Preventive information flow control through a mechanism of split addresses

Deepak Subramanian (deepak.subramanian@supelec.fr)*
Guillaume Hiet (guillaume.hiet@supelec.fr)*
Christophe Bidan (christophe.bidan@supelec.fr)*

Abstract: The security of the web-browser and JavaScript is pivotal in today's world. The potency of information flow control in data leak prevention is very appealing. In this paper, we propose a new secure data flow model specifically designed for interpreted languages, more specifically, JavaScript. Our model relies on three elements for access control and one information flow element. We aim to implement this model by splitting variables and subsequently isolating sensitive variable address spaces.

Keywords: Access Control, Information Flow Control, Web Security

1 Introduction

The current state of the Internet and web technologies makes it extremely appealing to the vast majority of the computerized world to depend on purely on-line services. The advent of HTML5 has triggered an array of approaches increasing the feature set of web applications. Some of these novel technologies such as CORS, message passing, WebRTC or web sockets, allow for communication on web pages on levels that were not feasible earlier. Others such as IndexedDB and local storage increase client side storage capabilities. This has led to an increasing need to improve the security of the web-browsers. These techniques stress on the need to monitor these new possibilities of information leakages since the web pages are converging to the functionalities traditionally reserved for desktop applications.

Over the years, there have been a lot of improvements including the key conceptualization of "same-origin policy". Despite all these current safeguards, constant threats to web-services prevail, thereby, increasing the desire for security enhancements to help in the progression towards a better Internet. Some of the most prominent vulnerabilities such as cross-site scripting and cross-site request forgery feature in the OWASP Top Ten [OWA]. These attacks can be characterized by malicious information flows and hence information flow control is an effective methodology to counter them.

In this paper, we describe a pragmatic approach to information flow analysis on a web-browser. We propose a preventive mechanism for the analysis and enforcement of security on an interpreted language such as JavaScript where several activities are triggered by events.

* Supélec/INRIA, CIDRE team, France

The rest of the paper is organized as follows. The various related works are described in the section 2. The section 3 provides the details of our model and section 4 presents the conclusion for this work.

2 Related work

Information flow control (IFC) is a very important model for assuring that information traverses from the assured source to the designated destination without data leaks. The most popular IFC models are possibilistic approaches. A possibilistic approach is when the model's objective is to eliminate any possibility of leak if the leak is considered feasible by the model. In this approach, the information flow is marked and when undesired flow is reached, the information flow is stopped. In IFC, the concept of lattice based models is often used to present a set of labels that indicates the sensitivity of information. There have been several efforts in IFC as detailed by Sabelfeld and Myers [SM03]. These approaches often rely on non-interference property [SS98, Bie13] which states that no secret inputs to the program can influence publicly observed outputs. Formulated in terms of program executions, if the program is run with different secret inputs, while holding the public values fixed, the public output must not change [HS12a]. Termination-insensitive non-interference [AHSS08, Bie13, SM03] only gives a guarantee about terminating programs, ignoring that non-termination may leak some confidential information. A relatively recent work by Bielova [Bie13] provides a clear and comprehensive list of notable IFC techniques in the context of JavaScript. The author compares the various techniques with their formal guarantees, the types of analysis (i.e. static vs. dynamic approaches), the implementation strategies used and clearly enlists the salient points of each approach. This work has been instrumental in providing a complete picture of this research area thereby becoming a valuable stepping stone to the design of our approach.

IFC could also be done using probabilistic approaches. In this case, every information flow is not only marked but also trailed for information leakage and attained information leakage values. Once these values reach a threshold, information flow needs to be stopped. This provides direct advantages over possibilistic approaches by allowing for more fine-tuning thereby reducing the margin for over-approximation. Shannon Entropy [Sha48] still sets itself as a hallmark for several analysis techniques. The other techniques that have gained popularity include min-entropy [Ŕ61], guessing entropy [Mas94], marginal guesswork [Pli00] and gain functions [ACPS12]. Most approaches assume the existence of an oracle which tries to find the value of the variable when each information is disclosed. Alvim et al. [AAP10] make a comparative study on the various probabilistic information flow techniques that are currently being used. The work on gain functions by Alvim et al. [ACPS12] is noteworthy. This work focuses on gain obtained when a attacker is able to guess a part of a secret or a property of the secret and also assures improvements over min-entropy. The authors show how every bit exposed actually affects the resulting entropy and highlights the improvements in min-entropy that are required to provide a more accurate analysis. However, this approach in its current state is not easily malleable to analyze variables where different segments of the variable have different levels of information sensitivity. The use of marginal guesswork [Pli00], that is relevant to dynamic information flow analysis, shows a realistic information flow scenario where multiple queries are posted to a single oracle to determine the value of the variable. This approach is especially useful

to our proposal since we expect different segments of the variable to have different levels of information sensitivity. For example, in the credit card industry the PCI-DSS model has successfully increased the adaptation of the PAN truncation mechanism [Wikb, Wika] which makes sure that only the last four digits of the card number are advisable to be disclosed and even receipts need to adhere to printing at max the first six digits and the last four digits. Since, the various digits have varying weights and different disclosure levels. This can be configured using probabilistic information flow models in cases where a script may need to identify the card provider (first six digits) but further information regarding the card is not disclosed.

The work by Hedin and Sabelfeld [HS12b] distinguishes itself by providing a view of the traditional approach described by Sabelfeld and Myers [SM03] in the context of JavaScript. The authors make an interesting case for the problem of the information-flow being flow sensitive in JavaScript. This increases the need to keep track of changing labels throughout the execution which becomes tedious with pure static approaches. The authors hence suggest a dynamic approach that is comparable to the no-sensitive-upgrade proposed by Austin and Flanagan [Aus13]. The difference is that Hedin and Sabelfeld allow some upgrade instructions before the behest of the implicit information-flow. The authors provide valid arguments for the implicit control flows generated by JavaScript especially by the eval function. It should also be noted that this approach is formally proved by the authors to fulfill the guarantees of termination-insensitive non-interference. This work is relevant to our understanding of how the general language based IFC can be correctly adopted to a weak typed interpreted language. Our model produces a more pliable approach designed specifically for the JavaScript context and aims to be more effective in this context.

FlowFox is a real implementation of IFC on the Firefox web browser by De Groef et al. [GD12]. FlowFox uses the concept of secure multi-execution that was introduced by Devriese and Piessens [DP10]. In secure multi-execution, the information flow across labels is segregated at the process level by providing a separate process for each level of sensitivity. This concept relies on the execution of completely isolated processes. If various halting processes are handled properly with correct rules, a termination-sensitive non-interference is achieved. This property is a key differentiator for FlowFox. The various “policy rules” proposed by the authors provide a justifiable entry-point to the dynamic policies that we propose in this paper. The authors have also taken into account some of the events that may cause information flows such as key-press events. The authors believe in the need to change the JavaScript interpreter of a full fledged browser thereby realizing a more significant result due to the ability to monitor various factors such as performance, model verification and implementation results. This model is related to our work and is currently the preeminent model in web-browser IFC. Our work aims to achieve similar security guarantees as FlowFox, but with reduced time and space complexities.

The faceted approach that has been proposed by Austin et al. [AF09, Aus13] is the most similar approach to our proposal. The authors are targeting the achievement of the same security guarantees as FlowFox without requiring additional process based executions. The similarity between our approach and the faceted approach exists in the use of public and private side of values for each variable. Using such an approach allows isolation without the need to allocate a separate process. The faceted approach is verified formally to provide termination-insensitive non-interference. The authors have implemented the analysis using

a plugin for Firefox called ZaphodFacets which uses the Narcissus engine. Cross-site scripting is handled effectively by the faceted approach. The main difference between our approach and the faceted approach is the use of principals in the faceted approach and the nature of IFC. In the faceted approach, all the execution paths need to be evaluated. In our approach we choose the path of least required execution and evaluate only programatically necessary code. Such an exhaustive evaluation of the faceted approach results in the faceted value having nested facets. This directly results in the number of versions of the variable necessary being at least two in the faceted approach while it is strictly two in our approach. This is a major difference between the approaches thereby intuitively resulting in a reduced time and space complexity. The faceted design extensively uses the concept of principals which serve as intermediaries between the functions requesting access and the variables. Thanks to the notion of dictionaries, our approach can directly link functions to variables which we presume to be simpler than the notion of principals used in the faceted approach. The final and crucial difference is that our approach targets to be a probabilistic information flow approach and the faceted approach is a hallmark possibilitistic information flow approach.

3 Address-split design approach

Our approach focuses on typical web-browser which generally consists of the JavaScript engine and a rendering engine. The JavaScript engine is composed of the interpreter and the interface to the rendering engine. We believe that a greater amount of IFC could be achieved by directly modifying the interpreter. This is consistent with related research [Aus13, GD12].

In this section we introduce the address split model. The naming is in accordance with the model's general ideology of variable replication. The simple split is described in the Figure 1 below. In JavaScript all the variables are pointers and hence access to values is

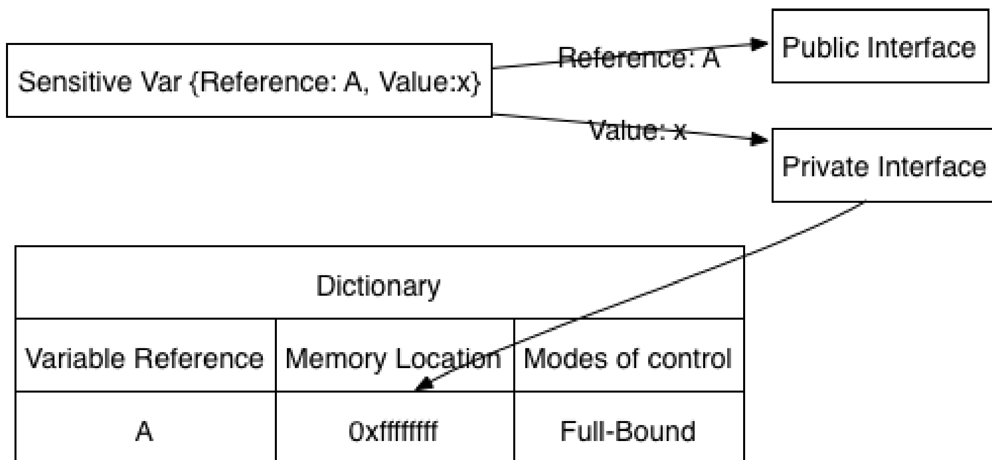

Fig. 1: The address split design

through reference. Let us consider a variable that contains sensitive information. This influences the decision of removing the variable from the easily accessible address space. To achieve this, we split the variable into two address spaces which we henceforth call as interfaces.

The public interface to the variable contains randomly generated dummy values. The private interface to the variable can be accessed only when the address space is known. This is managed at the interpreter end of the JavaScript.

Access to variables that are sensitive are given by the private interface to that variable. The rest of the model describes how this concept can be used effectively. To help with the usage of split addresses, we describe a host of elements and approaches that we adopt for a suitable information-flow theory to be present. The model is highly motivated by the need to control variable access with precise visibility and elegant failure of any functionality when access is not permissive. The components described in this model include the 3+1 modes of control, the types of functions, the concept of dictionaries and its various types, and the policies. The 3+1 modes of control, described in section 3.1, provide the details on the various modes of access control and IFC. Functions are objects where the modes of control are applied. Our approach requires functions to be segregated into varying types based on their required functionalities so that the information flow can be handled correctly as described in section 3.3. Dictionaries, described in section 3.4, are the components that implement the modes of control on the functions. They provide a means to access the private instances of the variable when the modes of control allow it. Finally, the policy specification, described in section 3.2, defines the modes of control on the objects to the policy engine which in turn implements the dictionaries.

To illustrate our approach, let us consider two common scenarios in modern day browsing. The ***password box scenario*** is one of the most common day-to-day sensitive information scenarios. The password box security often relies on a pure display-based security based mechanism. It aims at obfuscating visible output from the user. This mechanism does not provide any functionalities to limit scripts from accessing the actual password. The ***translator scenario*** is fast becoming a very common scenario. The presence of several human language make it a requirement to translate complete pages between languages and this poses the problem of passing sensitive information to untrusted servers. Translators have access to the entire page and monitor the page for any changes to translate instantaneously. A script typically accesses the website and monitors for changes. The entire text that needs translation is then sent to the translator servers and these are suitably translated.

3.1 3 modes for access control and 1 mode for IFC

The password box scenario shows the clear need for access control to be present. The password box still needs to be accessible to scripts to keep the existing functionality. However, it should be noted that not every property of the password box needs to be accessed by every script. For example, there might be scripts that require access to the style property of the password box but do not require its value. Similarly, the password box itself need not be available to unrelated scripts such as advertisements and social network feeds. To facilitate such an arrangement we have envisioned different modes of control.

The use of traditional access controls have been effective in various solutions such

as OS level file access rights. IFC models have been able to monitor the flow of data between components requiring different information sensitivity. Coupled together, the access control and IFC check if the flow of data is permissible and also maintain details on the sensitivity of data that is being transmitted. The "3+1 modes of control" is one such model combining the best of both worlds. The modes of control are given to functions to access variables. The IFC affects these privileges making the privileges pliable. We take this stance of pliability to keep up with the chaotic environment of JavaScript.

We propose three main types of access control: Full-Bound Access, Semi-Bound Access and Instance Access. The first two access control types represent the ideology behind traditional access controls while the third is specific to this model to handle temporary access conditions. A full-bound access is precisely like the name suggests. The function is permitted to perform both read and write operations. A semi-bound access is given to the function if it needs either read-only or write-only access to the variable. The mechanism gives direct access to the variable for the function.

Let us consider the password box scenario. There are plugins which help in auto-filling passwords. These functions might require full-bound access to the password box. For example, the function that submits the password field to the required destination needs semi-bound read-only access to the password box. The other functions do not require any access to the password box.

An instance can be defined as that point in time for the program where a set of conditions such as event triggers, function calls or variable values are met. Instance access is a special mode we propose to ensure that the value of a variable is accessible only at a particular instance. It must also be noted that the function never gets direct access to the variable but only to its copy at that instance.

Let us consider the translator example. Sometimes the web-page being translated might require sensitive input not intended for the translation server. This might be a page with a credit-card payment step. In this scenario, the card information is accessible to the translation server. The expected behavior would be to translate only the static page before the sensitive fields are filled in and then to stop the translation as soon as the keypress event in any of these fields is triggered. The translator would not crash since its copy remains unchanged and this functionality resumes on page load.

Transitive information flow represents the IFC element of the 3+1 modes of control. This is a complementary mode of control that is present to provide for a probabilistic approach to the problem. The transitive information flow can be defined as the amount of data belonging to the variable that can be declassified by the function without requiring additional privileges. While this is the disclosure entropy of the variable, the calculation of such entropy must take into account that some digit of the variable could be more important than the other and some patterns could be expected. For example, in a URL, the string before the GET parameters is relatively constant. In case of basic authentication, the user name and password can be part of the url string itself in the form of http://username:password@url?params. Such parts of the url need to be secured despite possible information leakage. The transitive model aims to do exactly that. Specifying unattainable weights for such digits in the string, one can ensure that the scenario is never reached where any character of the sensitive segments are leaked.

3.2 Policy Specification

Policy specification lists the set of policies that determine the IFC to be emphasized. The policies themselves help in the analysis of the information flow and provide proper basis for legitimate information flows. The policy specification is envisioned to be evolving with some machine learning as part of our future work. The policy specification can be augmented by the web browser user, or developer and there are background mechanisms in this model that also influence the policy specification. In this model, we assume that the web browser users' policies takes precedence over all other policies.

3.3 Functions

Functions are the objects where the modes of access are applied. This approach requires functions to be segregated into varying types based on their required functionalities so that the information flow can be handled correctly. The various functions can be classified into four types based on how the IFC needs to be handled: self-sufficient functions, utility functions, inheritance functions and guest functions.

The self-sufficient functions are the functions whose policies are explicitly defined by web browser users or web page developers as shown in section 3.2. It must be noted that the self-sufficient functions can never take the privileges of their caller. Moreover, all exceptions generated in a self-sufficient function can only be handled within its own scope. If the exception cannot be handled within the scope, it is handled by an empty exception handler that does nothing and does not pass the exception over to its caller. This allows for functions to be handled with the privileged assigned to them without unintended leak of information. The self-sufficient functions are envisioned to provide a means to assign privileges to functions. Let us consider the password box scenario. The function which needs access to the password box should be a self-sufficient function.

By default, a function is considered to be a utility function if it does not feature in the policy specification. A utility function the truest form of what a function signifies. It is considered as a modular piece of code that has been made into a function for easier maintenance and reuse. Considering this, the utility function does not have any privileges of its own. Every instance of a utility function adheres to the privileges of its caller. This behaviour is transitive over the function call. Let us consider the password box scenario. The function which has access to the submit button may have to send the password to a server over a POST request using the jQuery library. In this case the jQuery library's POST function would be considered a utility function so that it can perform the intended action as long as the caller function has the privileges to read the password.

Inheritance functions are a type of function that are present due to the nature of JavaScript that allows the creation of functions at runtime. In our approach, the functions that are created at runtime are defined as inheritance functions and have three important restrictions:

1. The inheritance function may not own more privileges than its parent.
2. The inheritance function will continue to be bound to its parent when things change. For example, if the parent drops privileges, the inheritance function may no longer have them. Similarly, if the parent function is deleted/destroyed, the inheritance function will loose all privileges. It will not become a utility function (unless it already was a utility function).

Guest functions are a special type of function which can use the privileges of a particular self-sufficient or inheritance function as a guest when specific conditions such as variable value, event trigger are met. One of the applications of this functionality is in the case of asynchronous communications and JSONP generated as part of a utility function. The privileges granted to the guest functions are temporary. This type of function is very useful since it allows us to permit for temporary privileges to be assigned to functions with definitive limitations. Let us consider the password box scenario. When the user name is being typed, there are options that pop-out from auto-fill elements. If the user selects one of these user names, the password needs to be auto-filled but not until then. Hence a guest access can be provided to write to the password field for one time when the user name is selected from the drop-down. This protects the field while preserving the functionality.

3.4 Dictionaries

To implement the different modes of access on functions to variables, our approach relies on a dictionary. Dictionaries provide a means for the functions to access the private instances of the variable. Each dictionary contains the variable reference as the key and a memory location as a value. The dictionaries provide data on the variable based on the various modes of control. There are a few types of dictionaries based on their purpose: private dictionaries, group dictionaries, subset dictionaries, and instance dictionaries.

Private dictionaries are exactly as the name suggests. They are private to the function they belong to. Every function has its own private dictionary. A group dictionary serves the purpose of a label based access. The label based access allows the functions to have common modes of control. If grouping were required, for example, in the case of multi-level models, such a grouping could be incorporated using the concepts of group dictionaries. The group dictionaries can be accessed by any number of functions as long as they have access to this dictionary. Subset dictionaries are those which represent a smaller set of private interfaces which can be accessed by their superset dictionary. Incidentally, the group dictionaries are usually a subset to the private dictionaries of the functions accessing them.

The main purpose of the instance dictionary is to provide the value in case of the instance access type. The instance access is handled when the instance occurs and hence there is a need to update the values at the particular points of access. This point of access is also known as the instance dictionary and this dictionary is generally a subset to the private or group dictionaries.

4 Conclusion

The dynamic nature of JavaScript makes it necessary for novel approaches to address the information leaks. We propose a model that uses policy specification in coherence with dictionaries and the address split design. Our objectives are to overcome several web-vulnerabilities and to assure the best compromise between functionality and security. The proposed model is to split the variables into interfaces that can be accessed as needed and the model also ensures direct memory mapping therby intuitively having a lower latency. The address split design is capable of handling various scenarios and also provides the key concepts required to control the information flow in a JavaScript enviroments. This

paper provides a viable model to achieve good security without the need to have numerous processes as per the secure multi-execution models or exponential growth in number of facets in each variable. It also takes into account both information flow control and access controls. The future work of the model is to work towards improving the policy specification, reinforcement learning model, and a Chromium implementation.

Acknowledgments

This work is fundeded by Supelec, CominLabs SECCLOUD working group and the administrative region of Brittany. We also thank the CELTIC and ASCOLA teams in the SECCLOUD project for their valuable inputs on our ideas.

References

[AAP10] Mário S. Alvim, Miguel E. Andrés, and Catus Palamidessi. Probabilistic Information Flow. In *25th Annual IEEE Symposium on Logic in Computer Science*, pages 314–321. IEEE, July 2010.

[ACPS12] Mário S. Alvim, Kostas Chatzikokolakis, Catuscia Palamidessi, and Geoffrey Smith. Measuring Information Leakage Using Generalized Gain Functions. In *2012 IEEE 25th Computer Security Foundations Symposium*, volume 0, pages 265–279. IEEE, June 2012.

[AF09] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. *ACM SIGPLAN Notices*, 44(8):20, December 2009.

[AHSS08] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security*, pages 333 – 348. Springer-Verlag Berlin, Heidelberg, 2008.

[Aus13] TH Austin. *Dynamic information flow analysis for Javascript in a web browser*. PhD thesis, University of California, Santa Cruz, 2013.

[Bie13] Nataliia Bielova. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *The Journal of Logic and Algebraic Programming*, 82(8):243–262, 2013.

[DP10] Dominique Devriese and Frank Piessens. Noninterference through Secure Multi-execution. *2010 IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[GD12] Willem De Groef and Dominique Devriese. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 748—-759, Raleigh, North Carolina, USA, 2012. ACM.

[HS12a] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In Benedikt Hauptmann Tobias Nipkow, Orna Grumberg, editor, *NATO*

Science for Peace and Security Series - D: Information and Communication Security, volume 33: Software Safety and Security, pages 319 – 347. IOS Press, 2012.

[HS12b] Daniel Hedin and Andrei Sabelfeld. Information-Flow Security for a Core of JavaScript. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 3–18. IEEE, June 2012.

[Mas94] James L Maseey. Guessing and Entropy. In *Proceedings of International Symposium on Information Theory*, page 204. IEEE, 1994.

[OWA] Category:OWASP Top Ten Project - OWASP.

[Pli00] John O Pliam. On the Incomparability of Entropy and Marginal Guesswork in Brute-Force Attacks. In *Progress in Cryptology - INDOCRYPT 2000, First International Conference in Cryptology in India*, pages 67–79, 2000.

[R6́1] Alfréd Rényi. On measures of entropy and information. In *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 547–561, 1961.

[Sha48] CE Shannon. A Mathematical theory of communication. *Bell System Technical Journal*, 27:379–423,625–656, 1948.

[SM03] Andrei Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[SS98] Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher-order and symbolic computation*, 14:40–58, 1998.

[Wika] Bank card number.

[Wikb] PAN Truncation.