length > 0)
then {x = true} else {y = true}
if (x == false) {A}; if (y == false) {B}
```

Figure 3. An implicit flow that cannot be detected using labeled scopes.

While scopes handle a limited class of implicit information flows, it is well-known that they cannot prevent all implicit flows. For instance, consider the example shown in Figure 3 (adapted from [19, 42]). In this figure, one of block A or B executes, depending on the result of the first conditional. Consequently, there is an implicit information flow from document.cookie.length to *both* x and y. However, a dynamic approach that uses scopes will only mark one of x or y as sensitive, thereby missing the implicit flow.

Precisely detecting such implicit flows requires static analysis. However, we are not aware of static analysis techniques for JavaScript that can detect all such instances of implicit flow. Although prior work [42] has developed heuristics to detect simple instances of implicit flows, such as the one in Figure 3, these heuristics fail to detect implicit flows in dynamically generated code, *e.g.,* code executed as the result of an eval. Large, real-world JSEs contain several such dynamic code generation constructs. For example, we found several instances of the eval construct in about 50% of the JSEs that we used in our evaluation (Section 4). Our current prototype of Sabre therefore cannot precisely detect all instances of implicit flows. In future work, we plan to investigate a hybrid approach that alternates static and dynamic analysis to soundly detect all instances of implicit flows.

Instruction provenance. In addition to propagating sensitivity values, Sabre uses the provenance of each JavaScript instruction to determine whether a JavaScript object is modified by a JSE. If so, it sets a Boolean flag (Section 3.1) and records the name of the JSE in the security label of the object for diagnostics. Because the JavaScript interpreter can precisely determine the source file containing the bytecode currently being executed, this approach robustly determines the provenance of an instruction, even if it appears in a XUL overlay that is integrated into the browser core.

3.4. Declassifying and Endorsing Flows

As discussed in Section 3.2, a benign JSE can contain information flows that may potentially be classified as violations of confidentiality or integrity. For example, consider the PwdHash [37] JSE, which customizes passwords to prevent phishing attacks. This JSE reads and modifies a sensitive resource (*i.e.,* a password) from a web form, which is then transmitted over the network when the user submits the web form. Sabre raises an alert because an untrusted JSE can use a similar technique to transmit passwords to a remote attacker. However, PwdHash customizes an input password passwd to a domain by converting it into SHA1(passwd||domain), which is then written back to a DOM element whose origin is domain. In doing so, PwdHash effectively *declassifies* the sensitive password. Consequently, this information flow can be whitelisted by Sabre.

To support declassification of sensitive information, Sabre extends the JavaScript interpreter with the ability to declassify flows. A security analyst supplies a *declassification policy*, which specifies how the browser must declassify a sensitive object. Flows that violate integrity can similarly be handled with an *endorsement policy*. Sabre supports two kinds of declassification (and endorsement) policies: *sink-specific* and *JSE-specific*. A sink-specific policy permits fine-grained declassification of objects by allowing an analyst to specify the location of a bytecode instruction and the object externalized by that instruction. In turn, the browser reduces the sensitivity of the object when that instruction is executed. For example, the security analyst would specify the file, function and line number at which to execute the declassification bytecode on the object being externalised. In case of PwdHash, the policy would be the tuple <`stanford-pwdhash.js, finish, 330, field.value`>. In contrast, a JSE-specific policy permits declassification of *all* flows from a JSE and can be used when a JSE is trusted.

Declassification (and endorsement) policies must be supplied with care because declassification causes Sabre to allow potentially unsafe flows. In the experiments reported in Section 4, we manually wrote declassification policies by examining execution traces emitted by Sabre and determining whether the offending flow is part of the advertised behavior of the JSE. If the flow was advertised by the JSE, we wrote a sink-specific policy to allow that flow.

4. Evaluation

We evaluated Sabre using a suite of 24 JSEs, comprising over 120K lines of JavaScript code. Our goals were to test both the effectiveness of Sabre at analyzing information flows and to evaluate its runtime overhead.

4.1. Effectiveness

Our test suite included both JSEs with known instances of malicious flows as well as those with unknown flows. In the latter case, we used Sabre to understand the flows and determine whether they were potentially malicious.

• **JSEs with known malicious flows.** We evaluated Sabre with four JSEs that had known instances of malicious flows. These included two JSEs that contained exploitable vulnerabilities (Greasemonkey v0.3.3 and Firebug v1.01) and two publicly-available malicious JSEs (FFSniFF [13] and BrowserSpy [41]).

To test vulnerable JSEs, we adapted information available in public fora [12, 35, 39, 45] to write web pages containing malicious scripts. The exploit against Greasemonkey attempted to transmit the contents of a file on the host to an attacker, thereby violating confidentiality, while exploits against Firebug attempted to start a process on the host and modify the contents of a file on disk, thereby violating integrity. In each case, Sabre precisely identified the information flow violation. We also confirmed that Sabre did not raise an alert when we used a JSE-enhanced browser to visit benign web pages.

To test malicious JSEs, we considered FFSniFF and BrowserSpy, both of which exhibit the same behavior—they steal passwords and other sensitive entries from web forms and hide their presence from the user by removing themselves from the browser's extension manager. Nevertheless, because Sabre records the provenance of each JavaScript bytecode instruction executed, it raised an alert when FFSniFF and BrowserSpy attempted to transmit passwords to a remote attacker via the network.

In addition to the above JSEs, we also wrote a number of malicious JSEs, both to demonstrate the ease with which malicious JSEs can be written and to evaluate Sabre's ability to detect them. Each of our JSEs comprised under 100 lines of JavaScript code, and were written by an undergraduate student with only a rudimentary knowledge of JavaScript. For example, *ReadCookie* is a JSE that reads browser cookies and stores them in memory. When the user visits a particular web page (in our prototype, any web page containing Google's search utility), the JSE creates a hidden form element, stores the cookies on this form, and modifies the action attribute to redirect the search query to a malicious server address. The server receives both the search query as well as the stolen cookies via the hidden form element. Sabre detects this malicious flow when the user submits the search request because the hidden form field that stores cookies (and is therefore labeled sensitive) is transmitted over the network.

• **JSEs with unknown information flows.** In addition to testing Sabre against known instances of malicious flows, we tested Sabre against 20 popular Firefox JSEs. The goal of this experiment was to understand the nature of information flows in these JSEs and identify suspicious flows.

Our experimental methodology was to enhance the browser with the JSE being tested and examine any violations reported by Sabre. We would then determine whether the violation was because of advertised functionality of the JSE, in which case we whitelisted the flow using a sink-specific declassification or endorsement policy, or whether the flow was indeed malicious. Although we ended up whitelisting suspicious flows for all 20 JSEs, our results described below show that information flows in several of these JSEs closely resemble those exhibited by malicious extensions, thereby motivating the need for a fine-grained approach to certify information flows in JSEs.

In our experiments, which are summarized in Figure 4, we found that the behavior of JSEs in our test suite fell into five categories. As Figure 4 illustrates, several JSEs contained a combination of the following behaviors.

(**1**) **Interaction with HTML forms.** An HTML form is a collection of form elements that allows users to submit information to a particular domain. Example of form elements include login names, passwords and search queries. While malicious JSEs (*e.g.,* FFsniFF) can steal data by reading form elements, we also found that PwdHash [37] reads information from form elements.

PwdHash recognizes passwords prefixed with a special symbol ("`@@`") and customizes them to individual domains to prevent phishing attacks. In particular, it reads the password from the HTML form, transforms it as described in Section 3.4, and writes the password back to the HTML form. This behavior can potentially be misused by an untrusted JSE, *e.g.,* a malicious JSE could read and maliciously modify form elements when the user visits a banking website, thereby compromising integrity of banking transactions. Consequently, Sabre marks the HTML form element containing the password as sensitive, and raises an alert when the form is submitted. However, because the information flow in PwdHash is benign, we declassify the customized password before it is written back to the form, thereby preventing Sabre from raising an alert.

(**2**) **Sending/receiving data over an HTTP channel.** JSEs extensively use HTTP messages to send and receive data, either via `XMLHttpRequest` or via HTTP channels. For example, Web-of-Trust is a JSE that performs an `XMLHttpRequest`

	JSE	Advertised Functionality of JSE	1	2	3	4	5
1.	Adblock Plus	Prevent page elements, such as ads, from being downloaded		✓	✓		
2.	All-in-One-Sidebar	Sidebar control to switch between sidebar panels and view dialog windows			✓		
3.	CoolPreviews	Preview links and images without leaving current page or tab.		✓	✓		
4.	Download Statusbar	Manage downloads from a tidy statusbar			✓		
5.	Fast Video Download	Easy download of video files from popular sites				✓	
6.	Forecastfox	Gets weather forecasts from AccuWeather.com		✓	✓	✓	
7.	Foxmarks Synchronizer	Keeps bookmarks and passwords backed up and synchronized		✓	✓		
8.	Ghostery	Alerts user's about web bugs, ad networks and widgets on webpages			✓		
9.	GooglePreview	Inserts thumbnails and ranks of web sites into Google search results		✓	✓		
10.	Greasemonkey (0.8.1)	Allows users customize webpages with user scripts		✓	✓		
11.	NoScript	Restricts executable content to trusted domains		✓	✓		
12.	PDF Download	Tool for handling, viewing and creating Web-based PDF files		✓	✓	✓	
13.	Pwdhash	Customizes user passwords to domains to prevent phishing	✓				
14.	SpeedDial	Easy access to frequently visited websites			✓	✓	
15.	StumbleUpon	Discovers web sites based on user's interests		✓	✓	✓	
16.	Stylish	Easy management of user styles to enhance browsing experience		✓	✓	✓	✓
17.	Tab Mix Plus	Enhances Firefox's tab browsing capabilities			✓	✓	
18.	User Agent Switcher	Switches the user agent of the browser			✓		
19.	Video DownloadHelper	Tool for web content extraction		✓	✓		
20.	Web-of-Trust	Warns users before they interact with a harmful site		✓	✓	✓	

Behavior key: **(1)** HTML forms; **(2)** HTTP channels; **(3)** File system; **(4)** Loading URLS; **(5)** JavaScript events.

Figure 4. Behavior of popular Firefox JSEs categorized as in Section 4.1.

for each URL that a user visits, in order to fetch security ratings for that URL from its server.

While this behavior can potentially be misused by malicious JSEs to compromise user privacy by exposing the user's surfing patterns, we allowed the `XMLHttpRequest` in Web-of-Trust by declassifying the request.

(**3**) **Interaction with the file system.** With the exception of two JSEs, the rest of the JSEs in our test suite interacted with the file system. For example, Video DownloadHelper and Greasemonkey download content from the network on to the file system (media files and user scripts, respectively), while ForecastFox reads user preferences, such as zip codes, from the file system and sends an `XMLHttpRequest` to receive weather updates from accuweather.com.

Both these behaviors can potentially be misused by malicious JSEs, to download malicious files on the host and steal confidential data, such as user preferences. However, we allowed these flows by endorsing the file system write operation in Video DownloadHelper and Greasemonkey and by declassifying the `XMLHttpRequest` in ForecastFox.

(**4**) **Loading a URL.** Several JSEs, such as SpeedDial and PDF Download, monitor user activity (*e.g.,* keystrokes, hyperlinks clicked by the user) and load a URL based upon this activity. For example, PDF Download, which converts PDF documents to HTML files, captures user clicks on hyperlinks and sends an `XMLHttpRequest` to its home server to get a URL to a mirror site. It then constructs a new URL by appending the mirror's URL with the hyperlink visited by the user, and loads the newly-construced URL in a new tab.

Similar behavior can potentially be misused by a JSE, *e.g.,* to initiate a drive-by-download attack by loading an untrusted URL. However, for PDF Download, we endorsed the JavaScript statements that load URLs in the JSEs that we tested, thereby preventing Sabre from raising an alert.

(**5**) **JavaScript events.** Unprivileged JavaScript code on a web page can communicate with privileged JavaScript code (*e.g.,* code in JSEs) via events. In particular, JSEs can listen for specific events from scripts on web pages.

We found one instance of such communication in the Stylish JSE, which allows easy management of CSS styles for web sites. A user can request a new style for a web page, in response to which the JSE opens a new tab with links to various CSS styles. When the user chooses a style, JavaScript code on web page retrieves the corresponding CSS style and throws an event indicating that the download is complete. Stylish captures this event, extracts the CSS code, and opens a dialog box for the user to save the file.

Sabre raises an alert when the user saves the file. This is because Sabre assigns a low integrity label to JavaScript code on a web page; in turn the event thrown by the code also receives this label. Sabre reports an integrity violation when the JavaScript code in Stylish handles the low-integrity event and attempts to save data on to the file system (a high-integrity sink). Nevertheless, we suppressed the alert by endorsing this flow.

Sabre provides detailed traces of JavaScript execution for offline analysis. We used these traces in our analysis of JSEs to determine whether an information flow was benign, and if so, determine the bytecode instruction and the JavaScript object at which to execute the declassification/endorsement policy. Although this analysis is manual, in our experience, it only took on the order of a few minutes to determine where to place declassifiers.

As the examples above indicate, several benign JSEs exhibit information flows that can possibly be misused and must therefore be analyzed and whitelisted. It is important to note that each of these information flows exhibited *real* behaviors in JSEs. Because such behaviors may possibly be misused by malicious JSEs, determining whether to whitelist a flow is *necessarily* a manual procedure, *e.g.,* of studying the high-level specification of the JSE to determine if the behavior conforms to the specification.

To evaluate the precision of Sabre, we also studied whether it reported any *other* instances of flows from sensitive sources to low-sensitivity sinks, *i.e.,* excluding the flows that were whitelisted above. We used a Sabre-enhanced browser for normal web browsing activity over a period of several weeks. During this period Sabre reported *no* violations. We found that Sabre's policy of reporting an information flow violation only when an object is modified by a JSE was crucial to the precision of Sabre.

The analysis above shows that benign JSEs often contain information flows that can potentially be misused by malicious JSEs. These results therefore motivate a security architecture for JSEs in which JSE vendors explicitly state information flows in a JSE by supplying a declassification/endorsement policy for confidentiality/integrity violating flows. This policy must be approved by the user (or a trusted third party, such as `addons.mozilla.org`, that publishes JSEs) when the JSE is initially installed and is then enforced by the browser.

It is important to note that this architecture is agnostic to the *code* of a JSE and only requires the user to approve *information flows*. In particular, the declassification policy is decoupled from the code of the JSE is enforced by the browser. As a result, only flows whitelisted by the user will be permitted by the browser, thereby significantly constraining confidentiality and integrity violations via JSEs. This architecture also has the key advantage of being robust even in the face of attacks enabled by vulnerabilities in the JSE.

4.2. Performance

We evaluated the performance of Sabre by integrating it with SpiderMonkey in Firefox 2.0.0.9. Our test platform was a 2.33Ghz Intel Core2 Duo machine running Ubuntu 7.10 with 3GB RAM. We used the SunSpider and V8 JavaScript benchmark suites to evaluate the performance of Sabre. Our measurements were averaged over ten runs.

With the V8 suite, a Sabre-enabled browser reported a mean score of 29.16 versus 97.91 for an unmodified browser, an overhead of 2.36×, while with SunSpider, a Sabre-enabled browser had an overhead of 6.1×. We found that the higher overhead in SunSpider was because of three benchmarks (3d-morph, access-nsieve and bitops-nsieve-bits). Discounting these three benchmarks, Sabre's overhead with SunSpider was 1.6×. Despite these overheads, the performance of the browser was not noticeably slower during normal web browsing, even with JavaScript-heavy web pages, such as Google maps and street views.

The main reason for the high runtime overhead reported above is that Sabre monitors the provenance of *each* JavaScript bytecode instruction to determine whether the instruction is from a JSE (to set the Boolean flag in the security label, as described in Section 3.3). Monitoring each instruction is important, primarily because code included in overlays (distributed with JSEs) is included in the browser core and may be executed at any time. If such overlays can separately be verified to be benign, these checks can be disabled. In particular, when we disabled this check, we observed a manageable overhead of 77% and 42% with the V8 and SunSpider suites, respectively. Ongoing efforts by Eich *et al.* [23, 24] to track information flow in JavaScript also incur comparable (20%-70%) overheads.

5. Related Work

Browser extension security. Prior work [22, 30, 31] has developed techniques to identify spyware behavior in untrusted browser extensions, particularly in plugins and BHOs, which are distributed as binary executables. These approaches rely on whole-system information flow tracking [22] and on monitoring plugin/browser interactions [30]. Like prior work [30, 31], Sabre also monitors JSE/browser interactions but supplements such monitoring with information on sensitivity/integrity of JavaScript objects. As illustrated in Section 4, this information is important for JSEs, because several benign JSEs interact with the browser in a manner akin to malicious JSEs. Spyshield [31] additionally offers containment by enforcing policies on data accesses by untrusted plugins; such techniques can possibly complement Sabre to contain malicious JSEs. Like prior work [22], Sabre also performs information-flow tracking, but does so at the JavaScript level within the browser. As discussed in Section 1, doing so eases action attribution and integration with the browser. Recent work has explored techniques to sandbox browser extensions [26], but such work is currently applicable only to extensions such as plugins and BHOs, which are distributed as binary executables.

Ter-Louw *et al.* [41] were the first to address the security of JSEs. However, as discussed in Section 1, their work was based on monitoring XPCOM calls; being coarse-grained, their approach can have both false positives and negatives.

JavaScript information flow. Netscape Navigator 3.0 first proposed the use of data tainting to detect confidentiality-violating JavaScript code [16]. This idea has been applied by Vogt *et al.*[42] to detect cross-site scripting attacks. More recently, Austin *et al.* [17] have proposed dynamic taint tracking techniques for JavaScript with promising results, but over considerably smaller benchmarks. In addition, there is rich literature on information-flow tracking for both web applications and for executable code; we do not survey that work in this paper.

Although we leverage the JavaScript label propagation rules developed in prior work, analyzing information flow in JSEs poses additional challenges, as illustrated throughout this paper. In particular, Sabre precisely tracks cross-domain information flows and provides support for fine-grained declassification or endorsement of flows. To our knowledge, prior work on JavaScript information flow tracking has not needed or incorporated such support.

JavaScript sandboxing. Early work by Hallaraker and Vigna [29] proposed XPCOM-level monitoring to sandbox JavaScript code. Recent work on sandboxing JavaScript has focused on the problem of confining untrusted third-party code that may be included in web pages as widgets and advertisements. Notable efforts include Adsafe [1], Caja [33] and FBJS [2], which perform rewriting to restrict the JavaScript constructs allowed in code included in web pages. Although such techniques may possibly be used to secure JSEs as well, restricting JavaScript constructs in JSEs may restrict their functionality.

BrowserShield [36], CoreScript [47], Phung *et al.* [34] and Erlingsson *et al.* [25] proposed the use of JavaScript instrumentation to ensure compliance with site-specific security policies. Such techniques can be used to enforce access control policies on a JSE's accesses to sensitive browser data. While such an approach can possibly constrain malicious JSEs, it is unclear whether it will also protect exploits against vulnerable JSEs (*e.g.,* those in Section 2).

Chugh *et al.* [20] and Yip *et al.* [46] have proposed sandboxing of third-party JavaScript executing on web pages. In contrast, Sabre sandboxes JSEs, which execute with more privileges and interact with more browser subsystems.

In addition to the above work on JavaScript sandboxing, recent research has investigated static analysis techniques for JavaScript code [32], particularly to statically ensure compliance with site-specific policies and to ensure the integrity of client-side JavaScript code of a web application [28]. We plan to explore whether Sabre can leverage similar static analysis techniques to reduce the runtime overhead of information flow tracking for JSEs. However, we expect that performing such analysis will be challenging for obfuscated JSEs as well as those that contain a large number of dynamic code generation constructs, such as `eval`.

6. Conclusion

This paper presented Sabre, an in-browser information-flow tracker that can detect confidentiality and integrity violations in JSEs, enabled either because of malicious functionality in JSEs or because of exploitable vulnerabilities in the code of a JSE.

In future work, we plan to improve the performance of Sabre by exploring static analysis of JavaScript code. For example, static analysis can be used to create *summaries* of fragments of JavaScript code that do not contain complex constructs (*e.g.,* `eval`). These summaries record how the labels of objects accessed by the fragments are modified. Sabre can use these summaries to update labels when the fragment is executed, thereby avoiding the need to propagate security labels for *each* bytecode instruction.

Acknowledgements. We thank Jan Jajalla for his help with experiments, members of DiscoLab and the anonymous reviewers for their comments. This work was supported by NSF awards 0831268, 0915394 and 0931992.

References

[1] AdSafe: Making JavaScript safe for advertising. http://www.adsafe.org.

[2] FBJS: Facebook developers wiki. http://wiki.developers.facebook.com/index.php/FBJS.

[3] Firebug: Web development evolved. http://getfirebug.com.

[4] Firefox Add-ons. http://addons.mozilla.org.

[5] Greasespot: The weblog about Greasemonkey. http://www.greasespot.net.

[6] Internet Explorer 8. http://www.microsoft.com/windows/internet-explorer.

[7] Mozilla.org XPCOM. http://www.mozilla.org/projects/xpcom.

[8] NoScript—JavaScript blocker for a safer Firefox experience. http://noscript.net.

[9] Signed scripts in Mozilla: JavaScript privileges. http://www.mozilla.org/projects/security/components/signed-scripts.html.

[10] XML user interface language (XUL) project. http://www.mozilla.org/projects/xul.

[11] FormSpy: McAfee avert labs, July 2006. http://vil.nai.com/vil/content/v_140256.htm.

[12] Mozilla Firefox Firebug extension—Cross-zone scripting vulnerability, April 2007. http://www.xssed.org/advisory/33.

[13] FFsniFF: FireFox sniFFer, June 2008. http://azurit.elbiahosting.sk/ffsniff.

[14] Firefox add-ons infecting users with trojans, May 2008. http://www.webmasterworld.com/firefox_browser/3644576.htm.

[15] Trojan.PWS.ChromeInject.B, Nov 2008. http://www.bitdefender.com/VIRUS-1000451-en--Trojan.PWS.ChromeInject.B.html.

[16] Netscape Navigator 3.0. Using data tainting for security. http://www.aisystech.com/resources/advtopic.htm.

[17] T. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *ACM PLAS*, June 2009.

[18] P. Beaucamps and D. Reynaud. Malicious Firefox extensions. In *Symp. Sur La Securite Des Technologies De L'Information Et Des Communications*, June 2008.

[19] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *DIMVA*, July 2008.

[20] R. Chugh, J. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *PLDI*, June 2009.

[21] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. Technical Report DCS-TR-648, Rutgers University, April 2009.

[22] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *USENIX Annual Technical*, June 2007.

[23] B. Eich. Better security for JavaScript, March 2009. Dagstuhl Seminar 09141: Web Application Security.

[24] B. Eich. JavaScript security: Let's fix it, May 2009. Web 2.0 Security and Privacy Workshop.

[25] U. Erlingsson, Y. Xie, and B. Livshits. End-to-end web application security. In *HotOS*, May 2007.

[26] B. Yee *et al.*. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE S&P*, May 2009.

[27] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE S&P*, May 2008.

[28] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *WWW*, April 2009.

[29] O. Hallaraker and G. Vigna. Detecting malicious JavaScript code in Mozilla. In *10th IEEE Conf. on Engineering Complex Computer Systems*, June 2005.

[30] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based spyware detection. In *USENIX Security*, August 2006.

[31] Z. Li, X. Wang, and J. Y. Choi. SpyShield: Preserving privacy from spy add-ons. In *RAID*, September 2007.

[32] B. Livshits and S. Guarnieri. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. Technical Report MSR-TR-2009-16, Microsoft Research, 2009.

[33] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript, June 2008.

[34] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In *ASIACCS*, March 2009.

[35] M. Pilgrim. Greasemonkey for secure data over insecure networks/sites, July 2005. http://mozdev.org/pipermail/greasemonkey/2005-July/003994.html.

[36] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic HTML. In *ACM/USENIX OSDI*, November 2006.

[37] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *USENIX Security*, August 2005.

[38] J. Ruderman. The same-origin policy, August 2001. http://www.mozilla.org/projects/security/components/same-origin.html.

[39] Secunia Advisory SA24743/CVE-2007-1878/CVE-2007-1947. Mozilla Firefox Firebug extension two cross-context scripting vulnerabilities.

[40] Secunia Advisory SA30284. FireFTP extension for Firefox directory traversal vulnerability.

[41] M. Ter-Louw, J. S. Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal of Computer Virology*, 4(3), August 2008.

[42] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, February 2007.

[43] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM CCS*, November 2002.

[44] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. Technical Report MSR-TR-2009-16, Microsoft Research, February 2009.

[45] S. Willison. Understanding the Greasemonkey vulnerability, July 2005. http://simonwillison.net/2005/Jul/20/vulnerability.

[46] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with bflow. In *EuroSys*, April 2009.

[47] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *ACM POPL*, January 2007.