17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing

Web Application Security using JSFlow

Daniel Hedin
Mälardalen University
Chalmers University of Technology

Andrei Sabelfeld
Chalmers University of Technology

I. Introduction

This extended abstract accompanies a tutorial on web application security using JSFlow. The interested reader is encouraged to try the JSFLow tool [1] and get a full account of the theory and practice behind JSFlow, as detailed in a journal article [2], whose exposition we draw on in parts of this abstract.

Increasingly, web applications combine services from different providers. The script inclusion mechanism routinely turns barebone web pages into full-fledged services built up from third-party code. Such code provides a range of facilities from helper utilities (such as jQuery or Modernizr) to readily available services (such as analytics or ads). Even stand-alone services such as Google Docs, Microsoft Office 365, and DropBox offer integration into other services. Thus, the web is gradually being transformed into an application platform for integration of services from different providers.

At the heart of this lies JavaScript. When a user visits a web page, JavaScript code from different sources is downloaded into the user's browser and run with the same privileges as if the code came from the web page itself. This opens up for abusing the trust, either by direct attacks from the included scripts or, perhaps more dangerously, by indirect attacks when a popular service is compromised and its scripts are replaced by an attacker. A recent empirical study [3] of script inclusion reports high reliance on third-party scripts. As an example, the study shows how easy it is to get code running in thousands of browsers simply by acquiring some stale or misspelled domains and publishing the scripts of their choice at the URLs that are mistakenly used by developers.

One important aspect of many web application is sharing and collaboration. This is particularly evident in social media applications, like Facebook and Google+, or web applications for sharing different types of content, like Youtube, Imgur and Instagram. In addition, applications like Google docs, Microsoft 365 and Evernote support collaborative, and even simultaneous editing of documents. Hence, in addition to the content provided by the service itself, users will download and display content generated by other users. This opens up for direct attacks, where a malicious user actively injects the payload, as well as indirect attacks, where users are tricked into injecting malicious payloads — potentially as part of being subjected to an attack.

Another important component of many web applications is that many are cost-free. Instead of paying for using a service like Facebook or Youtube the service is paid for by displaying ads. In order to be able to create targeted dynamic ads, both the ad service and the ad itself must be able to inject and run executable content. This opens up for malicious ad providers to inject attacks [4].

Traditional protection mechanisms vary depending on the type of attack. To protect against malicious ads, a suitable technique is "sandboxing" by language subsetting and/or static and dynamic checks to create and enforce a sandboxed execution environment [5], [6], [7], [4]. However, due to the complexity and dynamism of JavaScript, historically, such sandboxes have been rather brittle, and there have been numerous reports of ways to circumvent the protection and escape the sandbox. To protect against user injected content the most prominent technique is to sanitize the user input to remove or render inert any potentially hidden executable code. However, the abundance of different character encodings in combination with the best effort nature of browsers make sanitization very difficult, as indicated by the fact that XSS attacks are still number three on the OWASP top 10 list from 2013 [8]. When it comes to abuse of trust or misuse of libraries, the current protection mechanisms have little to offer.

The traditional security mechanisms are often limited to some form of access control. The sandbox prevents the injected code from accessing parts of the web application it should not be able to access, and the sanitization aims at removing executable code — in a sense similar to total isolation. Similar to the situation with trusted code, if the sandbox or the sanitization process is bypassed, no security guarantees can be provided. Thus, in the presence of code injection, access control is not enough to guarantee information security [9]. Rather, in the presence of code injection, what the application does with sensitive information after access has been granted is even more important

Further, even if code injection is prevented, the integration of untrusted or partly trusted libraries is challenging with access control for the same reasons as above. Thus, there is a need for an approach beyond access control to deal with the root of the problem in malicious code: *insecure information flow*.

Information-flow control [10] is at the center of our approach. Rather than specifying what information an application or part of application can access, we suggest using security policies that specify what the application is allowed to do with the information it accesses. To enforce such security policies, we suggest the use of information-flow control to guarantee

978-1-5090-0461-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SYNASC.2015.11

that the application does not violate the set security policies.

II. Information-flow control primer

Traditionally, information-flow control distinguishes between *explicit* and *implicit* flows [11]. Explicit flows amount to directly copying information, e.g., via an explicit assignment like l = h;, where the value of a secret variable h is copied into a public variable l. Tracking only explicit flows is equivalent to *taint tracking* [12], a successful technique typically used to track and limit the propagation of untrusted (tainted) data in, e.g., web applications. Taint tracking has proved to be useful when the attacker is not in control of the program, but only of the input fed into the program. In the presence of malicious code or code injection in non-malicious code, however, taint tracking can be circumvented by implicit flows.

Implicit flows may arise when the control flow of the program is dependent on secrets. Consider, for instance, the following program:

```
if (h) { l = true; } else { l = false; }
```

Depending on the secret stored in variable h, variable l will be set to either true or false, reflecting the value of h. There is an implicit flow from h into l.

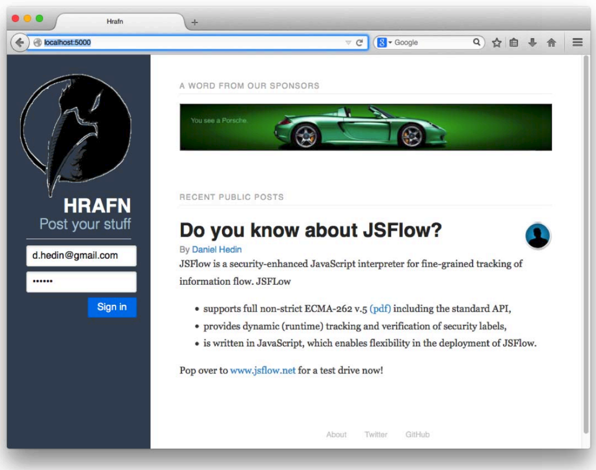
In order to handle implicit flows, a security level associated with the control flow is introduced, called the *program counter* level, or *pc* for short [11]. In the above example, the body of the conditional is executed in a secret context. The pc reflects the confidentiality of guard expressions controlling branch points in the program, and governs side effects in their branches by preventing modification of less confidential values.

III. JSFlow

JSFlow is an information-flow aware interpreter for full non-strict ECMA-262(v.5) [13], including information-flow models for the standard API. JSFlow is available online [1]. JSFlow is itself implemented in JavaScript. The choice of language allows for flexibility in the deployment. We have explored the possibility of deploying the interpreter via browser extension, via proxy, via suffix proxy, and as a security library [14]. It is also possible to use JSFlow on the server side by running on top of, e.g., node.js [15].

JSflow allows for the definition of policies that specify what information is allowed to flow where. The policies are formed by labeling information sources and sinks. One one hand, a security label on an information source, e.g., a password field, classifies the information originating from the source. On the other hand, a security label on an information sink, e.g., issuing a command to post information to a specific URL, classifies the maximum label of information that is allowed to flow to the sink.

Once the policies have been set, JSFlow monitors how information flows during the execution of the program and disallows flows that violate the security policy. In the current JSFlow Firefox extension, this is manifested by halting the execution and presenting a security alert to the user as illustrated in Figure 3. The user can then chose to allow the policy violation or terminate the execution.



Hrafn

localhost:5000

HRAFN

Post your stuff

d.hedin@gmail.com

••••••

Sign in

A WORD FROM OUR SPONSORS

You see a Porsche.

RECENT PUBLIC POSTS

Do you know about JSFlow?

By Daniel Hedin

JSFlow is a security-enhanced JavaScript interpreter for fine-grained tracking of information flow. JSFLow

- supports full non-strict ECMA-262 v.5 (pdf) including the standard API,
- provides dynamic (runtime) tracking and verification of security labels,
- is written in JavaScript, which enables flexibility in the deployment of JSFlow.

Pop over to www.jsflow.net for a test drive now!

Fig. 1. The Hrafn application.

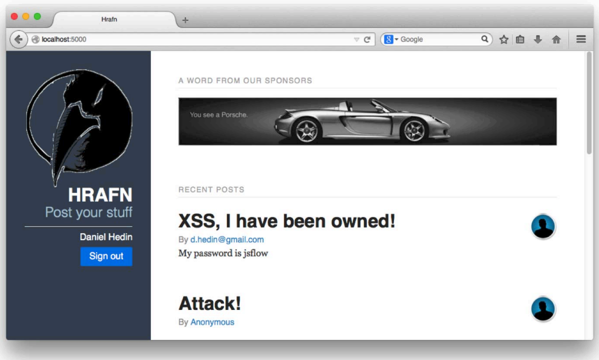
IV. Web applications security using JSFlow

Driven by the architecture of a typical web application, we focus on the scenario, where the attacker is able to inject malicious code into the application. By injecting code into the application it is possible for the attacker to access and steal all data the application has access to, unless additional security mechanisms are put into place.

As a basis for experimentation we have created an example web application, Hrafn, depicted in Figure 1. The application is constructed to be open for various code injection attacks, e.g., via buggy or compromised 3rd party services or via malicious user content.

Hrafn is an ad-financed lightweight forum. Users can view and post articles either anonymously or under their own identity, after having logged on their own account. When a user logs in, the credentials — the username and the password — are sent to the server, which, if correct, establishes an authenticated session. Hrafn has been built using the well know lightweight web application framework *express.js* [16] together with the industrial strength authentication middleware *passport.js* [17]. Thus, the flaws of the application do not originate from improper authentication and session handling. Rather, the flaws of the application come from the lack of proper sanitization of user input, and trusting a flawed ad-service. The flaws open up for two code injection attacks: 1) code injection via malicious ads, planted by customers of the ad service, and 2) code injection attacks via cross-site scripting (XSS), where a user crafts and posts a malicious article.

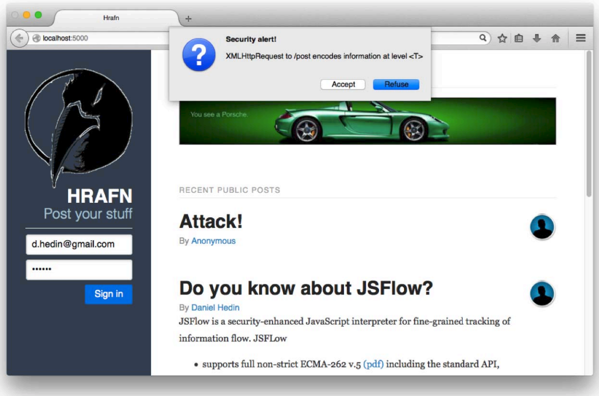
We have implemented two attacks — one for each category — that harvest user credentials as the user logs into the application. The first attack has the form of a specially crafted ad. The ad injects code that sends the username and password



XSS, I have been owned!
By d.hedin@gmail.com
My password is jsflow

Attack!
By Anonymous

Fig. 2. A successful XSS attack.



Security alert!
XMLHttpRequest to /post encodes information at level <T>

Attack!
By Anonymous

Do you know about JSFlow?
By Daniel Hedin
JSFlow is a security-enhanced JavaScript interpreter for fine-grained tracking of information flow. JSFLow

- supports full non-strict ECMA-262 v.5 (pdf) including the standard API,

Fig. 3. The XSS attack prevented by JSFlow.

to an attacker controlled server using XMLHttpRequests when the users log in. This models the scenario, where an attacker uses an ad service to serve malicious ads to harvest user information. The second attack is an XSS attack, where a specially crafted article is posted to the forum. The article displays as an ordinary article, but also injects code that makes users post their credentials to the forum when logging in. Figure 2 depicts the XSS attack. In the figure the attack article posted by an anonymous user with the title *Attack!* is visible as are the credentials of the recently logged in user.

Without information-flow control, both attacks successfully steal user credentials as unknowing users log into the application. To illustrate the power of information-flow control, we attach a natural security policy: that the password of the user should only be allowed to flow to the *login url* of Hrafn, hrafn.org/login[1]. In both cases, the attacks are thwarted by JSFlow under this intuitive security policy. In the first attack, an attempt is made to send the password to the attacker supplied URL. This violates the security policy and JSFlow prevents the information leak by halting the execution. In the second attack, an attempt is made to send the password to the Hrafn server, but not to the login URL. Rather, it is sent to hrafn.org/post. Again, this violates the security policy and JSFlow halts the execution as depicted in Figure 3. The latter attack illustrates the need for fine-grained security policies. To stop this attack it is necessary to distinguish between different URL on the same domain. Stating that passwords are allowed to flow back to the Hrafn domain is not sufficient. The Hrafn application, the attacks and the JSFlow Firefox extension are all available online [18].

V. Related Work

We discuss the most closely related work, referring the reader to a survey on language-based Information-flow security [10] for related work on information flow in genera and to the journal article [2] for related work on JSFlow in particular. Other surveys of relevance include overview of dynamic information-flow mechanisms by Le Guernic [19], a uniform presentation of information-flow security for a succession of increasingly powerful attackers by Hedin and Sabelfeld [20], and a survey on JavaScript-based security policies and enforcement by Bielova [21].

Vogt et al. [22] modify the source code of the Firefox browser to implement a flow-sensitive information-flow analysis to crawl around 1,000,000 popular web sites and, after white/black-listing 30 web sites, detect suspected attempts for cross-domain communication in 1,35% of the sites.

Mozilla's ongoing project FlowSafe [23] aims at giving Firefox runtime information-flow tracking, with dynamic information-flow reference monitoring [24] at its core. Our coverage of JavaScript and its APIs provides a base for fulfilling the promise of FlowSafe in practice.

Yip et al. [25] present a security system, BFlow, which tracks information flow within the browser between frames. In order to protect confidential data in a frame, the frame cannot simultaneously hold data marked as confidential and data marked as public. BFlow not only focuses on the client-side but also on the server-side in order to prevent attacks that move data back and forth between client and server.

Mash-IF, by Li et al. [26], is an information-flow tracker for client-side mashups. With policies defined in terms of DOM objects, the enforcement mechanism is a static analysis for a subset of JavaScript and treats as blackboxes the language constructs outside this subset. Executions are monitored by a reference monitor that allows deriving declassification rules from detected information flows. An advantage of this approach is fine-grained control at the level of individual DOM objects. At the same time, the imprecision of the static analysis leads to both false positives and negatives, opening up for attackers to bypass the security mechanism.

De Groef et al. [27] present *FlowFox*, a Firefox extension based on secure multi-execution and perform practical evaluation of user experience when simpler policies (such as labeling the cookie as sensitive) are enforced.

[1]Under the assumption that the domain of Hrafn is hrafn.org. It is not. For obvious reasons it would be a bad idea to make an intentionally flawed application available online.

Bichhawat et al. [28] present an information-flow analysis for JavaScript bytecode. The analysis is implemented as instrumented runtime system for the WebKit JavaScript engine.

Finally, an interested reader might benefit from trying out practical challenges on breaking and fixing information-flow protection for a succession of simple imperative languages [29] and a challenge that is based on JSFlow itself [30].

VI. Conclusion

Based on two practical attacks against the example application Hrafn, we have illustrated the power of code injection attacks. The attacks model the scenario, where the current standards protection mechanism are bypassed or not applicable. By using a simple and natural security policy we have shown how both attacks are stopped by JSFlow, an information flow aware interpreter for full non-strict ECMA-262(v.5) [13]. It is worthwhile to notice that, even though information-flow control has not been tailor made to stop this kind of attacks, it offers a uniform line of defense against the attacks, both in theory and in practice. This is in stark contrast to the current state of the art, which is based on numerous approaches to protection, one for each type of attack. This illustrates that shifting the focus from who can access what information to what is allowed to do with different pieces of information is not only fruitful, but effective, to ensure confidentiality of sensitive data.

Acknowledgments This work was funded by the European Community under the ProSecuToR and WebSand projects, and the Swedish agencies SSF and VR.

References

[1] D. Hedin, L. Bello, A. Birgisson, and A. Sabelfeld, "JSFlow," Sep. 2013, the JSFlow project, located at www.jsflow.net.

[2] D. Hedin, L. Bello, and A. Sabelfeld, "JSFlow: Tracking information flow in javascript and its APIs," *JCS*, 2015, To appear.

[3] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: large-scale evaluation of remote JavaScript inclusions," in *CCS*, Oct. 2012.

[4] J. Gibbs Politz, A. Guha, and S. Krishnamurthi, "Typed-based verification of web sandboxes," *J. Comput. Secur.*, vol. 22, no. 4, pp. 511–565, Jul. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2699784.2699787

[5] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Caja - safe active content in sanitized JavaScript," http://code.google.com/p/google-caja/downloads/detail?name=caja-spec-2008-06-07.pdf, Google Inc., Tech. Rep., Jun. 2008.

[6] M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan, "AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements," in *Proceedings of the 19th USENIX Security*, 2010.

[7] L. Meyerovich and B. Livshits, "ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser," in *Proc. of SP'10*, 2010.

[8] A. van der Stock, J. Williams, and D. Wichers, "OWASP Top 10 2013," http://www.owasp.org/index.php/Top_10_2013, 2013.

[9] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "JSFlow: Tracking information flow in javascript and its APIs," in *SAC*, 2014.

[10] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[11] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *CACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977.

[12] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld, "Explicit secrecy: A policy for taint tracking," in *EuroS&P*, 2016.

[13] ECMA International, "ECMAScript Language Specification," 2009, version 5.

[14] J. Magazinius, D. Hedin, and A. Sabelfeld, "Architectures for inlining security monitors in web applications," in *ESSoS*, 2014.

[15] Joyent, Inc., "Node.js," http://nodejs.org/.

[16] StrongLoop, Inc., "Express," http://expressjs.com/.

[17] Jared Hanson, "Passport," http://passportjs.org/.

[18] D. Hedin, "SYNASC'15 tutorial," Sep. 2015, tutorial, located at www.jsflow.net/SYNASC-2015.html.

[19] G. Le Guernic, "Confidentiality enforcement using dynamic information flow analyses," Ph.D. dissertation, Kansas State University, 2007.

[20] D. Hedin and A. Sabelfeld, "A perspective on information-flow control." in *Software Safety and Security*, 2012, pp. 319–347.

[21] N. Bielova, "Survey on javascript security policies and their enforcement mechanisms in a web browser." *J. Log. Algebr. Program.*, pp. 243–262, 2013.

[22] 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*, Feb. 2007.

[23] B. Eich, "Flowsafe: Information flow security for the browser," https://wiki.mozilla.org/FlowSafe, Oct. 2009.

[24] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, Jun. 2009.

[25] A. Yip, N. Narula, M. Krohn, and R. Morris, "Privacy-preserving browser-side scripting with bflow," in *EuroSys*. USA: ACM, 2009, pp. 233–246.

[26] Z. Li, K. Zhang, and X. Wang, "Mash-IF: Practical information-flow control within client-side mashups," in *DSN*, 2010, pp. 251–260.

[27] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "FlowFox: a web browser with flexible and precise information flow control," in *CCS*, 2012.

[28] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, "Information flow control in webkit's javascript bytecode." in *POST*, 2014, pp. 159–178.

[29] A. Birgisson and A. Sabelfeld, "Information Flow Challenge," Sep. 2012, located at http://ifc-challenge.appspot.com/.

[30] D. Hedin, L. Bello, A. Birgisson, and A. Sabelfeld, "JSFlow Challenge," Sep. 2013, located at http://www.jsflow.net/jsflow-challenge.html.