Preventing Client Side XSS with Rewrite Based Dynamic Information Flow

Conference Paper · July 2014

DOI: 10.1109/PAAP.2014.10

Hao Chen
Hunan University

2014 Sixth International Symposium on Parallel Architectures, Algorithms and Programming

Preventing client side XSS with Rewrite Based Dynamic Information Flow

Wenmin Xiao[1], Jianhua Sun[2], Hao Chen[3], Xianghua Xu[4]
[1-3]College of Information Science and Engineering, Hunan University, Changsha, China
[4]Zhejiang Provincial Key Lab of Data Storage and Transmission Technology, Hangzhou Dianzi University
[1]wenminxiao@hnu.edu.cn, [2]jhsun@aimlab.org,[3]haochen@aimlab.org

Abstract—**This paper presents the design and implementation of an information flow tracking framework based on code rewrite to prevent sensitive information leaks in browsers, combining the ideas of taint and information flow analysis. Our system has two main processes. First, it abstracts the semantic of JavaScript code and converts it to a general form of intermediate representation on the basis of JavaScript abstract syntax tree. Second, the abstract intermediate representation is implemented as a special taint engine to analyze tainted information flow. Our approach can ensure fine-grained isolation for both confidentiality and integrity of information. We have implemented a proof-of-concept prototype, named JSTFlow, and have deployed it as a browser proxy to rewrite web applications at runtime. The experiment results show that JSTFlow can guarantee the security of sensitive data and detect XSS attacks with about 3x performance overhead. Because it does not involve any modifications to the target system, our system is readily deployable in practice.**

Keywords-information security; taint model, information flow analysis, cross-site scripting, JavaScript

I. Introduction

In modern web applications, the client side scripts are embedded everywhere, and these scripts provide powerful functions and flexibility, such as form validation, dynamic rendering, interactive UI, and so on. However JavaScript lacks the isolation and protection mechanism at the language level. Web pages and third-party code coexist in a shared environment, and the interaction between them may lead to security issues. Even under the protection of the same origin model in the browser, third-party code has many ways to perform cross-domain requests, such as the HTML tags of IMG, cross IFRAME messaging, etc. This poses great risks of leaking sensitive information in web applications.

XSS is an important security issue in the domain of web in recent years, which enables the attacker to insert malicious code into web pages. When users browse the pages, they are not able to distinguish malicious code from the trusted one, leading to the potential revelation of sensitive data by the embedded malicious code. Existing XSS attacks can be divided into three categories, namely reflective XSS, stored XSS, and DOM XSS. Reflective XSS is that the attacker tricks users to click a link that carry malicious script payloads that will be shown in the linked page. In stored XSS, the attacker spreads malicious code to normal web servers, so users who browse the pages containing malicious code would suffer from XSS attacks. DOM XSS is also known as local XSS attacks, usually included in reflective XSS attacks. The difference is that the injected code is DOM, while reflective XSS injects JavaScript code.

The ideal solution to prevent XSS attacks would be to eliminate the vulnerabilities in the affected web applications. In order to achieve this goal, web applications need to validate user inputs at all possible locations. However, web application developers or service providers may not be able to repair every security hole in a timely way. Hence, deploying a security mechanism on the client side is usually a practically option.

The solution proposed in this paper is rewrite-based dynamic information flow tracking, which combines taint model [9] and information flow analysis [4] ideas, by rewriting the client JavaScript code to ensure the confidentiality and integrity of sensitive data. Confidentiality means that data cannot be read in an unauthorized or untrusted manner. Integrity means that data cannot be modified in an unauthorized or untrusted manner. Unlike traditional taint model that are generally used on the server side with labels to mark data as tainted, we use it on the client side to mark data with multiple labels to achieve different purpose. For instance, the label of 'secret' marked in sensitive data is to ensure the data confidentiality, and the label of 'public' is marked in all data, and the label of 'untrusted' marked in the data from URLs that are not trusted is used to ensure the data integrity. A check is performed for data confidentiality at every read operation and for data integrity at every write operation, where the violation would be recorded and reported to the user.

In this paper, we make the following contributions:

1. We present a rewrite-based dynamic information flow framework.
2. We implement a prototype implementation of our framework and deploy it as a browser proxy.
3. We present extensive evaluation of our framework for data confidentiality and integrity.

The rest of this paper is organized as follows: Section 2 presents some challenges and our approaches. Section 3 introduces taint model and security policy. Section 4 explains how the rewrite framework works. Section 5 presents the evaluation of our prototype. Section 6 presents related work. Finally, Section 7 concludes.

II. Challenges

The ultimate goal of this paper is to enforce information flow security. This section outlines the main challenges and illustrates our approach to resolving them.

EVAL: The runtime possibility to pause and execute a string in a JavaScript program, provided by the *eval* function, poses great challenges for static program analysis, since the string may not be available to the static analysis tool.

Our approach is to rewrite the string parameter of the *eval* function, and the rewrite is like the form of *eval*(code) → *eval*(RW(code)), where the RW parses code into abstract syntax tree (AST), transforms the AST according to specified information flow polices, and generates new security code that will perform dynamic analysis at runtime. This is the main reason for the use of dynamic information flow analysis.

Information integrity: JavaScript is dynamic as it has the ability to load script from remote site and execute it asynchronous, which has important security implications for information integrity, since it is easy to load any code from any place. Furthermore, with the absence of isolation mechanisms in JavaScript language, it is hard to prevent untrusted code from flowing into web pages.

Our approach: We introduce a multiple-taint model to track different sources to enhance information confidentiality and integrity. We use a set of taints to identify data from different sources and a flow policy specification that is configured by specifying a white list of URLs for what sensitive data are allowed to read or write. If a variable from an URL is not in the white list, it should be marked as 'untrusted', and the untrusted variable cannot flow into sensitive data as guaranteed by the taint checking operation.

Indirect and implicit flow: Traditionally, information flow analysis includes explicit, indirect, and implicit flows. Explicit flows amount to directly copy information via an assignment like y = x, where the value of variable x flows into variable y, represented as x → y. Implicit flow are dependent on control flows (branch, loop, procedure) of the program. Consider the code snippet,

```
    y=0; z=0;
    if(x) y=1; else z=1;
```

Depending on the value of variable x, variable y or z will either be set to 1 or remains 0. Hence, there are two indirect flows: from x to y (x → y) and from x to z (x → z). Inversely, we can induce the value of x according to the value of y and z; y=1 implies x is true, and z=1 indicates x is false. This means there is an implicit flow from the combination of (y, z) to x, which would lead to the leaking of the information of x.

Our approach: Combined with static analysis, we employ a checkpoint mechanism that backups current execution environment and restore it in the future using hook functions to enhance flow security. We provide two functions ControlEnter(id, leftValues), and ControlExit(id) as the abstract semantic of backup and restore, where the first parameter *id* is an abstract execution point and it is the same for the pairing functions that can backup and restore taint context to achieve indirect flow tracking, and the second parameter *leftValues* are a set of assigned variables inside all branches in the control structure, which is injected by static analysis. For instance, in the above code snippet, the variables y and z are extracted into the parameter leftValues of the controlEnter function. If x is tainted, all leftValues including y and z will be tainted, so the security of implicit flow is guaranteed.

Aliasing and field sensitivity: Sensitive data is identified by the data location that is a tuple of the name of the object and property, which is used to inject and check taint information. For instance, the tuple ('document', 'cookie') is a data location that denotes the property of the document object. In JavaScript, objects are heap allocated, and can be considered anonymous and represented by referencing their heap locations. Since a reference is a binding from the name to the object, objects may be aliased. An alias occurs when two references refer to the same object. Field sensitivity in the presence of alias poses a significant challenge for static analysis. For instance, for the following code snippet: c1=document.cookie; d=document; c2=d.cookie; where *d* is the alias of the document object, so they have different data locations and would escape the confidentiality check when reading the property 'cookie' of the object *d*. Similar situation exists when writing the property of aliased object.

Our approach: We add an alias field to every object to identify the object name, which are initialized lazily. Consider the above example, first, the document object has no alias field; after the assignment statement d=document, the document object would initialize the alias field with a value of literal 'document'. Because the object alias initialization is executed only once, we can assign the document object to any variable that would denote the same data location. This kind of programs is rejected by static analysis tools like JIF [3].

III. Dynamic data tainting

A. *Taint Model*

Taint Injection: First, we associate with each object a set of taints that describe where the object has come from, where it has went, and what primitives like number and string belongs to object. A confidentiality taint is of the form <CnfTag, URL> that specifies the tainted object is confidential to the code originating from URL. Every sensitive data is injected with this taint. An integrity taint is of the form <ItgTag, URL> that specifies the tainted object influenced by code originating from URL. Every object originating from URL is injected with this taint.

Taint Propagation: Second, we propagate the taints with the objects as they flow through the program via assignments, arithmetic and logic operations, control structures, loops, function calls, and eval etc. A static analysis would carry out this propagation via some form of dataflow analysis [4], while a dynamic analysis [1] would attach taints with the objects and copy them around together with the object.

Taint Checking: Third, at each point where value-flow occurs, such as read operation, assignment of a source object to a target object location, we check if the assignment is legal with respect to the taints associated with the object. From the policy file, a white list of URL for each object-location of sensitive data is specified. The flow is legal if: (a) for each integrity taint <ItgTag, URL> carried by the object, the URL is in the white-list, which means the target data-location is allowed to be influenced by the URL; (b) for each confidentiality taint <CnfTag, URL> carried by the object, the URL is in the white-list, which means the confidential data originate from the URL is allowed to read at the current URL.

B. Sensitive data source

For our system, what data are considered sensitive and how to identify them are two different problems that we should confront first. A data source is considered sensitive when it holds information that could be abused by an adversary to launch attacks or to learn information about a user (e.g.. cookie or history of visited web page). A list of tainted sources used by our system is provided in table Ⅰ. Since this list is provided by Netscape, and we make a little extension to it, by adding *image* and *iframe* object with attribute *src*, as they can asynchronously load resources from third-party sites that may transfer sensitive data to an adversary. Because those sensitive data are configured in a policy file, if additional sensitive data sources are discovered, our system can be easily extend to handle them as well.

Unlike the way of modifying or extending JavaScript engine to mark sensitive data source, we mark it at compile time or runtime, which make a big difference in implementing taint model. From the view inside the JavaScript engine, it is easy to identify what is the *window* object and what is the *document* object. But from the view of high-level code, we have to identify it from the name of strings. However, object aliasing makes it complicated that we cannot statically analyze the field operation (read or write) from object precisely. Therefore, we add an alias field to every object to store the real name of the object and evaluate the object's real name when reading or writing object field at runtime.

C. Information flow polices

In our framework, we provide a simple and fine-grained policy [17] for security information flow, which is specified by sensitive data and a white list of URL that are granted access permission. We formalize the notion of data-location and policy of confidentiality and integrity.

Data-location: Data-location is a tuple like (t1, t2) where t1 is the object name and t2 is the property name, both of them are strings. Every object that is created in the program is stamped with a name. DOM variable name is the node name, such as document variable name is '#document', and image variable name is '#image'. As primitives are not object, so they have no names. Hence, the tuple of ('#document', 'cookie') is the data-location corresponding to the cookie property of the global variable *document,* and the tuple of ('history', 'previous') is data-location corresponding to the previous property of global variable *history*.

Policies: Policies are categorized into two kinds: Cnf and Itg, where Cnf is a confidentiality policy that is a finite map from data-location to URLs that are allowed to read the data-location, and Itg is a integrity policy that is a finite map from data-location to URLs that are allowed to write the data-location. For example,

Cnf =[('document', 'cookie') → ['a.com', 'b.com']],

Itg =[('document', 'location') → ['a.com', 'c.com']],

which means the value of document.cookie should only be read by the code from 'a.com' and 'b.com', and document.location should only be written by the code at 'a.com' and 'c.com'.

TABLE I. SOURCE OF TAINT VALUES

Object	Tainted properties
Document	cookie, domain, forms, lastModified, links, referrer, title, URL
Form	action, src
Form input elements	checked, defaultChecked, defaultValue, name, selectedIndex, toString, value
History	current, next, previous, toString
Select option	defaultSelected, selected, text, value
Location and Link	hash, host, hostname, href, pathname, port, protocol, search, toString
Window	defaultStatus, status
Iframe, Image	src

IV. REWRITE

A. Architecture

The processing of a JavaScript document is shown in Figure 1. The original JavaScript code is first parsed into AST. Then, the AST is transformed into a new AST by instrumentation based on the predefined polices. Finally, the code generator produces traceable code that includes GRE (general representation engine) and taint analysis engine. The GRE is an abstraction of the JavaScript instruction set. For instance, read function is an abstraction of LOAD instruction, and write function is an abstraction of STORE instruction. The taint analysis engine is an implementation of GRE that is intended for the analysis of taint information flow. According to the purpose of the taint engine, it can be categorized into three parts: semantic, control flow, and security.

B. Rewrite of Semantics

The semantic rewrite functions [3] from Table II add a new operational semantic, taint tracking, to ensure the confidentiality and integrity of information while keeping the original operational semantics. For instance, for the expression e1 op e2, in addition to the general binary operation, it also needs to perform taint operations, taint accumulation, and the result is a combination of taints from e1 and e2. So the taint operation will be performed along with each operation in the program, such as reading, writing, function calls, and arithmetic, etc. The taint operations include taint injection, propagation, and checking. Taint injection and check are performed along with read and write operations, and taint propagation is performed along with arithmetic and logical operations.

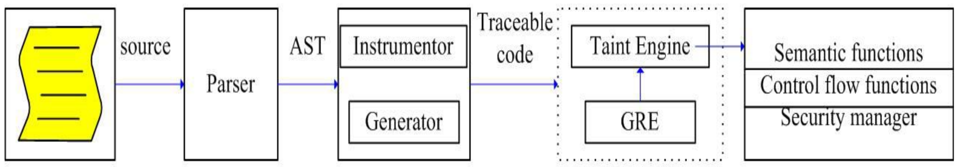

Figure 1. Processing of a JavaScript document

TABLE II. REWRITE SEMANTIC

Syntax	Rewrite	Abbr	Sample
c/constant	literal(c)	L	'abc' → l('abc')
x/variable	read(x)	R	foo → r(x)
x=e/assign	write('x', e)	W	foo=e → w('foo', RW(e))
x.f/field read	getField(object, property)	gf	foo.bar → gf(r(foo),' bar') foo[bar] → gf(r(foo), bar)
x.f=e/field write	putField(object, property, val)	pf	foo.bar = e → pf(r(foo), 'bar', RW(e))
f(e)l	functionCall(func, isCtor)(args)	fc	foo(e) → fc(foo, false)(RW(e)) new foo(e) → fc(foo, true)(RW(e))
x.f(e)	methodCall(object, property, isCtor)(args)	mc	foo.bar(e) → mc(foo, 'bar', false)(RW(e)) new foo.bar(e) →mc(foo, 'bar', true)(RW(e))
e1 op e2	binary(op, left, right)	bi	e1+e2 → bi('+', RW(e1), RW(e2))
op e1	unary(op, operand)	ui	!e → ui('!', RW(e))
with x	withObject(obj)	wo	with(x) → wo(r(x))
for key in obj	enumerate(obj)	em	for(var I in obj) →for(var I in em(obj))

condition expression	Condition(left)	cd	if(e1) → if(cd(e1))
			switch(e1) → switch(cd(RW(x))
			e1? e2: e3 →cd(RW(e1))? RW(e2): RW(e3)

case	caseOf(obj)	ca	case 2 → ca(l(2)):
return	returnOf(obj)	rt	return 2 → rt(l(2))
eval(code)	eval(instrument(code))	RW	eval('1+1') → eval(RW('1+1'))

C. *Rewrite of control flow*

The rewrite functions for the control flow translate the syntax of branch, loop, and procedure etc. Those functions insert a pair of hook functions before and after the flow transfer instruction. For instance, a pair of functions scripteEnter(id, url) and scriptExit(id) is inserted at the top and bottom of a script; functionEnter(id, func) and functionExit(id) are inserted inside a function; controlEnter(id) and controlExit(id) are inserted at the start and end of a branch. The parameter id is the location of the control flow, which is numerically assigned starting at 1 and increases gradually, and the ids of the pairing functions have the same value. These functions provides great flexibility for our system, such as backup and restore status information, statistics of the script loading times and the execution time, monitoring the execution of specific functions, and filtering script from dangerous URLs. In this paper, we use it to backup and restore the global taint context to implement the control of indirect and implicit flow.

D. *Security Management*

Security management is mainly responsible for policy management, taint injection, taint checking, and logging, etc. In order to protect sensitive data, when the sensitive data is under illegal access, the system will record the violation operation, and take the corresponding protective measures such as replacing sensitive data with null value or abort the program.

V. EVALUATION

We have implemented a prototype of the framework that employs rewrite based dynamic information flow analysis to prevent XSS attack, and deploy it as a browser proxy. The entire system is implemented with pure JavaScript, the proxy server uses node.js, and the parser is based on the JavaScript library esprima. The experiments focus on the test of information confidentiality and integrity.

Benchmark: Confidentiality and integrity test, <number>d is a HTML page, which includes multiple scripts named "sensitive.js" that are located in different domains, <number> represents the total number of domains.

Sensitive data mainly include location, cookie, the src attribute of image and iframe, and input, which are accessed in different syntax constructures of JavaScript in the script of "sensitive.js". The script "sensitive.js" contains multiple test cases: *aliases* test, *indirect* flow test, *implicit flow* test, *eval* test, *with* test, *DOM* test. Each domain has a flag denoting whether it is trusted. Trustability is the percentage of trust domains in all domains.

XSS test: There are four XSS test examples that come from webgoat, which is a deliberately insecure J2EE web application designed to teach web application security lessons. 1) DomXSS: an evil page is embedded into the host page named DomXSS using an IFRAME tag, and the src attribute of the IFRAME refers to the source of the evil page "cookieAlert.html", which can access sensitive data cookie and show it in an alert dialog. 2) ReflectXSS: searching employee with a value of JavaScript code, and the result page will show what input values you have searched. 3) StoreXSS: modifying the street information of an employee profile with the value of evil JavaScript code that would access the sensitive data cookie. Next, searching the employee, then an alert that contains information of cookie will be showed before rendering the result page. 4) PhishXSS: searching employee with a crafted code, which inject a login form into the current page and promote this feature needs authentication, once you click the login button, the input information of user name and password will be transferred to anther site.

Table format: The columns in the table are as follows: "bm" is the name of benchmark that test case is a simple HTML page, "dn" is the name of domain number, which is the total number of domains of the loaded external scripts in the page. "sfc" and "afc" represent the name of sensitive flow count and all flow count respectively. "sr" is the name of sensitivity ratio that equals sfc/afc. "vcc" is the name of violation confidentiality counter, and "vic" means violation integrity counter, where violation denotes an illegal access to the sensitive data. "tr" (taint ratio) is equal to (vcc+vic) / sfc, "ori" is the time of executing original code that is not rewrite, "rw" is the time of executing rewrote code, trustability is the ratio of the trusted domain in the proportion of all domains.

The experiments were performed on a machine equipped with 1.86 GHz Intel Core Duo processor and 2G of RAM. The operating system is Ubuntu 11.04 with chrome browser version 25.0.1364.160 installed. From table Ⅲ and table Ⅳ, we can see the system accurately detect the operation of security violation in confidentiality and integrity from domains that are not trusted, and the execution overhead of rewrite code is about 3x. In Table Ⅲ, as the number of domains increases, the number of accesses to sensitive data is increased, and the number of violation of confidentiality and integrity is increased accordingly. The violation ratio is about 51%, close to the trustability 50%. In Table Ⅳ, the number of domains is fixed at 100 with the trustability set to different value, the second column of the number of sensitive data flow accessed and third columns of the total number of data flow is almost the same, while the four and five columns of violation is inversely proportional to the first column of trustability, and the taint ratio is close to untrustability, which equals '1 - trustability'.

We applied this method to the case of XSS. We use examples from webgoat that is an open source j2ee security project and it contains different kinds of real vulnerability. From table Ⅴ, three different kinds of XSS including Dom XSS, Reflect XSS and Store XSS and a mixed XSS PhishXSS are tested. We can see the accessed sensitive data are cookie, forms, input, URL, and image.src, and the average proportion of sensitive data flow is 6.07%. It shows that it is feasible and practical for our method to detect XSS in real web applications.

TABLE III. STATISTIC OF SENSITIVE INFORMATION ACCESS AND VIOLATION WHEN TRUSTABILITY = 50%

bm	dn	Flow count		Taint flow		Time(s)		Ratio(%)	
		sfc	*afc*	*vcc*	*vic*	*ori*	*rw*	*sr*	*tr*
2d	2	182	1069	79	15	0.157	0.38	17.03	51.65
10d	10	933	5590	405	78	0.372	1.06	16.69	51.77
50d	50	4653	27950	2025	378	1.695	7.50	16.65	51.64
100d	100	9499	56900	4140	749	4.568	15.59	16.69	51.15
Avg	60	3794	22877	1662	205	1.698	6.14	16.77	51.55

TABLE IV. STATISTIC OF SENSITIVE INFORMATION ACCESS AND VIOLATION WITH DIFFERENT TRUSTABILITY WHEN DOMAINS=100

Trust (%)	Flow count		Taint flow		Time(s)		tr (%)
	sfc	*afc*	*vcc*	*vic*	*ori*	*rewrite*	
0	10300	59900	8776	1500	4.281	15.988	99.77
10	10418	59900	8010	1348	4.106	15.981	89.83
30	10260	59900	6230	1050	4.125	15.785	70.96
50	10097	59900	4450	747	4.252	15.594	51.45
70	9933	59900	2670	443	4.33	15.557	31.34
100	10269	59900	0	0	4.251	15.654	0
Avg: 43.3	10212	59900	5022	848	4.208	15.75	57.22

TABLE V. STATISTIC OF SENSITIVE INFORMATION ACCESS WITH WEBGOAT XSS ATTACK

bm	sfc	afc	sr (%)	sn	Accessed sensitive data
DomXSS	553	8771	6.3	7	cookie,forms,URL,image.src
ReflectXSS	774	15592	4.96	6	cookie,forms,URL,image.src
StoreXSS	1382	19848	6.97	6	cookie,forms,URL,image.src
PhishXSS	569	9007	6.30	6	Forms,image.src,input
Avg	908	14815	6.07	6	

VI. RELATED WORK

There has been a lot of work applying information flow [1, 2, 3] to formalize fine-grained isolation. Several static techniques guarantee that certain kinds of inputs do no flow into certain outputs. These include type system [4], model checking [5], and data flow analysis [6]. Unfortunately, fully static techniques [7, 8] are not applicable in our setting, as parts of the code only become available at run time, and as they often rely on the present of the underlying program structure.

Several researches has focus on using of dynamic taint [9, 10, 11, 12] tracking techniques, aiming to address the problem of identifying and separating untrusted data from the HTML output to ensure that untrusted data is sanitized before it is output. Two widespread techniques of dynamic taint analysis and symbolic execution are combined to analyses malware and discovery vulnerability in [9], and dynamic taint propagation for the java virtual machine is employed in [12]. For client side, such as XSSAuditor [13], are implemented in modern browsers, which are useful in nullifying common attack scenarios by observing HTTP requests and intercepting HTTP responses during the browser's parsing. AjaxScope [14] applies on-the-fly instrumentation of JavaScript code as it is sent to users browsers, and provides facilities for reducing the client-side overhead and giving fine-grained visibility into the code-level behaviors of the web applications.

Client [11, 16] side approaches try to protect sensitive information leakage by preventing attempts to send the sensitive data to third servers. Our approach is a client solution building on a proxy-based [15] rewriting framework for dynamically enforcing security policies, which provides great confidentiality and integrity for sensitive data. The main difference between our solution and other client-based approaches is that they use various heuristics for XSS detection, whereas we perform an in-depth and precise analysis of how sensitive data are propagated in the client. Using dynamic analysis, we can accurately identify information flows that static approaches cannot identify.

VII. CONCLUSION

In this paper, we propose a framework based on the dynamic information flow analysis to detect XSS. Our solution can accurately mark and trace the information flow of sensitive data, and detect the irregular accesses of sensitive data from untrusted domains. Because it does not need to modify the server or browser code, the approach can be easily deployed for enhancing the security of web applications.

ACKNOWLEDGMENT

This research was supported in part by the National Natural Science Foundation of China under grants 61173166 and 61272190, the Program for New Century Excellent Talents in University, the open research fund of Zhejiang Provincial Key Lab of Data Storage and Transmission Technology, Hangzhou Dianzi University (No. 201302)

REFERENCES

[1] D.E.Denning. A lattice model of security information flow. communications of ACM, vol 19, May, 1976, pp. 236-243.

[2] D.Hedin, A.Sabelfeld. Information flow security for a core of JavaScript. In Computer Security Foundations (CSF), 2012, pp. 3-18.

[3] L.Zheng and A.C.Myers. Dynamic security labels and static information flow. International Journal of Information Security, vol 6, March 2007, pp. 67-84.

[4] L.Meyerovich and G.Livshits. Conscript: Specifying and Enforcing Fine-Grained Security policies for JavaScript in the browser. In Proceeding of the 2010 IEEE Symposium on Security and Privacy, pp. 481-496, 2010.

[5] M.S.Lam,M.Martin, V.B.Livshits, and J.Whaley.Securing. web applications with static and dynamic information flow tracking. In PEPM, pp. 3-12, 2008.

[6] M. Martin and M. S. Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In 17th USENIX Security Symposium, 2008.

[7] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In PLDI, 2009.

[8] S.Guarinieri and VB.Livshits. Gatekeeper:mostly static enforcement of security and reliability policies for JavaScript code. In Usenix Security, 2009.

[9] E.J.Schwartz, T.Avgerinos, D.Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution. In Proceedings of the 2010 IEEE Symposium on Security and Privacy, pp. 317−331, 2010.

[10] Chin.E, Wagner.D. Efficient character-level taint tracking for java. In Proceedings of the 2009 ACM Workshop on Secure Web Service, pp. 3-12, New York, 2009.

[11] P.Vogt, F.Nentwich, N.Jovanovic, E.Kirda, C.Krugel, and G.Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In NDSS, 2007.

[12] V.Haldar, D.Chandra, and M.Franz. Dynamic Taint Propagation for Java. In Twenty-First annual Computer Security Applications Conference(ACSAC), 2005.

[13] D.Bates, A.Barth, C.Jackson. Regular expressions considered harmful in client-side xss filters. In Proceedings of the 19th International Conference on World Wide Web, WWW 2010, ACM, pp. 91−100.

[14] E.Kiciman,B.Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In Proc.SOSP 2007, pp. 17-30, 2007.

[15] H.Kikuchi,D.Yu, and A.Chander. Javascript Instrumentation in Practice. In APLAS 2008, pp. 326-341.

[16] Engin Kirda. Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks. In Proceedings of the 2006 ACM Symposium on Applied Computing, 2006.

[17] J.Goguen and J.Meseguer. Security Polices and Security Models. In IEEE Symposium on Security and Privacy, 1982.