Detection and Mitigation Of Malicious JavaScript Using Information Flow Control

Bassam Sayed, Issa Traoré, and Amany Abdelhalim
Department of Electrical and Computer Engineering
University of Victoria
Victoria, BC, Canada
bassam,itraore,amany@ece.uvic.ca

***Abstract*—JavaScript is the main language used to provide the client-side functionality of the modern web. It is used in many applications that provide high interactivity with the end-user. These applications range from mapping applications to online games. In recent years, cyber-criminals started focusing on attacking the visitors of legitimate websites and social networks rather than attacking the websites themselves. The dynamic nature of the JavaScript language and its tangled usage with other web technologies in modern web applications makes it hard to reason about its code statically. This poses the need to develop effective mechanisms for detecting and mitigating malicious JavaScript code on the client-side of the web. In this paper, we address the above challenges by developing a framework that detects and mitigates the flow of sensitive information on the client-side to illegal channels. The proposed model uses information flow control dynamically at run-time to track sensitive information and prevents its leakage. In order to realize the model, we extend the operational semantics of JavaScript to enable the control of information flow inside web browsers.**

***Keywords*-Information Flow Control, Client-side web attacks, Web 2.0, AJAX, Malicious JavaScript**

I. Introduction

Web-based attacks can generally be classified as server-side attacks or client-side attacks. Server-side attacks exploit vulnerabilities in server-side web application components causing harm to the organization hosting such servers. A significant amount of research has been produced on how to secure the servers hosting sensitive information. The focus on protecting the server-side and the emergence of Web 2.0 technologies made the attackers switch focus to the client-side. Instead of attacking directly the servers of an organization, the focus is now primarily on attacking clients inside such organization.

There are two categories of malicious websites. Firstly, a malicious website can be a legitimate website that has been attacked to host the malicious content, like the cross-site scripting (XSS) worms Samy and Mikeyy, which attacked My Space and Twitter, respectively [1], [2]. Secondly, the visited website could host the malicious content intentionally to attack the end-users. Usually, the end-user is tricked to visit these malicious websites through some form of social engineering medium, such as an email or a message posted in a forum with a link to the malicious website.

Client-side attacks can be widely categorized as either browser-specific or browser-agnostic attacks. Browser-specific attacks are the attacks that target a specific type of browser on a specific platform. For instance, attacks that rely on Microsoft ActiveX components are only possible on Windows platform running Microsoft Internet Explorer. If the end-user uses other types of browsers (e.g. Mozilla Firefox), the attack will not be successful. On the other hand, browser-agnostic attacks do not depend on a specific type of browser or platform. These types of attacks take advantage of the fact that all the browsers running on any type of platform (even mobile platforms) have to support specific set of web standards such as HTML, JavaScript, CSS, etc. For instance, cross-site scripting attacks that steal the end-users's cookie do not rely on a specific type of browser or platform since all major browsers have to support cookie mechanism to function properly. Browser-agnostic attacks are the stealthiest and hardest to detect.

One of the enablers of client-side attacks is the same-origin-policy (SOP), which is a common policy enforced by any browser. The same origin policy states that the browser should allow communication of sensitive information only to the website the current rendered page was downloaded from. At the same time, the browser should allow downloading some of the web contents such as images from any domain, even if it is different from the current visited domain, which contradicts the same-origin-policy and makes it ambiguous.

The usage of JavaScript as a dynamic scripting language combined with the ambiguous same-origin-policy lead to the failure to enforce adequate information flow policy. The structure and mode of operation of JavaScript provide a fertile ground for conducting client-side attacks. Furthermore, the central role played by JavaScript in the Ajax platform, makes it the main source of most Ajax-based attacks.

Our goal in this research is to develop a new approach to detect stealthy web attacks that steal information about the end-user without her consent. Attacks like XSS, CSRF, and online history sniffing (online tracking) prove to be the hardest to detect as they happen inside the web browser. To our knowledge the only existing work that attempt to address the previously mentioned challenges in detecting web attacks was carried out by Vogt and colleagues [3]. In [3], Vogt *et al.* developed an approach that combines both dynamic

data tainting and static analysis to detect XSS attacks on the client-side.

Although dynamic data tainting was used successfully by Vogt *et al.* in their proposed model, it suffers from some limitations. Firstly, it does not have the flexibility to define multiple security classes which could affect the applicability of the model in different contexts. Secondly, unless formally defined, the model cannot show precisely how the tainted information propagates throughout the system. Thirdly, the taint checking is based on static rules that are domain specific and cannot change dynamically. Fourthly, dynamic data tainting does not clearly define how a tainted data could be untainted (or declassified).

Despite the fact that the work of Vogt *et al.* represents a valuable contribution in securing client-side web context, it allows preventing only one type of attack, namely, XSS. Furthermore, the authors have not described formally how the tainted data is introduced, propagated, and checked in the proposed system. We believe it is essential to precisely describe the dynamic taint analysis in a formal manner and show how exactly the taint propagation happens when the JavaScript code is executed. In this paper, we tackle such challenges by proposing a more formal, and flexible information flow control model than the dynamic taint analysis approach used by Vogt *et al.* The proposed model has the following characteristics:

- Resilient to obfuscated JavaScript;
- Does not enforce the usage of any sort of SDK or libraries;
- Does not require any dataset for training;
- The detection of malicious JavaScript happens in real-time while the web page is being rendered and displayed;
- Does not depend on any heuristic rule sets;
- Builds upon the most complete JavaScript operational semantics in the literature, proposed by Maffeis *et al.* in [4].

The rest of the paper is structured as follows. Related work is discussed in Section II. The proposed approach is presented in Section III. The operational semantics of JavaScript is presented in Section IV. A case study is presented in Section V. We conclude and discuss future work in Section VI.

II. Related Work

In the literature, there are different proposed methods for detecting malicious JavaScript. These methods can be grouped into five main categories.

We review in this section representative works under each category.

A. Restricting JavaScript to a Safer Subset

The first group of related works is concerned with statically analyzing JavaScript code to either infer or prevent malicious intent. The proposed approaches concentrate on what is known as *language-based* sandboxing. Language-based sandboxing provides a framework to isolate the untrusted JavaScript from the main content of the web page they are embedded in. BrowserShield [5], FBJS from Facebook [6], Caja from Google [7], and ADsafe which is widely used by Yahoo [8], are well known implementations of such approach.

Guha *et al.* in [9] introduced λ_{JS} a small-step operational semantics of the JavaScript language excluding "eval()" construct. The idea was to reduce JavaScript language to a core calculus that models the JavaScript language and is simple enough so that it can be used to define a set of security properties. The authors of [9] proposed a type system to check the security properties on the core calculus λ_{JS}. In [10], the same authors extended the language-based sandboxing work done in [9] to verify browser extensions.

Building on the work done in [9], Poltiz *et al.* in [8] implemented a type-based verification system for sandboxed JavaScript to protect the main content of the web page from advertisement widgets downloaded from third-party servers.

In [11], Finifter *et al.* discovered a vulnerability in ADsafe that could allow leaking some of the capabilities of the main page to the third-parties advertisements. Finifter *et al.* proposed an improved statically verified subset of JavaScript that does not contain such limitation.

In [12], Taly *et al.* developed a tool to verify the soundness of a restricted version of JavaScript that cannot circumvent or subvert a given API. They applied their tool on ADsafe (widely used by Yahoo) and found a vulnerability that allows the third-party code to break the sandbox and access restricted content.

Although language sandboxing approach is effective when it comes to widgets and mashup web pages, it is completely ineffective if the main page is hosting itself the malicious code. In addition, for such approach to work, third-parties have to be forced to use Software Development Kits (SDKs) from the vendors of the sandboxes.

B. Malicious JavaScript Detection Using Machine Learning Techniques

The second group of related works concentrates on analyzing the contents of a web page for extracting specific set of features that might indicate that the web page is malicious. In [13], Cova *et al.* proposed JSAND, a system that relies on dynamic analysis and anomaly detection. JSAND relies on a set of ten features extracted from the web pages. An example feature extracted by JSAND: *Ratio of string definitions and string uses* which measures the number of invocations of JavaScript functions that can be used to define new strings and the number of string uses such as $document.write()$ and $eval()$.

regardless of the relevancy of the above feature, attackers could easily get away by not useing the $eval()$ function. As

the authors point out, most of the features are geared towards obfuscation. The problem is that obfuscation is not only used in malicious web pages, many benign web pages use it as well to protect their intellectual property. Thus Obfuscated JavaScript by itself could be a source of false positive.

Likarish *et al.* in [14] proposed a technique similar to JSAND that extracts features from the web page and uses multiple classifiers to detect malicious JavaScript. The approach suffers from the same problem as JSAND, which is the dependency on specific features in malicious JavaScript that can be simply avoided by the attackers or can be found in benign scripts which could be a source of false positive.

In [15], Curtsinger *et al.* developed a tool named Zozzle for in-browser detection of malicious JavaScript. Although the work done by Curtsinger *et al.* is an in-browser solution capable of detecting malicious JavaScript in real-time, it does not consider browser agnostic attacks (e.g. XSS) that utilize JavaScript and other web technologies.

C. Monitoring the Traffic Using Web Proxy

In [16], Ismail *et al.* proposed an approach to detect XSS attacks by implementing a proxy that analyzes the HTTP traffic exchanged between the client (web browser) and the web application. Their approach suffers from two main limitations. Firstly, it only detects what is known as reflected XSS. Second, The proxy could become a performance bottleneck as it has to parse all the requests and responses going back and forth between the client and the server.

In [17], Kirda *et al.* proposed a web proxy named "Noxes" that analyzes browsed web pages for dynamically generated links and compares those links with a set of filtering rules for allowing or disallowing such links. They use a set of heuristics to generate automatic filter rules for suspicious links and then depend on user intervention to allow or disallow such connection. If the heuristics used are specific for each connection, it would require excessive user intervention which can affect user browsing experience.

D. Honeynets Based Approach

In [18], [19], [20], [21], virtual machine (VM) based honeypots have been proposed to detect web pages with malicious JavaScript. Typically the VM-based approach decides if the visited web page is malicious or not by looking for abnormalities and adding the URL of the page to a blacklist. the problem is that configuring and running several VMs with different exploitable software could be time consuming and expensive.In addition, Clearly, if the blacklist is not updated regularly, the end-user could end up visiting one of the malicious websites.

E. De-obfuscating JavaScript

Several semi-automated tools have been proposed to de-obfuscate JavaScript such as Malzilla in [22] and in [23]. These tools help security experts to reverse engineer and de-obfuscate malicious JavaScript. The main issue with these tools other than they require human intervention, is that they are usually used postmortem after a web page has been identified as malicious already. They are more forensics rather than detection tools.

Blanc *et al.* in [24] proposed an approach to use term-rewriting to de-obfuscate JavaScript code. Although the authors mentioned that the DOM and native JavaScript API should be considered, they did not mention how exactly this will happen. In addition, the authors did not mention how the theorem prover will deal with dynamically loaded JavaScript code (e.g. as a result of Ajax call).

F. Discussions

Although there are many work done in the literature on the detection of malicious web pages, they suffer from one or more of the following issues:

- **Obfuscated JavaScript**: JavaScript embedded in web pages is served in text form (anybody could see the source code and copy it), the authors of the JavaScript code tend to obfuscate their script to make it hard to be copied. In other words, obfuscated JavaScript does not always mean malicious web page.
- **Language Sandboxing**: as mentioned previously, the problem with this approach is the usage of the restricted subset of JavaScript which, usually is shipped as a library or SDK with specific set of APIs that has to be enforced in one way or another. In other words, if an attacker find a way to inject his code in the main page or to bypass the enforced usage of the restricted library, the visitors of such website will end up being exposed to malicious scripts.
- **Proper Dataset**: the machine learning-based techniques mentioned previously need adequate datasets for training. Obtaining the proper dataset that genuinely represents web characteristics and usage patterns has proven to be quite challenging.
- **Real-time Protection**: with the exception of the tool Zozzle and the web proxies mentioned above, none of the proposed solutions can detect in real-time if the currently visited web page is malicious or not.
- **Usage of Heuristic Rule Sets**: the process of finding the balance between a specific rule set and a general one proved to be hard. Too specific rules could generate a lot of false positives and too general rules would be a source of false negative. Furthermore, using heuristic rule sets with dynamic web content is challenging due to the evolving nature of modern web applications.

We present in this paper a new approach that address the above challenges using dynamic information flow control.

III. Proposed Approach

In this section, we present in detail our proposed information flow model and corresponding modifications to the

JavaScript operational semantics that enable the enforcement of information flow control.

A. Information Flow Control Model

Based on the work of Denning [25], we define our flow model FM as a tuple as follows:

$$FM = \langle E, T, S, \oplus, \leq, \rightarrow \rangle$$

where:

- $E = \{e_1, e_2, ...\}$: a set of data elements.
- $T = \{t_1, t_2, ...\}$: a set of tasks in the system.
- $S = \{\underline{e_1}, \underline{e_2}, ...\}$: a set of security classes corresponding to the data elements.
- $\oplus$: a security class combining-operator.
- $\leq$: a partial order relation between security classes.
- $\rightarrow$: information flow between a pair of data elements.

Data elements are the information stores that need protection. Data elements can be obtained from and written to logical objects, such as program variables, objects properties, files, etc. There are three types of logical objects: input-channel, output-channel, and auxiliary objects. The difference between auxiliary objects and channels is that, an auxiliary object can be seen as a logical storage that supports both read and write, while channels can only support either read or write.

Tasks are processing units responsible for data elements computation and manipulation. Security classes correspond to the security levels associated with the information carried by data elements. Each data element e_i is bound to a security class denoted $\underline{e_i}$. Binding of data elements to the security classes is done dynamically. If the content of the data element changes, the security class of the data element has to change to reflect the security class of the new value.

In our proposed information flow model, security classes are assigned to logical objects statically before any information flow happens in the system. For example, we could assign security class "confidential" to a file object that corresponds to a file on the file system; subsequently every data element that is obtained from this file object will be initially bounded to the security class "confidential". Similarly, any data element obtained from any input-channel will be initially bounded to the security class of that input-channel.

The class combining operator "$\oplus$" returns the security class resulting from the application of a function f on a collection of input data elements. If the function takes n arguments $f(e_1, e_2, ...e_n)$ the combined security class will be evaluated as $\underline{e_1} \oplus \underline{e_2} \oplus ... \oplus \underline{e_n}$.

To show that the triple $\langle S, \oplus, \leq \rangle$ is a universally bounded lattice, we denote by $\top$ and $\bot$ the least upper bound and greatest lower bound of our lattice, respectively. $\bot$ corresponds to a security class that puts no restriction on the associated data elements. $\top$ corresponds to the most restrictive security class. To show that $\langle S, \oplus, \leq \rangle$ forms a bounded lattice structure, we must establish that:

1) $\langle S, \leq \rangle$ is a partially ordered set.
2) S is a finite set.
3) S has a lower bound $\bot$ such that any information flow $\bot \rightarrow A$ is allowed for all $A \in S$.
4) $\oplus$ is a least upper bound operator on S.

For assumption 1, we assert that $\langle S, \leq \rangle$ is a partially ordered set given to the model. Each element in S is a security class literal. Assumption 2, is a property of any practical system. The number of security classes in S cannot be infinite. Assumption 3, the existence of a lower bound $\bot$ in S is assumed even if no data element is bounded to such security class in the system. For assumption 4, the class-combining operator $\oplus$ is also a least upper bound operator, as demonstrated as follows:

- $A \leq A \oplus B$ and $B \leq A \oplus B$
- $A \leq C$ and $B \leq C \Rightarrow A \oplus B \leq C$

The realization of the above two properties is achieved by using a **Max()** function that returns the maximum security class of its arguments. So $A \oplus B = Max(A, B)$ for all $A, B \in S$. The assumptions 1-4 imply the existence of a greatest lower bound operator on the security classes, denoted "$\otimes$" such that $A \otimes B = Min(A, B)$ for all $A, B \in S$.

In traditional information flow model, like in [25], the information flow between a source e_1 and a sink e_2, $e_1 \rightarrow e_2$ is allowed if and only if $\underline{e_1} \leq \underline{e_2}$. The nature of client-side web architecture allows a more flexible definition of the notion of safe information flow. Given two data elements e_i and e_j, we allow unrestricted information flow $e_i \rightarrow e_j$ as long as e_j is not an output-channel. If e_j is an output-channel, the information flow $e_i \rightarrow e_j$ is allowed if and only if $\underline{e_i} \leq \underline{e_j}$.

In other words, the security requirement of the system can simply be stated as: a flow model is secure if it does not leak information to unintended output-channel. The system allows any information to flow into the system from input-channels or logical objects (both are considered as data sources). The data elements that represent the data that entered the system are labeled with the security classes bound to such data sources. The system allows information flow between data elements as long as the receiving data element is not an output-channel. If the receiving data element in the information flow is an output-channel, the flow will be allowed if and only if its security class is greater than or equal to the security class of the source element in the flow.

B. Enforcement Mechanism

In order to enforce the security requirement of the proposed information flow model, the semantics of the programming language used to implement the data processing tasks

must be modified to include and propagate the security class information along with the data elements being manipulated by these tasks. The enforcement can be done statically at compile-time or dynamically at run-time. As outlined by Denning in [25], dynamic checking at run-time cannot detect implicit flows. Our proposed model will not suffer such limitation as the security policy will be enforced when information flow happens between a source data element and an output-channel, which cannot happen implicitly.

Since the enforcement of the information flow policy has to be an integral part of the semantics of the programming language used to manipulate the data elements. We chose to modify the operational semantics of the JavaScript language as it is the only language used on the client-side of the web. Although there have been some studies on the operational semantics of JavaScript [26], [27], [28], to our knowledge, the work of Maffeis *et al.* in [4] is the only work that targeted the full core language on the scale defined by the informal ECMAScript specifications [29]. We focus on the most recent version of their work. We adapted the original version of the semantics to allow the assignment and propagation of security classes to the data elements to enable the control of information flow. In comparison to the work proposed in [26], [27], [28], the work of Maffies *et al.* does cover most of the standard and properly formalize the scope object and the prototype chain of the JavaScript language.

$t^{\sim}$		$t_1, ..., t_n$
$t*$		$t_1 ... t_n$ % t+ in the nonempty case
$[t]$		t % t is optional, in case of ambiguity % the [symbol is escaped by "["
$t \mid s$		% t or s
$\&[internal\ Value]$		% Internal values are prefixed % with & such as &NaN
H	::=	$(l : o)^{\sim}$ % heap
l	::=	$\#x$ % object addresses
x	::=	$foo \mid bar \mid \ldots$ % identifiers (do not include reserved words)
o	::=	" {"[$(i : ov)^{\sim}$]"} " % objects
i	::=	$m \mid @x$ % indexes
ov	::=	va[" {"$a^{\sim}$"} "] % object values $\mid fun$"("[$x^{\sim}$]") {"P"} " % function
a	::=	$ReadOnly \mid DontEnum \mid DontDelete$ % attributes
pv	::=	$m \mid n \mid b \mid null \mid \&undefined$ % primitive values
m	::=	"foo" | "bar" | ... % strings
n	::=	$-n \mid \&NaN \mid \&Infinity$ $\mid 0 \mid 1 \mid \ldots$ % numbers
b	::=	$true \mid false$ % booleans
va	::=	$pv \mid l$ % pure values
r	::=	ln" $*$"m % references
ln	::=	$l \mid null$ % nullable addresses
v	::=	$va \mid r$ % values
w	::=	$< va >$ % exception

Table I
SYNTACTICAL CONVENTIONS, META-VARIABLES AND SYNTAX FOR VALUES.

IV. JAVASCRIPT OPERATIONAL SEMANTICS

In this section we summarize the meta-notation used in defining the operational semantics of the JavaScript language as proposed in [4], and then introduce the extended semantics corresponding to our information flow control model.

A. Syntactic Conventions and Meta-notation

The operational semantics proposed by Maffies *et al.* in [4] uses the meta-variables and syntactical conventions outlined in Table I. The values outlined in table I are standard values and closely reflect the values in the JavaScript language. All the objects defined in the JavaScript programs are allocated in the heap memory, as such the proposed operational semantics define a set of functions that deal with the heap. In JavaScript, objects are records of values or functions indexed by strings or internal identifiers. As shown in table I, indexes of objects "i" could be a string "m" or an identifier preceded by "@" sign to distinguish them from user definable properties. Table II outlines heap and object manipulation functions.

Maffies *et al.* define JavaScript semantics in terms of three semantic relations (functions) for expressions, statements, and programs, denoted by $\xrightarrow{e}$, $\xrightarrow{s}$, $\xrightarrow{P}$, respectively. The semantic relations are recursive and mutually dependent. The semantics of programs depend on the semantics of statements, and the semantics of statements depend on the semantics of expressions. Each type of semantic function transforms a triplet of heap-scope-term into a new heap-scope-term triplet. The scope is a pointer in the heap to the current scope object and the term is the currently evaluated term. The result of execution of an expression is a value (pv) or an exception (w). The result of execution of statements and programs are completions. A completion (co) is the final result of evaluating a statement. Table III outlines the structure of a completion. A completion consists of a completion type (ct), a pure value (va), and an identifier (x) triplet. The pure value and identifier could be empty depending on the context, as such in the grammar outlined in Table III both "va" and "x" are suffixed by "e". It is a convention used throughout the proposed semantics. Another use of such convention is with exceptions. For example, if the result of an expression is either a pure value "pv" or an exception "w" it is denoted by "pvw".

B. Proposed Operational Semantics

In order to dynamically enforce the information flow control policy, we must define its operational semantics. The operational semantics is defined as rules that are applied when a specific JavaScript statement is evaluated. The general form of an operational semantic rule is as follows:

$$\frac{computation}{\langle current\ state \rangle \xrightarrow{e \mid s \mid P} \langle end\ state \rangle}$$

$alloc(H, o)$	$= H1, l$	% allocates object o in H yielding % a new address l for o in $H1$.
$H\,(l)$	$= o$	% retrieves o from the heap at l.
$o.i$	$= va$	% retrieves value of property i of o.
$o : i$	$= \{[a^{\sim}]\}$	% possibly empty set of attributes of property i of o.
$o - i$	$= fun"("[x^{\sim}]")\,\{"P"\}"$	% retrieves function % stored in property i of o.
$H\,(l.i = ov)$	$= H1$	% sets the property i of l in H to object value ov.
$del\,(H, l, i)$	$= H1$	% deletes property i of l in H yielding $H1$.
$i\ !< o$		% holds if object o does not contain property i.
$i < o$		% holds if object o contains property i.

Table II
HEAP AND OBJECT MANIPULATION FUNCTIONS.

co	::=	$"("ct, vae, xe")"$
ct	::=	$Normal \mid Break \mid Contine \mid Return \mid Throw$
vae	::=	$\&empty \mid va$
xe	::=	$\&empty \mid x$

Table III
GRAMMAR OF A COMPLETION.

The semantic rules are read bottom to top, left to right. Given a JavaScript expression e, it is pattern-matched to an expression evaluation semantic rule. Then the rule is applied performing the attached computation and transition to the end state happens. We start by the semantics of expressions since they are the basic building blocks of the operational semantics proposed by Maffies *et al.* in [4]. Table IV outlines the syntax for expressions. We use "$\Diamond_{bin}$" to denote binary operator, "$\Diamond_{un}$" to denote unary operator and "$\Diamond_{po}$" to denote postfix operator.

Statements semantic rules follow the same convention of expressions with the exception that the result of a statement is a completion "$co ::= (ct, vae, xe)$" as explained earlier, the grammar of the completion is outlined in Table III. There are two types of statements; internal statements that represent internal execution steps that are not directly defined by the user, and user statements that are part of the JavaScript language itself. User statements are outlined in Table VI. The completion type (ct) determines whether the flow of execution should continue or be interrupted. The completion value (vae) will be relevant if the completion type is $ct = Normal \mid Return \mid Throw$. If $ct = Throw$ the completion value will contain the exception to be thrown. If $ct = Return$, the completion value will contain the value to return. If $ct = Normal$, like in the execution of a function body, the completion value will be propagated to the next statement or expression. Otherwise, the completion identifier (xe) is relevant and it is pointing to where the execution flow show be diverted. This is the case when $ct = Break \mid Continue$.

Based on the operational semantics proposed by Maffies *et*

$e ::=$		
	$this$	% the "this" object
	x	% identifier
	pv	% primitive value
	$"["[e^{\sim}]"]"$	% array literal
	$"\{"[(pn : e)^{\sim}]"\}"$	% object literal
	$"("e")"$	% parenthesis expression
	$e.x$	% property accessor
	$e"["e"]"$	% member selector
	$new\ e["("[e^{\sim}]")"]$	% constructor invocation
	$e["("[e^{\sim}]")"]$	% function invocation
	$function[x]"("[x^{\sim}]")\,\{"[P]"\}"$	% [named] function expr
	$e \Diamond_{po}$	% postfix operator
	$\Diamond_{un}\ e$	% unary operators
	$e \Diamond_{bin} e$	% binary operators
	$"("e"\ ?\ "e"\ :\ "e")"$	% conditional expression
	(e, e)	% sequential expression
$pn ::=$	$n \mid m \mid x$	% property name

Table IV
SYNTAX OF EXPRESSIONS

T	::=	$Undefined \mid Null \mid Boolean$ % primitive types
		$\mid String \mid Number$ % primitive types
		$\mid Object \mid Reference$ % non-primitive types

Table V
JAVASCRIPT INTERNAL TYPES.

$s ::=$		
	$"\{"s*"\}"$	% block
	$var\ x["="e](x["="e])^*$	% assignment
	;	% skip
	e	% expression not starting with "$function$"
	$if"("e")"\ s\ [else\ s]$	% conditional
	$while"("e")"\ s$	% while
	$do\ s\ while"("e")";$	% do-while
	$for"("e\ in\ e")"s$	% for-in
	$for"("var\ x["="e]\ in\ e")"\ s$	% for-var-in
	$continue\ [x];$	% continue
	$break\ [x];$	% break
	$return\ [e];$	% return
	$with"("e")"\ s$	% with
	$id : s$	% label
	$throw\ e;$	% throw
	$try"\{"s*"\}"$ $[catch"("x")"\{"s1*"\}"]$ $[finally"\{"s2*"\}"]$	% try-catch-finally

Table VI
SYNTAX OF JAVASCRIPT STATEMENTS.

P	::=	$s\,[P] \mid fd\,[P]$
fd	::=	$function\ x"("[x^{\sim}]")\,\{"[P]"\}"$

Table VII
GRAMMAR OF A PROGRAM.

Security Class $\underline{e_i}$	::=	instance of security class defined in set $S = \{\underline{e_1}, \underline{e_2}, ...\}$
Data element e_i	::=	$\langle va_i, \underline{e_i} \rangle$

Table VIII
MODIFICATIONS TO THE JAVASCRIPT OPERATIONAL SEMANTICS TO ENABLE DYNAMIC INFORMATION FLOW.

al., programs are defined as sequences of statements and/or function declarations as outlined in Table VII. The result of evaluating a program is a completion (co) as mentioned earlier. In case of the initial parsing of a JavaScript program, variables declarations are added to the "$NativeEnv$" first; followed by functions declarations. A function declaration is equivalent to a no-op statement. If a statement in a program P evaluates to a "$Break \mid Continue$" outside a control structure, a syntax error exception is thrown.

The "$NativeEnv$" is the initial heap that contains the native objects that implement the functions, constructors, and prototypes. It also contains the initial scope object and the global object represented by "$@Global$". The global object is the root of the scope chain, as such its "$@Scope$" property is set to null and the "$@this$" property points to itself "$\#Global$".

C. Modified Operational Semantics

The modifications needed to the operational semantics to enable dynamic information flow are outlined in Table VIII where va_i is the pure value (va) outlined in Table I and $\underline{e_i}$ is the security class bounded to e_i which could be an instance of any security class defined in set $S = \{\underline{e_1}, \underline{e_2}, ...\}$. In other words, we associate a security class ($\underline{e_i}$) with every pure value (va_i) that corresponds to a data element e_i in the information flow model. Table IX illustrates some modified operational semantics with dynamic information flow. The `in-operator` semantic rule illustrates how the security class "$\underline{m}$" of the property "m" is transferred to the resulted boolean. The two `logic-operator` examples emphasize that the result of the logic operation is one of the operands rather than a boolean, as such the security class of the result will be the associated security class with the returned operand "$\underline{va}$" or the result of the expression "$\underline{e}$". The result of the `strict-equality` expression is a boolean "b" with a security class "$\underline{b}$" equals to the result of applying the security class combining-operator "$\oplus$" on the security classes of operands "$va1$" and "$va2$". When it comes to the statement and program semantic relations $\xrightarrow{s}$ and $\xrightarrow{P}$, the pure value "va" part of the completion "$co = (ct, < vae, \underline{vae} >, xe)$" will carry the associated security class in similar way to the expression semantics illustrated in Table IX.

$$\frac{\text{Type(l1)=Object} \quad \text{H,l1.@HasProperty(<m,}\underline{\text{m}}\text{>)=<b,}\underline{\textbf{b=m}}\text{>}}{H, l, < m, \underline{m} > \ \texttt{in}\ l1 \xrightarrow{e} H, l, < b, \underline{\textbf{b}} >}\ \texttt{in-op}$$

$$\frac{b1\ \texttt{xor}\ b2}{H, I, @L\,(b1, b2, < va, \underline{va} >, < e, \underline{e} >) \xrightarrow{e} H, l, < va, \underline{\textbf{va}} >}\ \texttt{logic-op}$$

$$\frac{!\,(b1\ \texttt{xor}\ b2)}{H, I, @L\,(b1, b2, < va, \underline{va} >, < e, \underline{e} >) \xrightarrow{e} H, l, < @GV\,(e)\,, \underline{\textbf{e}} >}\ \texttt{logic-op}$$

$$\frac{\text{Type(va1) = Type(va2)} \quad (\text{<va1,}\underline{\text{va1}}\text{> == <va2,}\underline{\text{va2}}\text{>}) = \text{<b,}\underline{\textbf{b= va1}} \oplus \underline{\textbf{va2}}\ \text{>}}{H, l, va1\ ===\ va2 \xrightarrow{e} H, l, < b, \underline{\textbf{b}} >}\ \texttt{strict-equality}$$

Table IX
EXAMPLE OPERATIONAL SEMANTICS WITH DYNAMIC INFORMATION FLOW MODIFICATIONS.

V. CASE STUDY

A. Case Description

Major service providers like Facebook, Microsoft, and Google enable their users to use what is known by single sign-on (SSO) to access their accounts while visiting other websites. For example, a web user could be reading a blog post on a blogging website and using the SSO, she could share such blog post through her Facebook account without leaving the blogging website. This is enabled by OAuth 2.0, a web resource authorization protocol that is employed by major service providers. In the terminology of the OAuth standard the service provider is referred to as Identity Provider (IdP) (e.g. Facebook) and the third-party website that is running the web application that uses the web user identity is referred to as relying party (RP). The RP relies on the OAuth protocol to authenticate the web user to the IdP and using the web user account information, the RP customizes the user experience on their website. Sun *et al.* in [30] conducted an experiment to evaluate the implementations of three major OAuth IdP, namely Facebook, Microsoft, and Google, and 96 popular RP websites that support the use of Facebook accounts for login. They discovered several vulnerabilities that would allow an attacker to gain unauthorized access to the victim user's profile, and impersonate the victim on the RP website. Listing 1 depicts an example malicious script that is used by Sun *et al.* in [30] to steal the access token of a web user. We use this malicious JavaScript code as a real-world example of leaking sensitive information to illegal channel. In the next section we show how our model detects such leakage and prevent it from happening.

B. Information Flow Model for Secure Web Browsing

As stated above, our flow model is defined as a tuple as follows:

$$FM = \langle E, T, S, \oplus, \leq, \rightarrow \rangle$$

In the context of a web browser the elements of FM could be defined as follows:

- $E = \{e_1, e_2, ...\}$: is the set of elements of the HTML page being rendered. The source of the HTML page elements is an input-channel from a particular domain. Based on the motivating example mentioned above the domain is "benignSite.com". As described in the proposed model, all the data elements obtained and/or derived from the "benignSite.com" will initially be bounded to the security class that corresponds to such domain, say "benignSite".

- $T = \{t_1, t_2, ...\}$: is the set of active tasks that manipulate the data elements. In a web browser that could be layout-task, rendering-task, parsing-task, etc.

- $S = \{\underline{e_1}, \underline{e_2}, ...\}$: is the set of security classes. In a web browser the security classes could be defined as $S = \{\texttt{unclassified}, \texttt{domain}, \texttt{confidential}\}$. The security classes have the following relation between them $\{\texttt{unclassified} < \texttt{domain} < \texttt{confidential}\}$. In this case, the `confidential` security class is the least-upper-bound (H = `confidential`) and the `unclassified` security class is the greatest-lower-bound (L = `unclassified`). The three classes form a lattice structure `unclassified` $\rightarrow$ `domain` $\rightarrow$ `confidential`. An important aspect of the proposed model is the initial assignment of the security classes to the logical objects and channels. In the context of web browser, the initial class assignment is intuitive. When the browser communicates with a website, two channels are created, input-channel and output-channel. Both channels are assigned security class "**domain**" that corresponds to the website's domain. Any kind of data that is obtained from such domain will also be labeled with the same security class, e.g. web cookie or web storage. The intuition behind that is any website should be able to access its own data stored at the client-side. Any data that is private to the browser such as bookmarks and browsing history are labeled as "**confidential**". Other kind of non-sensitive data, such as browser type and version could be labeled as "**unclassified**".

- $\oplus, \leq, \rightarrow$: The class combining-operator $\oplus$ will combine the security classes of two data elements if there is a flow between them and the second element is not an output-channel e.g. $e_1 \rightarrow e_2$ = `unclassified` $\oplus$ `domain` = `Max(unclassified,domain)` = `domain`. If a flow is happening between a data element and an output-channel, the secure flow policy is enforced and the flow is only allowed if and only if the security class of the output-channel is greater-than or equal to the security class of the data element, $e_1 \rightarrow e_2$ `iff` $\underline{e_1} \leq \underline{e_2}$ and e_2 is an output-channel.

Based on the motivating example illustrated in Listing 1, lines number 23 up to 31 create a hidden "*iframe*" element that communicates with the "*__AUTHZ_ENDPOINT__*" or the identity provider (e.g. Facebook) passing the RP web application ID and a redirection URL that points back to the RP's web application. When executed, a forged authorization request to the IdP will be created and an access token will be obtained in return and the web user will be directed back to the RP's web application. When the access token is obtained the code listed in lines number 14 to 21 will extract the access token from the URL and will call the "*harvest()*" function. The "*harvest()*" function in turn creates an "**image**" element pointing its source to the evil website "*__HARVEST_URL__*" and passes the stolen access token along. This is illustrated in lines number 2 to 8. Per our definition of information flow model and extended JavaScript operational semantics, the *url* variable in line 16 is a data element with string value representing the URL of the RP's web application and security class `domain = __RP_HOSTNAME__`. On line number 17 the *token* variable is created by splitting the *url* variable and extracting the access token part. The *token* data element will inherit the same security class `domain = __RP_HOSTNAME__` of the *url* data element as the result of the assignment statement. The *token* variable is then passed to the *harvest()* function as an argument. The result of the addition operation in line number 3 between the function parameter *access_token* and the string constant "*__HARVEST_URL__?access_token=*" is a concatenation of the value "*__HARVEST_URL__?access_token=*" of type string and the value of the *access_token* parameter. Table X illustrates the operational semantics of the addition operation between the *access_token* parameter and the string constant "*__HARVEST_URL__?access_token=*". The result of the addition operation is a concatenation of the pure value "va" of type string and the result of applying to-string "$@TS$" function on the function parameter that contains the access token.

The security class is elevated to `domain = __RP_HOSTNAME__` based on the class combining-operator and is assigned to the *src* variable:

$$\texttt{"unclassified"} \oplus \texttt{"__RP_HOSTNAME__"} \Rightarrow$$

$$\texttt{Max("unclassified","__RP_HOSTNAME__")} \Rightarrow$$

$$\texttt{"__RP_HOSTNAME__"}.$$

On line number 6 the *src* is assigned to the source of the image element. When line number 7 is about to execute, an output-channel will be created with security class `domain = __HARVEST_URL__` since the source property of the image element points to "*__HARVEST_URL__*". As the resulted concatenation from line 3 is about to flow to the output-channel, the secure flow requirement has to be enforced. Since the security class of output-channel `domain = __HARVEST_URL__` is $\not\leq$ to `domain = __RP_HOSTNAME__`, such flow will be prevented and the stolen access token will not be sent to the "*__HARVEST_URL__*".

```
1  //send access token via img element
2  function harvest(access_token) {
3  var src="__HARVEST_URL__?access_token=" +
       access_token
4  var d = document; var img, id = "harvest";
5  img = d.createElement("img"); img.id = id; img.
       async = true; img.style.display="none";
6  img.src = src;
7  d.getElementsByTagName("body")[0].appendChild(img)
       ;
8  }
9
10 (function(d){
11   var rp_host_name="__RP_HOSTNAME__";
12   var rp_app_id="__RP_APPID__";
13   // begin: this page is inside an iframe
14   if(top!=self) {
15       if(d.location.hash != "" ) {
16       var url=d.location.href;
17       var token = url.split("access_token=")[1];
18       token=token.substring(0, token.indexOf("&"))
             ;
19       harvest(token); }
20   return; // end: this page is inside an iframe
21   }
22   // begin: this page is not inside an iframe
23   var redirect_uri= d.location.href;
24   var iframe_src="__AUTHZ_ENDPOINT__?client_id="+
       rp_app_id+"&redirect_uri="+redirect_uri+"&
       response_type=token";
25
26   var f, id = "iframe-hack";
27   if (d.getElementById(id)) {return;}
28   f = d.createElement("iframe"); f.id = id;
29   f.async = true; f.style.display="none";
30   f.src = iframe_src;
31   d.getElementsByTagName("body")[0].appendChild(f)
       ;
32 }(document));
```

Listing 1. Information Flow Example.

VI. Conclusions and Future Work

Modern web applications rely mainly on JavaScript to provide functionality on the client-side. At the same time, JavaScript is used by attackers to launch attacks against the visitors of such web applications. The dynamic nature of the JavaScript language and its tangled usage with other web technologies makes it hard to reason about its code statically. In this research, we proposed an information flow control model that tracks the information flow and prevents the communication of sensitive information to illegal channels dynamically at run-time. In order to formalize such model in a web browser setting, we extended the JavaScript operational semantics proposed by Maffies *et al.* in [4] to enable information flow control. We used a real world example of vulnerability in the OAuth protocol to demonstrate how the proposed model along with the modified operational semantics could prevent access token leakage from happening. We are in the process of implementing the information flow control model in an open-source web browser. In future work, we will conduct a large-scale experimental evaluation where we will test the implementation of the model against websites that contain malicious JavaScript as well as benign websites from Alexa top-100 list[1].

References

[1] Wikipedia.com, "Samy computer worm," 2013. [Online]. Available: http://en.wikipedia.org/wiki/Samy_(computer_worm)

[2] Wikipedia, "Mikeyy computer worm," 2013. [Online]. Available: http://en.wikipedia.org/wiki/Mikeyy

[3] 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 *In Proceeding of the Network and Distributed System Security Symposium (NDSS'07*, 2007.

[4] S. Maffeis, J. C. Mitchell, and A. Taly, "An operational semantics for javascript," in *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, ser. APLAS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 307–325. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89330-1_22

[5] C. Reis, J. Dunagan, H. Wang, and O. Dubrovsky, "BrowserShield: Vulnerability-driven filtering of dynamic HTML," *ACM Transactions on the Web (TWEB)*, vol. 1, no. 3, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1281481

[6] Facebook, "Javascript sdk," 2013. [Online]. Available: https://developers.facebook.com/docs/reference/javascript/

[7] Google, "Google caja," 2013. [Online]. Available: https://developers.google.com/caja/

[8] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi, "ADsafety: Type-Based Verification of JavaScript Sandboxing," in *SEC'11 Proceedings of the 20th USENIX conference on Security*, 2011.

[9] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of JavaScript," in *ECOOP 2010–Object-Oriented ...*, 2010, pp. 1–25. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-14107-2_7

[10] A. Guha, M. Fredrikson, and B. Livshits, "Verified security for browser extensions," *SP '11 Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pp. 115–130, May.

[1] **http://www.alexa.com/topsite**

$$\frac{\begin{array}{c}\text{Type(va) = String}\\ \underline{\text{va1}} = \text{"unclassified"}\\ \underline{\text{access\_token}} = \text{"\_\_RP\_HOSTNAME\_\_"}\end{array}}{H, l, va \ + \ access\_token \xrightarrow{e} H, l, < m = concat(va, @TS(access\_token)), \underline{m} = \underline{va} \oplus \underline{access\_token} >}$$

Table X
INFORMATION FLOW EVALUATION FOR LINE 3 IN LISTING 1.

[11] M. Finifter, J. Weinberger, and A. Barth, "Preventing Capability Leaks in Secure JavaScript Subsets," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, 2010. [Online]. Available: http://www.cs.berkeley.edu/ finifter/talks/ndss2010.pdf

[12] A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra, "Automated analysis of security-critical javascript apis," in *2011 IEEE Symposium on Security and Privacy*. Ieee, May, pp. 363–378.

[13] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious JavaScript code," in *Proceedings of the 19th international conference on World wide web - WWW '10*. New York, New York, USA: ACM Press, 2010, p. 281. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1772690.1772720

[14] P. Likarish, E. Jung, and I. Jo, "Obfuscated malicious javascript detection using classification techniques," in *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, 2009, pp. 47–54.

[15] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: fast and precise in-browser javascript malware detection," in *Proceedings of the 20th USENIX conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028070

[16] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi, "A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability," in *Proceedings of the 18th International Conference on Advanced Information Networking and Applications - Volume 2*, ser. AINA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 145–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977394.977497

[17] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes : A client-side solution for mitigating cross-site scripting attacks," *Proceedings of the 2006 ACM symposium on Applied computing*, pp. 330–337, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1141357

[18] A. Moshchuk and T. Bragin, "A Crawler-based Study of Spyware on the Web," in *Network and Distributed Systems Security Symposium*, 2006. [Online]. Available: http://cs.uno.edu/ dbilar/11CSCI6621-NetworkSecurity/papers/spycrawler.pdf

[19] A. Moshchuk, T. Bragin, and D. Deville, "Spyproxy: Execution-based detection of malicious web content," in *USENIX Security Symposium*, 2007, pp. 1–16.

[20] Y. Wang, D. Beck, and X. Jiang, "Automated Web Patrol with Strider HoneyMonkeys," *Network and Distributed Systems Security Symposium*, pp. 35–49, 2006.

[21] N. Mavrommatis and M. Monrose, "All your iframes point to us," in *USENIX Security Symposium*, 2008, pp. 1–16.

[22] B. Spasic, "Malzilla: Malware hunting tool," 2013. [Online]. Available: http://malzilla.sourceforge.net

[23] J. Nazario, "Reverse engineering malicious javascript," in *CanSecWest*, 2007.

[24] G. Blanc, R. Ando, and Y. Kadobayashi, "Term-rewriting deobfuscation for static client-side scripting malware detection," in *New Technologies, Mobility and Security (NTMS), 2011 4th IFIP International Conference on*, 2011, pp. 1–6.

[25] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, May 1976. [Online]. Available: http://doi.acm.org/10.1145/360051.360056

[26] D. Yu, A. Chander, N. Islam, and I. Serikov, "Javascript instrumentation for browser security," in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '07. New York, NY, USA: ACM, 2007, pp. 237–249. [Online]. Available: http://doi.acm.org/10.1145/1190216.1190252

[27] C. Anderson, P. Giannini, and S. Drossopoulou, "Towards type inference for javascript," in *IN PROC. 19TH EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING*. Springer, 2005, pp. 429–452.

[28] P. Thiemann, "Towards a type system for analyzing javascript programs," in *Proceedings of the 14th European conference on Programming Languages and Systems*, ser. ESOP'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 408–422. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31987-0_28

[29] E. International, "Ecmascript language specification, standard ecma-262," 2013. [Online]. Available: http://www.ecma-international.org/ecma-262/5.1/

[30] S.-T. Sun and K. Beznosov, "The devil is in the (implementation) details: An empirical analysis of oauth sso systems," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 378–390. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382238