International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395 -0056
Volume: 03 Issue: 12 | Dec -2016 www.irjet.net p-ISSN: 2395-0072

Analyzing Information Flow in Java based Browser Extensions

Dr.T.Pandikumar[1], Teklish Girma[2]
[1]*Ph.D. Department of Computer & IT, College of Engineering, Defence University, Ethiopia*
[2]*M.Tech. Department of Computer & IT, College of Engineering, Defence University, Ethiopia*

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.*

Keywords: Sabre, Java Script, JSE, Browser, Window, HTML, Extension

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 plug-in (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 Grease monkey [5], which allows user-defined scripts to customize how web pages are rendered, Firebug [3], a JavaScript development environment, and No Script [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 plug-in 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, 14, 15, 18, 41]. Indeed, we wrote several such JSEs during the course of this research work.

(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/Grease monkey 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 plug-in 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.

To summarize, the main contributions of this paper are:

• Sabre, an information flow tracker for JSEs. 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 white listed. We chose Firefox as our implementation and evaluation Platform because of the popularity and wide availability of JSEs for Firefox. 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). 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

```
<script type="text/javascript">
 window._GM_xmlhttpRequest = null;
function trapGM(...) {
window._GM_xmlhttpRequest=
window.GM_xmlhttpRequest;
 . ...
 }
 function checkGM() {
if (window._GM_xmlhttpRequest) {
window._GM_xmlhttpRequest(
{method: 'GET', url: 'file:///c:/boot.ini',
 onload: function(Response) {
 document.formname.textfield.value=
Response.responseText;
  }});
  }
  }
 if (typeof window.addEventListener != 'undefined')
{
window.watch('GM_apis', trapGM);
window.addEventListener('load', checkGM, true);
 }
</script>
```

Figure 1. Example of malicious JavaScrip code that exploits the Greasemon key vulnerability to read the contents of boot.ini from disk

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.

2.1 Grease monkey/Firefox Vulnerability

Grease monkey 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 Grease monkey that would customize the background of web pages that he visits. Grease monkey exports a set of APIs (prefixed with "GM") those 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 xml http Request API allows a user-defined script to execute an XMLHttpRequest to an arbitrary web domain, and is not constrained by the same-origin policy. Although the script 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.

```
function do_sniff() {
var hesla =
window.content.document.getElementsByTagName("i
nput");
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.

JavaScript code from Grease monkey to access the local file system consequently and response. Response Text, which this function returns, is also marked sensitive. 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. 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.

2.2 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 vulnerability in the Firefox extension manager (CVE-2006-6585) to delete its entry from the add-ons list presented by Firefox. 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.

2.3 Tracking Information Flow with Sabre 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].

(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 white list the flow.

(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

often involve multiple browser subsystems. We implemented Sabre by modifying Spider Monkey, the JavaScript interpreter in Firefox, to track information flow. We modified Spider Monkey's representation of JavaScript objects to include security labels. We also enhanced the interpretation of JavaScript byte code 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. 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 byte code 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.

3. SECURITY

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. The DOM node that stores the response from the GM xml http Request call is marked sensitive. Further, the data contained in the node is modified by executing code contained in Grease monkey, via the return value from GM xml http Request. 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.

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. 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 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. 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.

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 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. 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. 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. 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. In addition to propagating sensitivity values, Sabre uses the provenance of each Java Scrip instruction to determine whether a JavaScript object is modified by a JSE.

3.4. Declassifying and Endorsing Flows

JSE can contain information flows that may potentially be classified as violations of confidentiality or integrity. For example, consider the Pwd Hash [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, Pwd Hash 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, Pwd-Hash effectively declassifies the sensitive password. Consequently, this information flow can be white listed 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 byte code 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 byte code on the object being externalized. In case of Pwd Hash, 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.

4. EVALUATION & PERFORMANCE

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.

4.1.1 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 (Grease monkey v0.3.3 and Firebug v1.01) and two publicly-available malicious JSEs (FFSniFF [13] and Browser Spy). To test vulnerable JSEs, we adapted information available in public fora [12] to write web pages containing malicious scripts. The exploit against Grease monkey 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 Browser Spy, 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 byte code instruction executed, it raised an alert when FFSniFF and Browser Spy 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 fewer than 100 lines of JavaScript code, and was written by an undergraduate student with only a rudimentary knowledge of JavaScript.

4.2. Performance

We evaluated the performance of Sabre by integrating it with Spider Monkey in Firefox 2.0.0.9. Our test platform was a 2.33 GHz Intel Core2 Duo machine running Ubuntu 7.10 with 3GB RAM. We used the Sun Spider 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 Sun Spider; a Sabre-enabled browser had an overhead of 6.1×. We found that the higher overhead in Sun Spider was because of three benchmarks (3d-morph, access-nsieve and bitops-nsievebits). Discounting these three benchmarks, Sabre's overhead with Sun Spider 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 byte code instruction to determine whether the instruction is from a JSE. 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 Sun Spider suites, respectively. Ongoing efforts by Eichet al. [23, 24] to track information flow in JavaScript also incur comparable (20%-70%) overheads.

4. CONCLUSION

This research review 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 byte code instruction.

REFERENCES

[1] T. Austin and C. Flanagan. Efficient purely-dynamic information flowanalysis. In ACM PLAS, June 2009.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[16] 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.

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

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

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

[20] 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.

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

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

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

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

[25] 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.

[26] 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.

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

[28] 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.

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

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

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