Towards Precise and Efficient Information Flow Control in Web Browsers *

Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, Michael Franz
{ckerschb, eric.hennigan, perl, s.brunthaler, franz}@uci.edu

University of California, Irvine

Abstract. JavaScript (JS) has become the dominant programming language of the Internet and powers virtually every web page. If an adversary manages to inject malicious JS into a web page, confidential user data such as credit card information and keystrokes may be exfiltrated without the users knowledge.
We present a comprehensive approach to information flow security that allows precise labeling of scripting-exposed browser subsystems: the JS engine, the Document Object Model, and user generated events. Our experiments show that our framework is precise and efficient, and detects information exfiltration attempts by monitoring network requests.

1 Motivation

The JS programming language forms a key component in today's web architecture, especially in Web 2.0 applications which regularly use JS to handle sensitive information, such as corporate customer accounts. The current web page architecture allows source and library code from different origins to share the same execution context in a user's browser. Attackers take advantage of this execution model to gain access to a user's private data using Cross Site Scripting (XSS).

XSS is a code injection attack that allows adversaries to execute code without the user's knowledge and consent. Without any observable difference in runtime behavior, a malevolent script can exfiltrate keystrokes, or traverse the Document Object Model (DOM) to exfiltrate all visible data on a web page. Vulnerability studies consistently rank XSS highest in the list of the most mounted attacks on web applications [1, 2]. A recent study [3] confirms the ubiquity of sensitive user data exfiltration currently practiced on the Internet.

*This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contract No. D11PC20024, by the National Science Foundation (NSF) under grant No. CCF-1117162, and by a gift from Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA) or its Contracting Agent, the U.S. Department of the Interior, National Business Center, Acquisition Services Directorate, Sierra Vista Branch, the National Science Foundation, or any other agency of the U.S. Government.

As a first line of defense, browsers implement the same origin policy (SOP) that limits a script's access to information. This policy allows scripts from the same origin to access each other's data and prevents access for scripts of different origins. Regrettably, attackers can bypass the SOP, e.g., by exploiting a XSS vulnerability of a web page, or by providing code, such as a library that is integrated in the same JS execution context as the original page.

Tracking the flow of information in the user's browser seeks to address the limitations of the SOP. Unfortunately, previous work [4–7] either limits tracking to a single bit of information, focuses solely on the JS engine or the DOM, or introduces significant runtime overhead. These limitations make wide browser adoption unlikely. Tracking only one bit of information leads to a high false positive rate in Web 2.0 applications, where pages commonly use Content Distribution Networks (CDNs).

We take inspiration from all of these approaches and present a comprehensive tracking framework (Section 3) that supports precise labeling for the dynamic tracking of information flows within a browser, including: (1) the JS engine, (2) the DOM, and (3) user generated events. We evaluate our system (Section 4) showing that it satisfies the following properties: *a) Secure:* Our system can stop information exfiltration attempts; in particular we show this by injecting malicious code that performs a keylogging attack, and attempts to exfiltrate HTML-form data. *b) Precise:* Our framework makes information flow tracking feasible for Web 2.0 applications by supporting multi-domain label encoding. We confirm this feasibility by visiting the Alexa Top 500 pages using our implemented web crawler. *c) Efficient:* Our approach incurs an average overhead of 82.82% in the JS engine (on SunSpider benchmarks) and 5.43% in the DOM (on Dromaeo benchmarks). Note, that the fastest dynamic information flow tracking frameworks [5, 7] introduce overhead on the order of 200-300%.

2 Threat Model

Throughout this paper, we assume that attackers have two important abilities: (1) attackers can operate their own hosts, and (2) can inject code in other web pages. Code injection into other pages relies either on exploiting a XSS vulnerability of a page, or the ability of attackers to provide content for mashups, advertisements, libraries, etc., which other sites include. The attacker's abilities, however, are limited to JS injection and attackers can neither intercept nor control network traffic. Our framework protects against several threats, including, but not limited to:

Information Exfiltration Attacks: By sending a `GET` request to a server under the attacker's control, the attacker can exfiltrate information in the URL of an image request: `elem.src = "evil.com/pic.png?"+creditcard_number;`. The attacker uses the request for the image as a channel to exfiltrate a user's credit card number as a payload in the `GET` request, when loading the image from the server.

Keylogging Attacks: An attacker might also craft code that logs keystrokes by registering an event handler: `document.onkeypress = listenerFunction;`. Our framework can track the flow of information for generated events, and can therefore also detect and prevent keylogging attacks.

3 Design and Implementation

We implement a framework, WIF (WebKit Information Flow), which extends the WebKit browser (version 1.4.2) with support for dynamic tracking of information flows. Several industrial strength desktop and mobile browsers use the WebKit code, e.g., Google's Chrome, or Apple's Safari. To protect the information accessible by an executing script, we use a labeling model that enforces the memory semantics of a non-interference security policy [8].

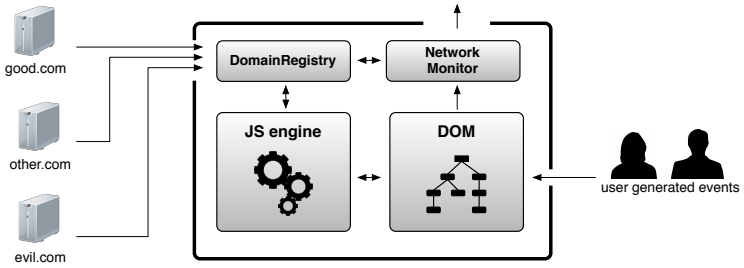

Fig. 1. Architecture of WIF.

Our approach extends the browser's JS engine with the ability to tag values with a security label indicating their originating domain. WIF introduces a DomainRegistry (Figure 1) to manage these labels.

A single web page can incorporate data from several different domains, therefore we associate a unique label with each domain. In the JS engine, data and objects originating from different domains may interact, creating values which derive from more than one domain. To model this behavior, we take inspiration from Myers' decentralized label model [9] and represent security labels as a lattice join over domains.

DomainRegistry: When the browser loads HTML or JS, it registers the code's domain of origin in the DomainRegistry before processing. The DomainRegistry maps every domain to a unique bit in a 64 bit label. During execution, our framework attaches these labels to new JS values and HTML-tokens based on the origin. This design allows us to use efficient bit arithmetic for label join (`0001|0010=0011`) operations that propagate labels.

Information Flow in the JS engine: As a foundation for WIF, we implement information flow tracking within the JS engine using an approach similar to other researchers [4, 3, 5].

We implement security labeling by extending every JS value from 64 to 128 bits, where the upper 64 bits represent the actual JS value and the lower 64 bits indicate the domain of ownership. For example, a simplified example of a binary operation like `c = a + b`, where `a` comes `a.com` (mapped to `0001`) and `b` originates from `b.com` (mapped to `0010`) would cause `c` to hold the value of `a + b` in the upper 64 bits, and the labels of both, `a` and `b` (`0001|0010=0011`) in the lower 64 bits.

This design lets us directly encode 63 different domains in one label. We reserve the highest bit in the label to indicate that the direct encoding of 63 domains overflows. The overflow-bit indicates that the page incorporates code from more than 63 different domains. In such a case, our system switches to a slower label propagation mechanism, where the lower 63 bits become an integer index into an array of labels. When visiting the Alexa Top 500 pages with our web crawler, we discovered that pages, on average, include content from 12 different domains.

Conventional static analysis techniques for information flow, such as those developed for the Java-based Jif [10], are not directly applicable to dynamically typed languages, such as JS. However, we adapt these techniques by introducing a *control-flow stack* that manages labels for different security regions of a running program. At runtime, the JS engine updates the label of the *program counter* at every control flow branch and join within a program. The top of the control-flow stack always contains the current security label of the program counter. Using the control-flow stack, our system is able to track:

- *Explicit Information Flows,* which occur when some value explicitly depends on another variable, e.g., `var pub = secret;`.
- *Implicit Direct Information Flows,* which occur when some value can be inferred from the predicate of a branch in control flow, e.g., `if (secret) { pub = true;}`. An attacker can gain information about the secret variable by inspecting the value of the variable `pub` after execution of the `if` statement. The handling of implicit direct information flows therefore requires joining the label of the variable `pub` with the label of `secret`. The latter assignment to `pub` occurs in a labeled (secure) region, which causes `pub` to be tainted with the label of the current program counter.

We refer the reader to an accompanying technical report [11] for further details about maintaining the control-flow stack.

Information Flow in the DOM: The DOM provides an interface that allows JS in a web page to reference and modify HTML elements as if they were JS objects. For example, JS can dynamically change the `src`-attribute of an image so that the image changes whenever the user's cursor hovers over it. Malicious JS can use the DOM as a communication channel to exfiltrate information present within a web page. WIF prevents such exfiltration attempts by labeling DOM objects based on the origin of their elements and attributes. During HTML parsing, browsers build an internal tree representation of the DOM. Our framework

uses this phase to attach an initial label, indicating the domain of origin, on all element and attribute nodes in the newly constructed DOM-tree.

JS code that calls `document.write` can force the tokenizer to pause and process new markup content from the script, before continuing parsing the regular page markup. WIF applies labels to HTML tokens so that tokens generated by the call inherit the label of the script, while regular markup inherits the label of the page.

JS can make use of different syntactical variants to assign a value to an HTML attribute in the DOM, e.g., *element*`.name =` *value*`;`. Internally, all the different variants dispatch to a function, `setAttribute`. We extend the argument list to include a label, which supports precise labeling, even for custom attributes available in HTML5. Performing labeling solely on attributes in the DOM, however, does not provide a complete solution. For example, a call to the `innerHTML` property of a `div`-element that returns only plain text of the displayed data without a label. To contain dynamically calculated properties, such as `innerHTML` and `value`, WIF modifies these functions to apply the label of the DOM element to the data before returning it to the JS engine.

User Events: In a web browser, the execution context for every script corresponds to the domain of that document. Whenever JS code triggers an event, WIF handles this event similar to a control-flow branch. It creates a new security region for handling the event, and joins the PC-label (top of the control-flow stack) with the label of the execution context. Once the event handler has finished execution, our framework restores the browser's previous state. Using this technique, our framework attaches a label to user generated JS events.

Network Monitor: At every network request, WIF checks whether the label of the URL-string matches the server domain in the network request. To do so, WIF extracts the domain of the `GET` request and looks up the corresponding 64-bit label in the DomainRegistry. Then WIF checks whether the 64 bit label of the URL-string matches the 64 bit label of the domain of that `URL`. Based on the result of an XOR operation on the two labels, our system decides whether the request is allowed.

Policy: We consider inequality of labels ($0011 \neq 0001$) to be a privacy violating information flow. When WIF detects such a violating flow, it records the event and notifies the user.

4 Evaluation

Security Evaluation: To verify that WIF is able to detect information exfiltration attempts, we inject custom exploit code into ten mirrored web pages with known XSS vulnerabilities. To find such web pages, we use XSSed (`xssed.com`), which provides the largest online archive of XSS vulnerable web sites, listing more than 45,000 web pages, including government pages, and pages in the Alexa Top 100 world wide. We inject malicious code that exfiltrates all keys

pressed by a user into a mirrored vulnerable web page of `amazon.com`. This mirrored page pulls and integrates code from eight different origins on the Internet. Our framework successfully detects the attempt to exfiltrate logged keys, HTML-form data, and other exfiltration attempts.

Web Crawler Statistics: To perform a quantitative evaluation of our system, we implement a web crawler that automatically visits the Alexa Top 500 (`alexa.com`) web pages and stays on each web page for 60 seconds. To simulate user interaction, we equip this web crawler with the ability to fill out HTML-forms and submit the first available form. We found information flows across domain boundaries on 433 of the 500 visited web pages. This frequency emphasizes the importance of providing an opportunity to retrace the flow of information in a user's browser. The following statistics show a snapshot of consistently changing web pages, taken on December 24th, 2012.

Distinct Content Providers	3,061
Violating Information Flows	8,764
Flows labeled with one domain	5,947
Flows labeled with more than one domain	2,817

Table 1. Overall findings when browsing the Alexa Top 500 pages.

Distinct Content Providers: As shown in Table 1, the Alexa Top 500 pages include content from a total of 3,061 distinct domains on the Internet. Verification and proof that all these content suppliers are benign and trustworthy is not available. A recent study [12] shows that web sites expand their "circle of trust" by introducing about 45% new JS inclusions each year. This trend encourages our efforts, because hacking just one of those inclusions gives immediate access to sensitive user data.

Statistics about information flow violations: When visiting the Alexa Top 500 pages we detect a total of 8,764 information flow violations (Table 1) which target a total of 1,384 distinct domains on the Internet. Our precise labeling reveals interacting domains that cause an information flow violation. We found that 2,817 out of the detected 8,764 violating information flows have more than one domain encoded in their label. One such information flow violation was found on `t-online.de`, where information was labeled with domains of `t-online.de`, `stats.t-online.de`, `im.banner.t-online.de`, `imagesrv.adition.com`, `ad2.adfarm1.adition.com`. Using such a multi-domain labeling strategy allows our system to clearly identify CDNs, like e.g., `stats.t-online.de`.

When crawling the Alexa Top 500 pages, our network monitor also reported a flow, where information was labeled with more than one domain, to the hardcoded IP-address `124.17.1.253`. We used the service of `whois.net` and discovered that *China Science & Technology Network* owns the IP-address. Put differently, this IP-address might belong to almost anyone in China, benign or malicious. Manual inspection of payloads in such network requests

is almost impossible, because information is often encoded in a highly obfuscated manner.

This result lets us conclude that only web site authors are able to provide information about permitted flows. Defining a policy of permitted flows and tracking the flow of information in a user's browser therefore seems the most promising approach to prevent information exfiltration attacks.

Performance Evaluation: We execute all benchmarks on a dual Quad Core Intel Xeon E5462 2.80 GHz with 9.8 GB RAM running Ubuntu 11.10 (kernel 3.2.0) using gcc-4.6.3, where we use `nice -n -20` to minimize operating system scheduler effects.

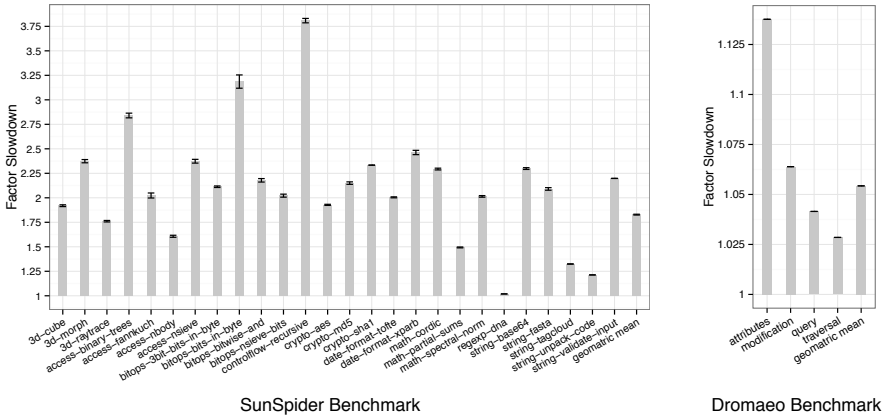

Fig. 2. left: Detailed JS engine performance impact per SunSpider benchmark, **right:** Detailed DOM performance impact per Dromaeo benchmark (both normalized by WebKit's JS interpreter, `JavaScriptCore`).

Figure 2 (left) shows the results for executing the SunSpider benchmarks using WIF. Our system has an average slowdown factor of 1.8×, or 82.82% when normalized to WebKit's original JS interpreter, `JavaScriptCore`. WIF introduces this overhead in the JS engine because it propagates labels for all created and modified JS values during execution of an application. To the best of our knowledge, the fastest information flow tracking systems run two to three times slower with tracking enabled [5, 7], which indicates that our implementation is substantially faster.

The results of the DOM benchmarks in Figure 2 (right) show that WIF introduces an average overhead of 5.43%, on Dromaeo benchmarks. This overhead is due to WIF managing not only the attribute value in the DOM, but also the corresponding label.

Current Limitations, Discussion and Future Work: Our system does not yet handle *implicit indirect information flows*, where information can be inferred by inspecting values in the non-executed path. The efficient handling of such flows still remains an open research question.

Currently, consumer browsers do not support any kind of information flow control to provide security against information exfiltration attacks. We believe that the introduced overhead for tracking the flow of information is the major obstacle for widespread adoption. We have shown that labeling the DOM introduces only around 5% overhead. To the best of our knowledge there is no just-in-time (JIT) compiler that performs information flow tracking for interpreted languages, such as JS. Other information flow tracking systems also integrate their tracking mechanisms in the JS interpreter (cf. [4, 7, 5]). Comparing the performance of our tracking framework against WebKit's JIT compiler reveals that our system introduces a slowdown of 6.3×, or 536.48% (the `JavaScriptCore` interpreter itself introduces an overhead of 3.5×, or 248.14%, compared to it's JIT compiler). We are planning on exploring the performance impact of dynamic information flow tracking using a JIT.

Showing the tradeoff between security and performance, the reader might remember the introduction of *Address Space Layout Randomization*, which after years of research finally found deployment in real world systems because the introduced overhead became negligible compared to the security gain.

5 Related Work

Vogt et al. [4] presents work closely related to ours. This pioneering work shows the practicality of using information flow control to enforce JS security. In contrast, they only use one bit as label information whereas our approach allows multi-domain labeling. Unfortunately they do not provide performance numbers which would make comparison to other work more comprehensive. Just et al. [5] presents an information flow framework improving on the results of Vogt et al [4]. They also use a stack for labeling secure regions of a program, but solely focus on the JS engine. Russo et al. [6] provides a mechanism for tracking information flow within dynamic tree structures. This work, in contrast, solely discusses information flow tracking in the DOM.

De Groef et al. [7] presents a system that uses secure multi-execution to enforce information control security in web browsers. Even though their approach presents a general mechanism for enforcing information flow control, their approach introduces substantial overhead. This is due to the nature of secure-multi-execution, which requires them to execute JS up to 2^n times, for n domains.

Hedin and Sabelfeld [13] present a dynamic type system that ensures information flow control for a core of JS. They do not provide an implementation but address the challenge of tracking the flow of information for objects, higher-order functions, exceptions, arrays as well as JS's API to the DOM.

6 Conclusion

We have presented a framework for the dynamic tracking of information flows across scripting exposed subsystems of a browser that allows precise labeling of values. To achieve this objective we added a DomainRegistry to the browser, modified the underlying JS engine and APIs to handle DOM and user events. We demonstrated that our framework is (1) able to detect information exfiltration attempts, (2) allows precise statistics about domains involved in an information flow violation, and (3) lowers performance overhead down to 83%. Thus, our system provides a major step towards precise and efficient information flow control in web browsers.

References

1. OWASP: The open web application security project. (`https://www.owasp.org/`)
2. Microsoft: Microsoft security intelligence report, volume 13. `http://www.microsoft.com/security/sir/default.aspx` (2012)
3. Jang, D., Jhala, R., Lerner, S., Shacham, H.: An empirical study of privacy-violating information flows in JavaScript web applications. In: Proceedings of the Conference on Computer and Communications Security, ACM (2010) 270–283
4. Vogt, P., Nentwich, F., Jovanovic, N., Kruegel, C., Kirda, E., Vigna, G.: Cross site scripting prevention with dynamic data tainting and static analysis. In: Proceedings of Annual Network and Distributed System Security Symposium. (2007)
5. Just, S., Cleary, A., Shirley, B., Hammer, C.: Information flow analysis for JavaScript. In: Proceedings of the ACM International Workshop on Programming Language and Systems Technologies for Internet Clients, ACM (2011) 9–18
6. Russo, A., Sabelfeld, A., Chudnov, A.: Tracking information flow in dynamic tree structures. In: Proceedings of the European Symposium on Research in Computer Security, Springer (2009) 86–103
7. Groef, W.D., Devriese, D., Nikiforakis, N., Piessens, F.: FlowFox: a web browser with flexible and precise information flow control. In: Proceedings of the ACM conference on Computer and Communications Security, ACM (2012)
8. Goguen, J., Meseguer, J.: Security policies and security models. In: Proceedings of IEEE Symposium on Security and Privacy, IEEE (1982)
9. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. In: ACM Transactions on Software Engineering and Methodology. Volume 9. ACM (2000) 410–442
10. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java information flow. `http://www.cs.cornell.edu/jif` (2001)
11. Hennigan, E., Kerschbaumer, C., Brunthaler, S., Franz, M.: Tracking information flow for dynamically typed programming languages by instruction set extension. Technical report, University of California Irvine (2011)
12. Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: You are what you include: Large-scale evaluation of remote javascript inclusions. In: Proceedings of the Conference on Computer and Communications Security, ACM (2012)
13. Hedin, D., Sabelfeld, A.: Information-flow security for a core of JavaScript. In: Proceedings of the Computer Security Foundations Symposium. (2012) 3–18