International Research Journal of Computer Science (IRJCS)
Issue 8, Volume 2 (August 2015)
ISSN: 2393-9842
www.irjcs.com

A Critical Review of Information Flow Analysis Tools Aiming at Detecting Potential Vulnerabilities in JavaScript Code

Amr A. Mohallel
Department of Computing, Engineering and Technology
University of Sunderland, UK

Abstract— Information flow analysis (IFA) tools have proved its effectiveness towards tracing information flow in order to prevent private information from being exposed to public attackers. This paper critically reviews and evaluates the latest information flow analysis tools used to detect vulnerabilities in JavaScript code. Both static and dynamic systems are defined, compared, and evaluated in terms of both efficiency and effectiveness. Conclusions are drawn showing the advantages of dynamic IFA systems over static ones. Recommendations are given regarding which dynamic IFA systems works best with securing users online.

Keywords— JavaScript security, information flow analysis, Staged Lambda JS SlamJS, VEX, Gatekeeper, SIFEX

I. INTRODUCTION

While web applications are becoming larger and more complex, the need for JavaScript (JS) and other client-side languages increases. According to Google (2010), 92 out of the top 100 visited websites include advertisements. Such advertisements are normally offered by third-party providers who provide the ad as a block of JS code.

Chugh, Ravi et al. (2009) reviewed Google's research on ads security and concluded that the majority of the techniques used by attackers are by embedding JS code within the trusted advertisements to redirect the user to a malicious untrusted website. Google never intended to give the attacker such privileges, neither the main website vendor did. Furthermore, such ads, widgets, add-ons, or plugins are used in the majority of websites nowadays. The problem is that they all contain JS generated from third-party sources; therefore, they should all be considered insecure.

Each browser has its own techniques and procedures to prevent the extensions/plugins from gaining access to any unauthorized data. Mozilla implements the SABRE system that blocks plugins from accessing any sensitive data [10]. Mozilla also offers some functions to help developers keep the inserted JS code isolated from accessing the main page's properties i.e. Mozilla's *evalinsandbox*. On the other hand, Google Chrome rechecks the whole extension to keep it isolated from gaining the browser's privileges.

The main problem with SABRE and its equivalences is that they add additional code blocks to the original JS code causing serious delays in compilation speed. Added blocks may also conflict with the JS functions leading to getting the whole program bugged. For these reasons, information flow analysis is used.

Information flow analysis (IFA), also known as *non-interference* is applied when a program gives the same output level of security as the input [11]. IFA tools are expected to guarantee that the input of a certain program that is considered as low-level inputs (untrusted/ unauthenticated) must not have access to high-level outputs [12]. For example, when a low-level user inputs data into a certain webpage, he should only retrieve data that matches his authentication level. Policies are set to define which data are considered public, and which are considered private for a certain user/set of users [11].

This paper discusses how such attacks could be blocked using IFA tools. Non-interference can either be done by checking for information flow paths before the code is executed by the user (i.e. static IFA), or by checking the information flow paths while the code is being executed on run-time (i.e. dynamic IFA). In sections 2 and 3, latest static and dynamic IFA tools will be reviewed followed by a critical evaluation for each.

II. STATIC INFORMATION FLOW ANALYSIS

In this section, we will review five of the most common static information flow analysis tools used by website developers and browser vendors.

A. Static Information Flow (SIF) Analysis Framework

Chugh, Ravi et al. (2009) introduced and implemented SIF that is a framework for staging integrity of information flow properties in webpages in order to handle and perform security checks on retrieved JS code. Experiments that are made on real-world websites returns positive results proving how effective SIF is in terms of data privacy protection without affecting JS functionality. SIF mainly separates any JS code into two parts: *context*, and *hole*. *Context* is the static JS functions that are written by website developers between any <*script*> tags. While *hole* is the part of code that is generated dynamically and is considered anonymous for the website - this part was their biggest concern. *Hole* code is normally left to be written by a third-party that is powered by ad providers (such as Google Adwords). Three stages are followed by SIF to audit *hole* code and make sure that none can have access to the main *post* parameter. Gaining access to *post* would redirect the main page address – leading to the possibility of information leakage.

The SIF framework enables the *context* developer to define some policies which grant/ revoke privileges to/ from the *hole* code. Each policy element defines what code is disallowed to be executed by the *hole* developer. A policy element is defined as a pair (e.g. *"eval, document.location"*). In the previous policy, the *context* developer disallows the *eval* flow to affect the *document.location* property. This will prevent any *eval* function from redirecting the target website at run-time.

In the SIF approach, the code is pre-analysed statically before it is executed. Since the *hole* code may contain another *hole* codes generated at run time, the staged information flow recursively checks for the inner *holes*. Simply, if the code passes the policy restrictions, it will be allowed to run, otherwise, it will be blocked. Computing residual policy is reached by using a static constraint-based analysis to see which code would affect a higher level of properties. The residual policy states that no variable declared in the *hole* should flow into any other variables [1].

B. Gatekeeper

Microsoft's *Gatekeeper* is a mostly-static approach created to impose security and reliability policies to JS code before it is run. It focuses on the most suspicious JS functions as *eval, innerHTML, post,* and *document*. *Gatekeeper* is based on nine pre-defined policies. Each given JS function should bypass these policies in order to be executed. *Gatekeeper* ensures that the JS code within any widget is neither harmful, nor poorly written. The policies are highly extendible; they are written as a set of queries that prevent any code injections. To be followed, is a brief definition of each of the nine policies.

Since popup forms, pages, and alerts may cause serious disturbance for any browser user; the first policy ensures that no JS alerts or popups should be fired during the widgets runtime unless verification checks are made. The next policy ensures that the widget code does not affect the main head function properties (e.g. *window, document, array,* etc.). The third policy blocks malicious <*script*> calls. Next policy audits the current page *location* changes. One of the biggest challenges they faced while implementing this policy is that some JS functions (e.g. *document.location*) is normally permitted to <*script*> blocks. Such code not only allows malicious -probably sniffing- websites to popup, it also opens the door for *cross-site scripting* (*XSS*) attacks. Policies 5 and 6 focus on the *Live.com* widgets. The first policy blocks the use of *XMLHttpRequest*. A Mozilla research showed that using *XMLHttpRequest* allows dangerous request headers to be set [9]. In addition, using the same namespace name in a given widget as the one used by the target website could cause conflicts during runtime. Some websites use their own naming criteria to avoid such conflicts, but *Gatekeeper* solved the problem by renaming any given namespace to a custom pattern-based name.

Gatekeeper was tested on over 8000 JS widgets (mostly from *Google Adwords* and *Live.com*) and returned 1341 confirmed warnings and policy breaches [2]. *Gatekeeper* showed a great success in prohibiting JS malicious attacks.

C. Vetting Browser Security for Security Vulnerabilities (VEX)

Extensions/Plug-ins are JS programs that are installed on/ attached to a browser. Therefore, they not only gain access to all the contents of any HTML document that is run within the browser. Even if an extension/plugin developer has no harmful intents, if the JS code is poorly secured, attackers can use them to gain access to the same privilege level given to the extension from the browser itself (which would enable them to save/extract data from/to the user's machine).

For these reasons, the VEX framework was proposed. VEX is a framework that statically analyses information flow for JS code to check for security threats in any given browser extension/plug-in. VEX can be used to manually download an extension/plugin, extract its code, and analyse it for unsafe or malicious patterns. It then gives a feedback whether the given extension/plugin is safe or not according to some given outlined steps [3].

D. SIFEX

Agarwal, S (2010) investigated SIF, Gatekeeper, and VEX and came up with the SIFEX tool that is used by developers in order to build a more secure extension/plugin. SIFEX testing results came up with more potential threats than VEX discovered solely. When SIFEX was reviewed and tested, authors confirmed the ability of SIFEX to detect the previously unknown exploitable vulnerabilities.

E. Staged Lambda JS (SLamJS)

Lester, M et al. (2013) developed a static IFA tool that audits dynamically-written JS code using stage-based metaprogramming. They called it SlamJS as "it exhibits a number of JavaScript's interesting features in an idealised, lambda calculus-based setting" [12]. Their main focus was on having the *eval* constructors watched. Their research showed that recent surveys are proving the extensive use of *eval* in modern websites. *eval* is not a simple JS operator, it actually executes one or more JS functions/statements, then returns/prints out the final output. The *eval* constructor usually builds its JS syntax dynamically, which makes it one of the most important concerns when dealing with JS security. The SlamJS research also stated that *eval* can either be used directly by retrieving *JSON* formats or *XML*, or be used to execute complex sentences dealing with higher order functions. Such sentences can be injected within *eval* strings as follows:

eval("x=5;y=3;document.write(x-y)");

For such reason, SLamJS focused on IFA even when the input is a pure string. This would not only help impeding *eval* attacks, but any other form of code injections (e.g. *SQL Injection, XSS*, etc.). In summary, SLamJS converts the given code to stages, and then statically analyses each stage [12].

III. DYNAMIC INFORMATION FLOW ANALYSIS

Shroff, P et al. (2008) introduced a remarkable comparison between static and dynamic IFA in terms of path determination. They showed the main advantages of dynamic IFA systems over static ones. Here is a simple example of those they gave as an illustration to prove their point of view:

x := 0; if l < 10 then x := h else (); output(deref x);

Suppose that (*h*) is a high-level data and *(l)* is a low-level one. In the above code, whenever *(l)* is less than 10, *(x)* will be assigned to a high-level data (*h*), which is insecure since the input level is low. If such code is analysed by a static IFA system, it will mark the whole code as insecure, while dynamic systems will only block the execution of the *" x := h "* part only when *(l)* is less than 10. They also offered other examples showing indirect assignment of high-level data that is not detected by static systems [6]. For these reasons, they developed a run-time dynamic IFA system that is based on tracking indirect assignments between high and low-level data explained in the following paragraph.

The authors had a definite goal behind their research on dynamic IFA systems i.e. to develop a run-time based analysis to detect direct and indirect critical information flow for any JS code that is about to be executed and let the rest of the code to function. They proposed and implemented an algorithm that supplies their system with all the dependency information between program points. Given these dependencies, their system will be able to capture indirect information flow between program points. This enables them to capture the critical path flows as and when the code is to be executed.

They offer two techniques to capture these dependencies: one is totally dynamic that tracks dependencies on run-time, and the other one is completely static having a set of dependencies that can run on the initial code before it is executed. They discussed further noting that their static system will fail to capture critical paths before the code is executed, while the dynamic system will detect them while the code is being executed. Their approach has proved to reach non-interference between high and low level data.

Austin, P et al. (2012) stated that the key challenge in dynamic IFA lies in handling implicit flows. They were motivated by this uprising problem after getting dynamic IFA systems reviewed and tested. As a solution, they introduced the *"faceted values"* that is a new mechanism to convert any value in any function into two *"faces"*. An example of this faceted value is getting a value *(V)* which is shown in the following expression:

(k ? V_H : V_L)

Where *(k)* is a property that is assigned with the input level (indication low/ or high). So, if the user is authorized to access high level data, *(V_H)* is returned, otherwise *(V_L)* is returned. Given the fact that *(V_L)* is either a dumb data or the initial value of *(V)*. Therefore, they solved the main problem with static and dynamic systems, as any code will never be stopped, but rather return low-level/or dumb data if the user is not authorized. They offered remarkable more complex algorithms to handle implicit flows of code.

IV. Evaluation

SIFEX has detected 142 alerts when it tested about 3 million LOC, when VEX showed 50 alerts only [4]. This proves how effective SIFEX is when compared to the rest of static IFA tools.

Due to the flexible nature of JS, developers can give an order to a function indirectly by using constructors as eval. Such attempts would be impossible to be checked by static code tracing, as their threats only appear during run-time. This proves how efficient dynamic IFA systems can be when compared to static ones.

The following table gives a brief comparison between the two most commonly used dynamin IFA techniques (i.e. dynamic capture of dependencies, and the faceted values mechanism) given the following JS function:

TABLE I
COMPARING OUTPUTS OF DYNAMIC CAPTURE OF DEPENDENCIES TO FACETED VALUES MECHANISM

Authorized User?	**Dynamic capture of dependencies**	**Faceted values mechanism**

Yes	x=false;
	if (h<1) x=true;
	else return x;
		x=false;
		if (h<1) x=true;
		else return (k ? true : false);

Final output:	True	true

No	x=false;
	*Error
		x=false;
		if (h<1) x=true;
		else return (k ? true : false);

Final output:	*Error	false

As shown in the previous table, the function does not quit when an unauthorized user tries to access a high-level data, instead, it returns the initial value (or dumb data otherwise).

V. Conclusions

In this paper, the current IFA tools used to detect potential vulnerabilities in JS code were reviewed. Although static IFA systems proved its efficiency in detecting malicious code; they still fail to detect complex information flow paths (such as those discussed in section 3), and since modern websites nowadays use high-level JS functions that are sometimes run asynchronously; therefore static systems are individually not a valid technique to rely on when implementing security checks that cover all the information flow paths with the desired reliability. In addition, using static IFA systems needs manual human checking. Moreover, static IFA tools blocks the whole extension/plugin from being executed if a malicious flow is detected, while dynamic IFA only blocks the malicious path as and when they are executed – leaving the rest of the code to execute. This gives the dynamic systems an advantage. In addition, dynamic IFA systems have proven to detect more expected malicious information paths than static systems do.

SLamJS is of the latest researches made to reduce the gap between static and dynamic information flow analysis as it is a staged meta-programming tool that recursively check for run-time generated code.

Although both static and dynamic IFA systems need an overhead processing time to function, it is a price worth paying if they managed to keep private information secured.

VI. FUTURE WORK

- Based on the conclusions, IFA systems are expected to secure websites users from being attacked. Although current dynamic IFA tools are effective, but they still need to have more complex algorithms that uses less overhead processing speed, while auditing every flow of data, leading to less information leakage.
- We believe that Gatekeeper's nine policies should be treated as a guideline for software developers aiming at creating highly secured web systems.
- Although web developers have a big responsibility towards keeping home users safe, some JS functions as *eval* are impossible to be checked by web developers. This leads to giving the browser vendors the biggest responsibility towards maintaining a secure navigation for home users online.

REFERENCES

[1] R. Chugh, JA Meister, and S. Lerner. "Staged Information Flow for Javascript", in *ACM SIGPAN Notices*, 2009, vol. 44(6).
[2] B. Livshits and S. Guarnieri, *Gatekeeper: Mostly static enforcement of Security and Reliability Policies for Javascript Code*, 18th USENIX Security Symposium, USENIX Association, 2009, vol. 151.
[3] S. Bandhakavi, N. Tiku, W. Pittman, S. King, P. Madhusudan, and M. Winslett, *Vetting Browser Extensions for Security Vulnerabilities with VEX*, Communications of the ACM, 2011, vol. 59(6).
[4] Shikar Agarwal (2010) Annual Computer Security Applications Conference on *ACSAC*. [Online]. Available: http://www.acsac.org/2010/program/posters/agarwal.pdf
[5] (2011) Google. [Online]. Available: http://www.google.com/adplanner/static/top1000/index.html
[6] P. Shroff, S. Smith, and M. Thober. *Securing Information Flow via Dynamic Capture of Dependencies*, Journal of Computer Security, 2008, vol. 16(5).
[7] J. Magazinius, A. Russo, and A. Sabelfeld. "On-the-fly Inlining of Dynamic Security monitors" in *IFIP/SEC Security & Privacy – Silver Lining in the Cloud, Computer & Security*, 2012, vol. 31(7)
[8] T. Austin and C. Flanagan, "Multiple Facets for Dynamic Information Flow", in *ACM SIGPLAN NOTICES*, 2012, vol. 47(1).
[9] Darin Fisher (2005), "Bugzilla@Mozilla Bug 302263" on Mozilla Website. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=302263
[10] M. Dhawan and V. Ganapathy, "Analyzing information flow in Javascript-based Browser Extensions," in *ACSAC'09: Proceedings of the 25th Annual Computer Security Applications Conference*. Rep. 382-391, 2009.
[11] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. "Reactive non-interference for a browser model", in *5th International Conference on Network & System Security (NSS)*, 2011.
[12] M. Lester, L. Ong, M. Schaefer (2013) ARXIV on Cornell University Library Website. [Online]. Available: http://arxiv.org/abs/1302.3178?

