Cross-Site Scripting Attacks in Social Network APIs

Yuqing Zhang
University of Chinese Academy of Sciences
Beijing, China,
zhangyq@ucas.ac.cn

Xiali Wang
University of Chinese Academy of Sciences
Beijing, China,
wangxl@nipc.org.cn

Qihan Luo
University of Chinese Academy of Sciences
Beijing, China,
luoqh@nipc.org.cn

Qixu Liu
University of Chinese Academy of Sciences
Beijing, China,
liuqixu@ucas.ac.cn

***Abstract*—Nowadays, it is becoming more popular that RESTful APIs are used by web developers to enhance the functionality of websites. However, this might raise potential XSS attack threats. Unlike traditional XSS attacks, XSS attacks in this scenario may take advantage of more characteristics of RESTful APIs. RESTful APIs are common in social networks. Consequently, in this paper, we took social networks as motivating examples to illustrate XSS attacks in RESTful APIs.**

This paper presents the first systematic and deep security analysis on XSS attacks in RESTful APIs in social networks. We designed a tool to automatically detect XSS vulnerabilities in APIs and discovered several serious XSS flaws in eleven popular social networks. We also examined 143 web-based apps and verified the prevalence of Cross-API XSS (XAS) vulnerabilities. Based on the results, we conclude the root causes of XAS vulnerabilities and explain their differences to traditional XSS vulnerabilities in depth. Finally, we propose preliminary measures both for social networks and third-party application developers to alleviate XAS.

Keywords- Web security; social eco-system; RESTful APIs; Cross API Scripting; Cross Site Scripting; social network APIs

I. INTRODUCTION

More and more websites are opening their cloud services to third-party developers as RESTful Application Programming Interfaces (RESTful APIs) [1]. These APIs are introduced to open up a channel where third-party applications can interact with those websites for data and resources. RESTful APIs are common especially in social networks. Consequently, in this paper, we take social networks as motivating examples to illustrate security problems in RESTful APIs. Social network APIs have promoted the forming of social eco-systems which are composed of not only social networks, but also all kinds of third-party applications and Internet services (e.g. Web mash-up applications). It was reported that by the end of March 2012, more than 9 million apps and websites had been integrated with *Facebook* [2]. While social eco-systems bring convenient and integrated experiences to their users, social networks and their users are confronted with more and more security problems.

Cross Site Scripting (XSS) vulnerabilities in social network APIs have been exposed in the wild. A XSS flaw was found in twitpic.com [3] in May 2009, which was due to the missing sanitization of a *Twitter* API response. In March 2011, a XSS flaw in *Facebook* mobile API allowed the attacker to launch a self-propagating spam worm [4]. During the same period, a security researcher discovered a self-XSS flaw in a *Google Code* example page due to the *Google* Map API [5]. All these flaws can be exploited directly or indirectly to compromise the privacy of the users. These cases indicate that new threats involved with XSS in social network APIs have arisen. We need to further understand the causes and details of the XAS flaws lying behind the APIs.

In order to differentiate from traditional XSS attacks, we referred to XSS flaws exploited via RESTful APIs as *Cross-API Scripting (XAS for short)*. We discuss the difference between traditional XSS and XAS after we give analysis on some real-world XAS cases in Section II.

So far, studies on XAS are rare. In 2009, Hristo Bojinov et al. [6] analyzed a new type of vulnerability called Cross-Channel Scripting (XCS). XCS used electronic devices such as security cameras to launch XSS attacks in web interface. In their paper, XSS in insecure RESTful APIs were regarded as an example to prove the existence of *reverse* XCS. The security characteristics and causes on RESTful APIs were not deeply studied. Moreover, the examples given were only one type of XAS attacks.

In this paper, we provide the first systematic study on analyzing XAS flaws in social eco-systems. We designed a tool to identify XAS flaws within social APIs in eleven popular social networks and detected a variety of security issues such as *tainted* API responses (API responses containing unsanitized user inputs, e.g. on *Facebook* and *LinkedIn*), inconsistent handling of user-input data (e.g. on *Twitter* and *Flickr* APIs), and incorrect API responses (e.g. on *t.qq.com* and *t.sohu.com* APIs). Our analysis supports that design and implementation faults of social APIs contribute to XAS.

Further probing of 143 web-based applications turns out to confirm the fact that XAS has become an extended threat against Web applications, especially social eco-systems. Of the total 143 web applications, 107 were found vulnerable to XAS. We reported several vulnerabilities to corresponding operators and all of the flaws we reported had been fixed by the time we finished this paper. Finally, we also proposed preliminary measures to mitigate XAS.

Contribution. This is the first paper to propose systematic and deep security analysis on XAS attack and demonstrate its threats to real-world social networks in detail.

(1) We analyze real-world XSS vulnerabilities in social APIs and summarized their unique features compared with traditional XSS vulnerabilities.
(2) We implement a fuzzing tool to detect XAS vulnerabilities automatically. Utilizing this tool, we further discuss the root causes of XAS by discovering XAS flaws in eleven popular social networks and 143 third-party online applications.
(3) Based on our security analysis of XAS, we propose preliminary measures to mitigate XAS.

Organization. The rest of the paper is organized as follows. In Section II, we demonstrate the XAS attack cases in real world. In Section III, we propose the design of our XAS detecting tool and in Section IV we conduct experiments and analyze the results in depth. Then, we give preliminary mitigation techniques against XAS in Section V. We introduce related work in Section VI, and finally conclude our work in Section VII.

II. XSS IN SOCIAL NETWORK APIS

In this section, we examine RESTful APIs in many popular social networks including *Twitter*, *Facebook*, *LinkedIn*, *Renren*, *Weibo*, *etc*. We also checked for XAS within social networks themselves.

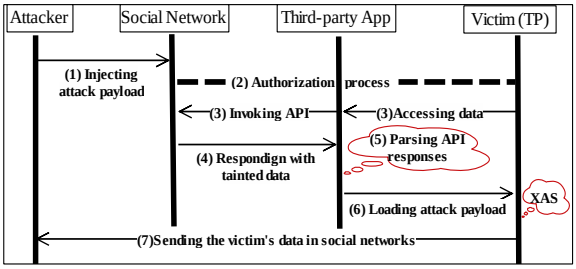

Figure 1. A typical XAS attack on third-party apps

A typical XAS attack is illustrated in Figure 1. The XAS attack process is described as follows:

Step 1: *The attacker injects malicious code into the social network via web UI or APIs to affect a news feed. The tainted feed will be displayed to all of the attacker's friends on the social network.*
Step 2: *The victim who is a friend of the attacker on the social network authorizes the third-party app to access his / her personal data.*
Step 3: *The vulnerable application invokes APIs to retrieve the tainted data from social network.*
Step 4: *The data with malicious code is sent to the app.*
Step 5: *The third-party app parses the tainted API responses and generates HTML responses for the victim.*
Step 6: *The app sends responses containing malicious code to the victim's browser.*
Step 7: *The attack payload is executed in the victim's browser and the victim's data in social networks is sent to the attacker.*

In the rest of this section, we present five case studies to demonstrate different types of XAS in real-world web-based applications.

A. Mash-up Applications

Mash-up apps provide integrated and convenient management for users' accounts on different social networks. We give three instances of powerful second-line attacks against social networks below.

• ***Controlling the mash-up app accounts***

TweetDeck is one of the most popular mash-up applications, managing social networks including *Twitter*, *Facebook*, and *Foursquare*. The Chrome extension version of *TweetDeck* is vulnerable to XAS due to its failure to sanitize the tainted API responses from *Facebook* and *Foursquare*. Specifically, all contents are HTML-escaped except for the *GroupName* field from *Facebook* APIs and the *FirstName* and *LastName* fields from *Foursquare* APIs. Although Chrome's extension architecture is designed with security considerations [7] [8], script execution vulnerabilities in Chrome's extensions still make websites vulnerable to attack [9] [10].

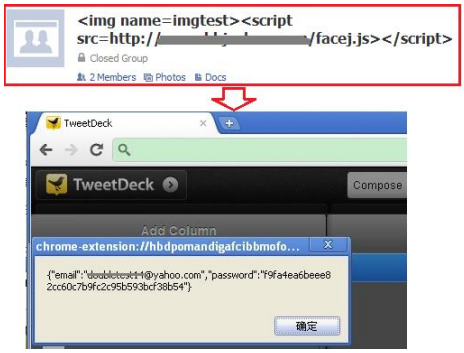

Figure 2. Injecting POC in Facebook to take over a the TweetDeck account

By exploiting XAS flaws in the *TweetDeck* Chrome extension, the attacker can control the victim's *TweetDeck* username and password which are stored in the *LocalStorage* of Chrome. Figure 2 shows the attack process of stealing *TweetDeck* accounts. First, the attacker injects malicious code into the *GroupName* field in his *Facebook* profile, as shown in the upper part enclosed with a red pane in Figure 2. Second, when another group member accesses the group in *TweetDeck*, the code is executed to obtain the account information and transmit it to the attacker. The proof of concept to exploit the XAS vulnerability is given as follows:

```
function exploit()
{
Document.all.imgtest.src=http://www.XXX.com/XXX.asp?name="
+escape(document.title)+"&supper="+escape(window.localStorage.
getItem("tweetdeck_account"));
}
setTimeout("exploit()", 3000);
```

• ***Injecting malicious code into visited Web pages via extension permission vulnerability***

We also found that the permission vulnerability in *TweetDeck* Chrome extension could be exploited to break same origin policy and launch XAS attack to control different social networks. The fragment of manifest file in *TweetDeck* Chrome extension is as follows:

```
"permissions": [ …"tabs", … "https://*.twitter.com/",
"http://search.twitter.com/", "http://*.tweetdeck.com/",
"https://*.tweetdeck.com/", …. "https://*.facebook.com/", … ]
```

As a result, the extension has the privilege of injecting code into sites like *Twitter* and *Facebook*. Hence, the attacker can inject malicious JavaScript code into the victim's *Facebook* and *Twitter* pages via the vulnerable extension when these websites are logged in at the same time. This class of attacks indicates that XAS flaws in extensions can bring second-level XSS attacks to the Web pages that the vulnerable extensions include in their manifests.

• ***Collecting social data by harnessing mash-up worms***

With further research, we found that some mash-up applications accessed more data from social networks than they actually needed. Taking *HootSuite* as an example, when we authorized it to connect *Facebook*, *HootSuite* was granted: accessing basic information, profile information, family & relationships, etc.; managing our pages, our events, our custom friend lists, etc. Apparently, nearly all kinds of user data in social networks can be leaked indirectly from high-profile applications like *HootSuite* if these apps are vulnerable to XAS.

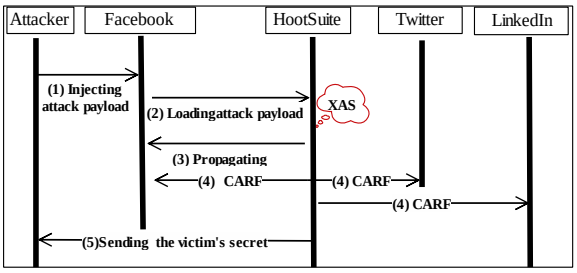

Figure 3. A CARF worm against social networks in HootSuite

Vulnerable mash-up apps such as *HootSuite* can be viewed as a base for attackers to launch XAS worms and collect private data from multiple social networks. Since these worms utilize the API features of the mash-up apps, we call them **CARF (Cross-API Request Forgery) worms**. A typical CARF worm attack against *HootSuite* is illustrated in Figure 3. The steps of the attack are depicted as follows:

Step 1: *The attacker injects the worm payload into Facebook.*

Step 2: *The payload is loaded into HootSuite from the victim's Facebook via Event or Group API when the victim browses his / her social data in HootSuite. Then, the worm payload will be executed.*

Step 3: *The worm injects its payload into the victim's Facebook* page.

Step 4: *The worm payload sends requests to all the integrated social networks to steal the victim's social secrets.*

Step 5: *The worm payload sends stolen secrets to the attacker.*

B. *Interconnected Services*

Some third-party applications play a part in interconnecting social networks and other services (e.g. mail services) to become an interrelated eco-system. The interconnection is often done in two styles. The first is to provide features from social networks to other services. The second is to provide stream synchronization of one social media with another.

• ***Webmail and gadget services***

As displayed in Figure 4, malicious JS code is previously injected into a feed's *Description* field on the attacker's *Facebook* wall. When the victim opens an email from or to the attacker, *Rapportive* will load the evil code via *Facebook* APIs. The code subsequently launches a CSRF attack in the context of the victim's session to control his or her *Gmail* account. Although *Gmail* itself is secure enough, the XAS vulnerability introduced by third-party applications can compromise it.

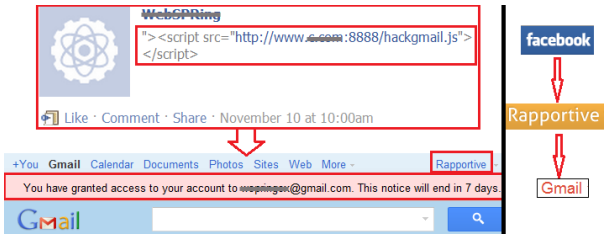

Figure 4. An XAS attack in Gmail

Similar to *Rapportive*, many gadgets are developed to deliver news feeds from social networks to *iGoogle* or *Gmail*. We probed eight gadgets for potential XAS: three for *Facebook*, three for *Twitter*, one for *Flickr*, and one for *Renren*. Surprisingly, except for one Facebook gadget (link: http://facebookiggadget.appspot.com/), the others were all vulnerable to XAS owing to their direct display of the insecure data from social networks. By abusing the social features, the unique characteristics of XAS or the implicit trust between users and *Google* services, attackers can launch malicious attacks (e.g. phishing and theft of private data) stealthily via XAS flaws in these vulnerable gadgets [11]. The only difference from the case of *Rapportive* is that the affected party is third-party gadgets.

• ***Stream synchronization services***

There are many stream synchronization services such as *ifttt.com*, *facebook.involver.com* and *tarpipe.com*. For example, photo streams in *Flickr* can be synchronized to *Facebook* via a third-party online application named "*Flickr*

for Pages" (*facebook.involver.com*). We found XAS vulnerability in this application framed in *Facebook*. Attackers can exploit the XAS flaws to conduct phishing attacks, spoofing for malware attacks [12], or sharing photos abusively. API-based applications not only provide a new channel to leak social media data, but also breach the firm safeguards established for Internet services.

C. Desktop Applications

Social APIs also have led to the bloom of third-party desktop applications. *Pokki* is a desktop application running Web extensions on Windows. It's built on Chromium sandboxing, WebKit and V8 JavaScript engine. Insecure APIs lead to XAS in *Pokki's* extensions for *Facebook*, *Tumblr*, *Gmail*, *Instangram*, *etc.*

Figure 5. XAS attack on Pokki's extension for Tumblr

Figure 5 depicts a XAS attack on Pokki's extension for *Tumblr*. The attacker first injects malicious code via *Tumblr*'s webpage. Then the XAS vulnerability in *Pokki*'s extension for *Tumblr* invokes the execution of the code. For all extensions in *Pokki*, cross-origin *XMLHttpRequest* is supported and *LocalStorage* is used to store data retrieved by extensions. As a result, although sandbox is used, attackers can still steal the victim's privacy data even when the victim's system cannot be accessed.

D. Third-party Mobile Clients

More and more third-party mobile applications are designed for social networks while mobile devices are becoming smarter and more prevailing. We probed nine *Twitter* mobile Web applications including its official mobile version, and found six of them as listed in TABLE I were vulnerable to XAS caused by directly displaying the tainted data from *Twitter's Search* or *List* APIs.

TABLE I. NINE TWITTER MOBILE WEB APPLICATIONS

Vulnerable		Not Vulnerable
m.slandr.net	twetmob.com	mobile.twitter.com
dabr.co.uk	itweet.net	twittme.mobi
m.tweete.net	www.tweetree.com	www.twittermobile.net

A recent report [13] has highlighted the growing use of mobile devices to connect with social networks and that it's becoming a preferred method for cyber criminals to spread malware. These vulnerable third-party mobile social apps will likely be opportunities to boost the trend.

E. Social Networks

XAS attacks also occur within social networks themselves. In total, there are at least four potential situations to lead to XAS vulnerabilities in the context of social networks. We analyze them one by one hereafter.

- ***Insecure internal APIs***

Several social networks employ internal APIs to contribute to their own functionality. *Foursquare* loads user data to Web pages on the server side via JSONP (JSON Padding) generated by internal APIs. The common ground for them is that user-input data is HTML-escaped on the client side. We refer to this API invoking as static loading of API responses. The scripts quoted in the JSONP are treated as valid code to be executed due to error-tolerant HTML document parsing of browsers. Therefore, *Foursquare* suffers from XAS since tainted user data is encapsulated in the JSONP directly. A static JSONP loading of API responses is like:

```
<script type="text/javascript">//<![CDATA[
fourSq.tiplists.setupHistoryPageListControls
([{"id":"v4e90699293adc15b620c2632","todo":false,"done":true,"visit
edCount":1,"venue":
{"id":"4e90699293adc15b620c2632","name":"{name}","contact":
{},"location":{"address":"<script>alert(document.domain);</script>",
"crossStreet":"<script>alert(document.domain);</script>",
"lat":44.3,"lng":37.2,"city... ...}}]);
```

In this case, social networks suffer for their own vulnerable APIs. These vulnerabilities are unequivocally harmful to social networks without any mitigating factors.

- ***Less safeguards taken for APIs than Web UI***

So far, less attention has been paid to the security of RESTful APIs than web interfaces. According to our examination, *Tumblr* and *Renren* were vulnerable to XAS. Testing on *Tumblr* APIs, we found that two functional APIs *Text* and *Video* had XAS vulnerabilities. However, posting text or videos via Web interfaces was not vulnerable. For another example, user-input data from *blog.addBlog* API in *Renren* was displayed without HTML-escaping, resulting in XAS vulnerability in *Renren*. Similarly, posting blogs from the Web interface caused no problem.

Besides the above cases, microblog services *t.163.com* and *t.sohu.com* are exposed to XAS due to missing HTML-escaping for certain APIs: *statuses/retweet/:id* in *t.163.com* and *direct_messages/new* and *account/update_profile* in *t.sohu.com*. These flaws can only be exploited via flawed APIs rather than corresponding web interfaces.

As we see, these cases indicate that social networks make inconsistent treatments on user-input data from different channels: RESTful APIs and traditional Web interfaces. RESTful APIs turn out to be weaker on security than Web interfaces.

- ***More controllable fields***

While invoking APIs, third-party applications may manipulate some input fields which do not exist in traditional Web interfaces of social networks. As a result, these input fields look like overlooked areas of security during development. We noted that the *message* field in *Renren* API *-checkins.checkin* - could be manipulated to bring about

XAS. The XAS payload was executed in the context of *Renren* profile and home page.

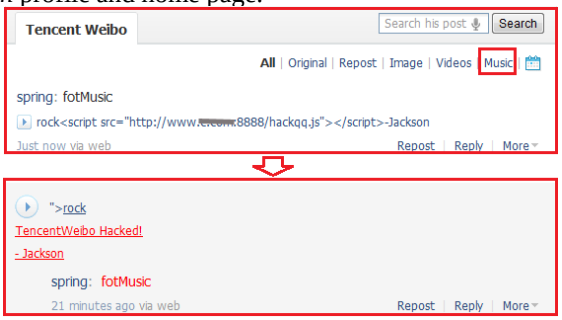

Figure 6. XAS attack in t.qq.com

Figure 6 describes a XAS attack on *t.qq.com*. The upper part enclosed with a red pane is a post injected with XAS payload in the title field via API *api/t/add_music*. When the victim clicks the Music link, the payload will be executed stealthily. We gave a POC in the lower part enclosed with a red pane, writing "*TencentWeibo Hacked!*" and highlighting it.

- ***Incorrect API response***

Generally, a legal API response is supposed to be in the format of JSON or XML. However, several exceptions exist in actual deployment. *t.sohu.com*, for example, returned their API response in a HTML format. In the case of *t.qq.com*, it did return a JSON-formatted API response to its users but the Content-Type header was set as "text/html" instead of "application/json". As a consequence, an evil third-party application could add unsanitized code into the responses of *t.sohu.com* and *t.qq.com* based on OAuth protocol, and the malicious code would be then parsed in the victim's browser as HTML files.

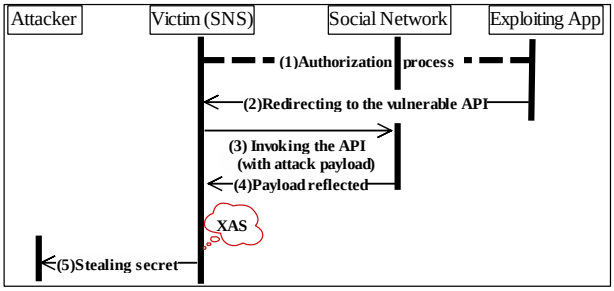

Figure 7. An XAS attack based on OAuth in social networks

The complete attacking process is depicted in Figure 7. A third-party application (exploiting application) is needed to trick victims to complete the OAuth process for a successful XAS exploit. In this type of XAS, a redirected request to vulnerable API is needed. APIs with incorrect response format include *api/statuses/home_timeline*, *api/private/recv* and *api/private/send* in both *t.sohu.com* and *t.qq.com.*

F. The Features of XAS

Based on the instances illustrated, we concluded the following new features and potential exploiting conditions for XAS in social ecosystems:

- ***Malicious code transmitted through RESTful APIs.*** In order to exploit XAS vulnerabilities, malicious code is transmitted via RESTful APIs either from social networks to third-party applications or in reverse. As a result, it's more complicated to launch XAS than traditional XSS.
- ***Inherited social relationship.*** All the XAS vulnerabilities can be exploited based on the social relationship of users in social networks, regardless in social networks or third-party applications. In third-party apps, social relationship is inherited from connected social networks and can be harnessed by attackers.
- ***Not limited by same-origin policy (SOP).*** Social networks are able to provide APIs to third-party applications without same-origin limitation. This is because invoking APIs is generally accomplished in two modes: making requests on (1) the server side rather than the client side or (2) the client side with Access-Control-Allow-Origin mechanism. Hence, XAS attacks in third-party apps can affect connected social networks directly although most websites are still protected by SOP.
- ***Affect multiple parties.*** APIs interconnect multiple parties and SOP does not exist between these parties. Therefore, XAS in third-party applications, especially mash-ups, is more destructive and can affect multiple parties including third-party applications and integrated social networks.

III. XAS Detection Tool

To systematically analyze the security implications of XAS flaws, we designed a tool automatically detecting XSS vulnerabilities in social APIs. In this section, we present the overview of our tool and describe challenges in our implementation.

A. Design Overview

Almost all social network APIs are RESTful. A RESTful API is represented by a unique URI. JSON and XML are the principal data formats of social API responses. These formats are more normalized than HTML. Generally, the supported operations of social API include four standard HTTP methods: POST, GET, PUT and DELETE.

TABLE II. An example of normalized API entries

Auth_Method = OAuth2.0	CallMethod = POST
API_Provider = dev.facebook.com	ParamsCount = 1
API_Key = 191742207560268	Param0 = msg
API_Secret = af6ddd003cc0e2de697ace0406d4dfc8	Type0 = String
Response_Format = JSON	Initial_value0 = Test
Scope = publish_stream, create_event, …	DoTest0 = true
Authorization_URI = https://www.facebook.com/dialog/oauth	
Access_Token_URI= https://graph.facebook.com/oauth/access/token	
API_ URI=https://graph.facebook.com/***/comments?message=Test	

We implemented a tool to assist us in identifying API flaws. The architecture of our XAS detection tool is portrayed in Figure 8. Our tool is composed of two units: configuration and detection. The goal of the configuration unit is to convert raw API entries into normalized API entries. Raw API entries only contain API URI and invoking methods. They are manually extracted from API documents

in platforms of social networks. An example of normalized API entries is given in TABLE II. The normalized API entries contain all the indispensable information used for detection unit.

In the detection unit, identification of API flaws is based on regular expression matching. The detection unit first injects test vectors which are valid JavaScript code to the API parameter when invoking configured Web APIs. When the responses are received, the tool analyzes whether the responses contain tainted user-input data or are ill-formed.

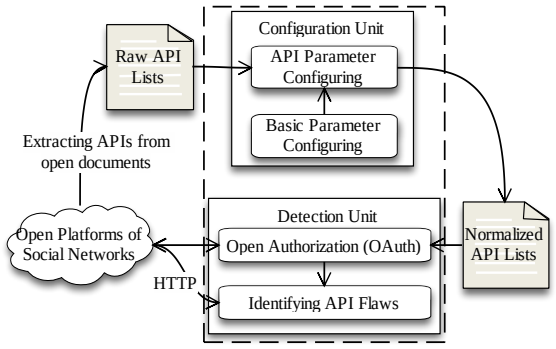

Figure 8. Architecture overview of our tool identifying Web API flaws

Our tool identified *tainted* API responses according to the following three rules:

(1) If the API response contains the JavaScript code we inject as API parameters, the response will be identified as tainted. (The JavaScript code we inject into the parameters of the API request is called test vector. The code is chosen randomly from our previously created file containing XSS testing vectors.) In other words, if the test vectors occur in the API response, the response is tainted.

(2) If the API response contains simple-escaped test vectors in which the character "/" is converted into "\/" and "'" into "\'", the response is identified as tainted, too. This is because such escaping doesn't interfere with those responses which could cause potential XAS in third-party applications. For instance, injected vector "<script>alert(/xas/);</script>" is escaped into "<script>alert(\/xas\/);<\/script>" in an API response, and the escaped vector will be unescaped automatically after that API response is parsed in third-party applications.

(3) If the API response contains the Unicoded or the Hex-encoded form of the test vectors like "\u003Cscript\u003E alert(131425); \u003C\/script\u003E" and "\x3c iframe onload=alert (/xas/)>\x3e", the response is also identified as tainted. Although Unicoded or Hex-encoded test vectors cannot be executed directly, the tainted API response can still potentially affect third-party applications since the encoded JavaScript code will be decoded implicitly when third-party applications parse the API response.

Our tool also identified ill-formed API responses, containing two aspects: (1) Content-Type Header is incorrectly configured, e.g. "Content-Type: text/html"; (2) the response is in HTML format rather than expected JSON or XML.

B. Implementation Challenges

When identifying the flaws of these social APIs, we needed to address the following challenges in the implementation of our fuzzing tool:

- ***URI path parameters***

The supported types of parameters in social APIs are GET query parameters, POST parameters and URI path parameters (e.g. "***:id"*** is a parameter in this *Twitter* API: *http://api.twitter.com/1 /statuses/**:id**/retweeted_by/ids.json*). URI path parameters in the APIs of different social networks have diverse styles. We designed a regular expression "(/:\w+(-\w+)*)[/|\?|\.]" to match all the potential URI path parameters based on our analysis over all the tested social APIs.

- ***Rate limiting***

Most social networks only allow third-party applications to invoke APIs for limited times in a specific interval, and it is even stricter before applications are verified formally.

In our study, regular expressions of HTML-escaped vectors are used to identify whether injected test vectors in any API response are sanitized. If so, our tool would skip the current API parameter and go to the next. This mechanism is effective to control the rate of API calls.

We configured tested APIs from each social network before detecting the API flaws. For each configured API, there are two levels: for each parameter of API, (1) *API_URI*, *Call_Method* and *Parameter_Count* are the first level, (2) and the four-tuple {$Name_i$, $Type_i$, $Initial\_value_i$, $DoTest_i$} is the second level. $DoTest_i$ is the test flag marking whether parameters are tested or not. The possible values for test flags are true or false. This configuration allows us to avoid unnecessary API calls, e.g. "type" usually represents the response type of APIs and makes no sense in our experiment.

In addition, the following principles are completed in the configuration step of tested APIs to assure that all initial values of API parameters are valid:

(1) Assign a random value for parameters according to the marked type of API documents, and check whether these parameters are independent and free from any other constraints.

(2) Generate valid values for dependent parameters by calling proper independent APIs. Dependent parameters are generated by social networks, for example, the ID of a blog in *Facebook* is dependent on *Facebook* system.

According to the principles, valid API calls can be guaranteed to a great extent and the rate of API invoking can be kept under the limitation.

- ***Multiple OAuth versions***

Authentication and authorization mechanisms are adopted to protect social networks and their users' security and privacy. OAuth 1.0a [15], OAuth 1.0 [16] and OAuth 2.0 [17] are the principal adopted protocols. Social networks may deploy different versions of OAuth. Hence, we needed

to integrate all three OAuth protocols in our tool to test APIs validly.

IV. Results and Analysis

In our experiment, we tested APIs in the following eleven popular social networks: *Twitter*, *Facebook*, *Foursquare*, *LinkedIn*, *Flickr*, *Tumblr*, *Renren*, *Weibo*, *t.qq.com*, *t.163.com* and *t.sohu.com*. The social networks were selected since they had millions of registered users and provided a sample for different categories of social network services (e.g. friendship, location-based service, etc.). In this section, we summarized the results of APIs' security properties testing and provided evidence to illustrate the prevalence of XAS. We also analyzed the root causes of XAS.

A. Results

Commonly, there are two ways to escape user inputs. One is to escape user inputs when they are sent to the server and then stored in sanitized form in the database. The other is to store user inputs as they are and to escape them when they are displayed. The latter must be done by third-party websites. We mark these two HTML-escaping methods as **Scheme I** and **Scheme II** respectively for convenience.

If either scheme is deployed, the attack of XAS will be prevented. However, since the HTML-escaping tasks of the two schemes are undertaken by different parties (i.e. the social network and the third-party application separately), the inconsistency in handling the escaping often occurs and leads to potential XAS attacks on third-party applications. For instance, *Twitter* deployed Scheme I to most of its APIs and Scheme II to its *Search* and *List* APIs. If a third-party application using *Twitter* did not deploy Scheme II, user inputs via *Search* or *List* APIs would not be escaped.

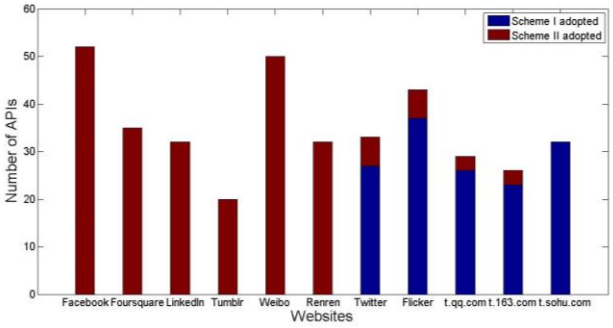

Figure 9. The ratios for adoped HTML-escape schemes in tested APIs

The number of sites adopting different HTML-escape schemes for tested APIs in terms of JSON response format is provided in Figure 9. *Twitter*, *Flickr*, *t.qq.com* and *t.163.com* employ inconsistent HTML-escape schemes in the same response format and Scheme I is the principal one. *Facebook*, *Foursquare*, *LinkedIn*, *Tumblr*, *Renren* and *Weibo* only employ Scheme II and thus all their APIs respond the same as user-input data without HTML-escape. Only *t.sohu.com* solely adopts Scheme I. Interestingly, some social networks deploy inconsistent HTML-escape schemes for different API response format, and e.g. *Facebook* and *Renren* adopts Scheme I for API responses in XML while adopting Scheme II for those in JSON.

On one hand, the inconsistent deployment of Scheme I and Scheme II leads to XAS attacks on third-party applications. On the other hand, incorrect API responses as mentioned in Section II.E may cause XAS attacks on Social Network itself. XAS attacks on both social networks and third-party apps will affect the security and privacy of users in social networks.

TABLE III. API FLAWS AND VALID HTML TAGS DISCOVERED

		Twitter	Facebook	Foursquare	LinkedIn	t.qq.com
The API Flaws	ISSRF	√	×	×	×	√
	ISDRF	×	√	-	×	×
	ICT	√	√	×	×	√
	ICF	√	×	×	×	×
	VHT	<p>, <a>	<p>	-	-	<a>

		Tumblr	Renren	Weibo	Flickr	t.163.com	t.sohu.com
The API Flaws	ISSRF	×	√	×	√	√	×
	ISDRF	-	√	√	×	×	×
	ICT	×	√	×	√	√	√
	ICF	×	×	×	×	√	√
	VHT	-	<p>	-	<a>	<a>	-

ISSRF: Inconsistent HTML-escape Schemes for the Same Response Format **ISDRF:** Inconsistent HTML-escape Schemes for Different Response Format (JSON and XML). **ICT:** Incorrect Content-Type in API responses. **ICF:** Incorrect Content Format in API responses. **VHT:** Valid HTML Tags in normal API responses (VHT is not a flaw but a feature of tested APIs).

"√" denotes the corresponding flaw exists. "×" denotes the corresponding flaw doesn't exist. "-" for the API flaws denote XML response format is not supported. "-" for VHT denotes no valid HTML tags exist in the normal API responses.

With the results of our experiment, we summarized the flaws relating to all the tested APIs in TABLE III. We divided all the API flaws we tested into four categories, namely, Inconsistent HTML-escape Schemes for the Same Response Format (ISSRF), Inconsistent HTML-escape Schemes for Different Response Format (ISDRF), Incorrect Content-Type in API responses (ICT), and Incorrect Content Format in API responses (ICF). The abbreviations are explained below TABLE III. Flaws of ISSRF and ISDRF are caused by the inconsistent deployment of Scheme I and Scheme II. Meanwhile, flaws of ICT and ICF are caused by incorrect API responses.

The statistics in TABLE III indicate that all or part of APIs in all tested social networks respond with tainted data without HTML-escaping. Furthermore, certain APIs in *Twitter*, *t.163.com* and *t.sohu.com* respond in HTML format when an invalid parameter is provided. Fortunately, the APIs in *Twitter* and *t.163.com* are not vulnerable to XAS because API responses in *Twitter* don't include any user-input data and API responses in *t.163.com* encapsulate the HTML-escaped user-input data. APIs provided by *Facebook*, *Twitter*, *Flickr*, *Renren*, *t.qq.com*, *t.163com*, *t.sohu.com* are all configured with incorrect Content-Type, including text/html, text/javascript and text/plain.

In addition, we found that all the tested APIs in the social networks allowed certain simple valid HTML tags in their normal responses. In our statistics, only two HTML tags

were used: <*a*> and <*p*>. In other words, all third-party applications we tested added <*a*> and <*p*> tags to their HTML responses to users.

B. Prevalence of XAS

The analysis clearly indicates that a new challenge has arisen for online social eco-systems: APIs have extended the attack surface of social networks to third-party applications, and the security and privacy of users in social networks face threats from API channels due to poor design, implementation and invoking of Web APIs. In order to confirm the prevalence of XAS vulnerabilities in real world, we examined 143 web-based applications for the eleven tested social networks. These examined applications all met a condition that users' data in social networks was stored or retrieved and displayed on them via APIs.

TABLE IV. THE RATIOS OF XAS FLAWS DUE TO DIFFERENT CAUSES IN EXAMINED APPLICATIONS

	Twitter	Facebook	Foursquare	LinkedIn	t.qq.com
Scheme I	-	-	-	-	1/15
Scheme II	13/21	17/19	7/8	8/9	9/15
API Response	-	-	-	-	1/15

	Tumblr	Renren	Weibo	Flickr	t.163.com	t.sohu.com
Scheme I	-	-	-	-	1/11	4/11
Scheme II	3/5	11/12	17/21	9/11	5/11	-
API Response	-	-	-	-	-	1/11

"-" denotes the website does not contain corresponding flaws of a certain cause. "A/B" denotes the ratio of XAS flaws due to a certain cause where "B" represents the total number of third-party applications we checked in the website and "A" represents the number of third-party applications containing XAS flaws of a certain cause.

As shown in TABLE IV, 107 examined applications are vulnerable to XAS. In *t.qq.com*, we checked fifteen third-party applications in total and one of them were vulnerable to XAS because *t.qq.com* stored user data without HTML-escape (Scheme I was not deployed to some APIs). Nine out of fifteen were found vulnerable to XAS since these nine third-party applications did not escape user inputs on some APIs (Scheme II was not deployed to some APIs). Another third-party application for *t.qq.com* could be leveraged to cause XAS due to the incorrect API response.

C. Analysis

From the results of our experiment, we argue that it's better for social networks than third parties to take principal responsibility for XAS mitigation. Users' data in social networks are shared via APIs to numerous third-party applications and any tainted data via APIs, from social networks to third-party applications or in reverse, is apt to cause XAS vulnerabilities if missing sanitization in any party. Scheme I and Scheme II are the common two ways to escape user inputs. As shown in TABLE IV, when only Scheme I is deployed, XAS is less likely to occur. (Theoretically, when Scheme I is deployed, XAS will be prevented. However, in the actual deployment of Scheme I, social networks often miss escaping a number of APIs and thus XAS still occurs.) On the contrary, when only Scheme II is deployed, we detected XAS vulnerabilities in the majority of our examined third-party applications. Consequently, the deployment of Scheme I makes a better contribution in blocking XAS than the deployment of Scheme II. In other words, if social networks deploy Scheme I for all their APIs in any response format, it is expected that XAS flaws will be reduced significantly, since the sanitized data is ready for all the third-party developers and they can dedicate themselves to the features of their apps.

Actually, APIs in social networks which employ Scheme II are still likely to cause XAS flaws in third-party applications even when developers pay attention to sanitizing of API insecure responses. There are four major reasons for these XAS flaws:

- Data in some API responses may include diversified fields which comprise a single post. These fields are generally a share post, an album feed, etc. When sanitizing these fields separately, third-party developers are apt to overlook certain fields. Two Facebook *iGoogle Gadgets* and *Gmail* are exposed to XAS flaws mainly due to these insecure fields.
- In third-party applications, some required fields (e.g. user name and group name in social networks) are likely to be treated as credible items without any sanitization.
- In practice, some social networks embed certain HTML tags into API responses for decorating data simply, such as <*a*>. However, this makes third-party developers confused when sanitizing the API responses.
- Social networks make inconsistent treatments on user-input data from different channels: APIs and traditional Web interfaces, such as the cases illustrated in the second case of Section II.E.

In the cases of inconsistent HTML-escape schemes, another two reasons also contribute to XAS vulnerabilities:

- As shown in Figure 9, Scheme II is always considered as an additional sanitization method by social networks. This implies that only few APIs respond with tainted data so that third-party developers are prone to consider all the responses from APIs as being sanitized.
- Even when user inputs are carefully sanitized at input time, social networks can suffer from XAS attacks because API is also an input channel where HTML-escaping sanitization at input time is likely to be missed.

Based on the above findings and analysis, we were convinced that little attention had been paid to XAS vulnerabilities and more threats were unearthed in the wild.

V. MTIGATION MEASURES

As an extended version of XSS, it is more complicated to mitigate XAS than XSS. Traditional XSS defenses based on browser-web application collaboration, such as BLUEPRINT [19], DSI [20] and Noncespaces [21], were only suitable for the first case in Section II.E, but not effective for other types of XAS attacks because more than one part was affected. Besides, some server-side schemes [22] [23] [24] may not be feasible to mitigate XAS because

there were too many third-party apps for popular social networks. There were also some client-side solutions, such as Noxes [25], Spectator [26] and MPP [33], which could contribute to detecting XAS attacks in client-side. In this section, we recommended some preliminary measures to mitigate XAS in social networks and third-party applications.

A. For Social Networks

The following rules provide suggestion for developers of Web APIs to prevent XAS attacks:

(1) All the API responses should be set with proper Content-Type headers: "application/json;charset=utf-8" for JSON responses and "text/xml;charset=utf-8" for XML responses.

(2) All the API responses should have consistent data format whatever user-input data is provided to API parameters, rather than responses in HTML format for invalid API invoking.

(3) User-input data from APIs should be sanitized in the same way as data from Web UI.

(4) Data should be loaded dynamically on the client side rather than statically on the server side via JSONP if APIs are used in social networks.

(5) All the user-input data of all the APIs should be handled with Scheme I regardless of the API response formats if possible.

B. For Third-Party App Developers

HTML-escape of API responses appears to be subtle in third-party applications. However, according to our statistics in TABLE III only two simple HTML tags <*a*> and <*p*> were used in responses of some APIs. This means that the application developers can apply a white list to replace dangerous characters with HTML-escaped ones before parsing API responses. The process of our mitigation measure contains two steps:

(1) The characters "<", ">" and their valid encoding expressions (including the Hex-encoded and Unicoded ones) in API responses are all HTML-escaped.

(2) The tags in the white list are once again unescaped to meet the intention of normal API responses.

VI. Related Work

XSS Analysis. Some research on XSS focused on the sanitization [27] in Web application frameworks and the trend [28] of XSS. In [27], the authors evaluated the XSS abstractions in fourteen major commercially-used Web frameworks and extracted the requirements of XSS sanitization primitives. Moreover, Ref. [29] [30] [31] were dedicated to discovering XSS vulnerabilities based on identifying faulty sanitization procedures and untrusted data.

XSS worms in social networks are another point worth studying. Analysis [32] and defense [33] related works have been done over XSS worms. However, there has been no deep discussion on the security implications associated with XSS based on Web APIs of social networks.

Web APIs Analysis. APIs are considered powerful agents for expanding functionalities of social networks. Hence, APIs and third-party applications based on them are concerned as another area requiring privacy protection. In 2008, Adrienne Felt et al. [35] addressed the privacy risks associated with social network APIs by presenting a privacy-by-proxy design for preserving privacy. In 2009, Kapil Singh et al. [36] presented a privacy control framework to control what untrusted third-party applications could do with the information they received. In addition, Rui Wang et al. [37] found many serious logic flaws in leading merchant websites that accepted payments through third-party cashiers. Those flaws resulted from the complexity for an application to coordinate its internal states with third-party services via APIs and the Web client across the Internet. InteGuard [39] offered the first security protection against logic flaws in social API, instead of protection against XAS. All the APIs related works didn't focus on XAS.

VII. Conclusions

In this paper, we presented the first comprehensive analysis on XSS vulnerabilities exploited via Cross-API Scripting (XAS) and discussed their difference to traditional XSS attacks. We demonstrated several XAS cases in different contexts and illustrated new critical threats and unexpected exploiting opportunities for both server security and user privacy in social eco-systems.

Furthermore, we designed a tool to assist us in analyzing the design and implementation flaws of RESTful APIs. In our experiment, we chose APIs in eleven popular social networks as well as 143 third-party apps as our test objects. The results showed that all the social networks suffer from XAS flaws. According to our findings, we summarized XAS causes in depth. Finally, we provided preliminary measures to mitigate XAS for both social networks and third-party applications.

The interaction among diversified Internet services has become more and more frequent due to use of RESTful APIs in social networks. API flaws have brought about new security challenges to both social networks and other Internet services. Our contribution to analysis over XAS is limited and more concerns need to focus on it in the future.

Acknowledgements

We thank Charles Reis and the anonymous reviewers for their suggestions. This work was supported by the National Natural Science Foundation of China (Grant No. 61272481).

References

[1] Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures. *Doctoral dissertation, University of California, Irvine*, 2000.

[2] Facebook. News on Facebook Platform, 2013. http://newsroom.fb.com/content/default.aspx?NewsAreaId=137.

[3] Ryan Naraine. Twitter API ripe for abuse by Web worms, 2009. http://www.zdnet.com/blog/security/twitter-api-ripe-for-abuse-by-web-worms/3451.

[4] Softpedia.com News. Facebook Mobile API XSS Vulnerability Used To Launch Spam Worm, 2011. http://cyberinsecure.com/facebook-mobile-api-xss-vulnerability-used-to-launch-spam-worm/.

[5] Amol Naik. Exploitation of "Self-Only" Cross-Site Scripting in Google Code, 2011. http://www.exploit-db.com/ download_pdf /17017/.

[6] Hristo Bojinov, Elie Bursztein and Dan Boneh. XCS: Cross Channel Scripting and its Impact on Web Applications. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[7] Adam Barth, Adrienne Porter Felt, Prateek Saxena and Aaron Boodman. Protecting Browsers from Extension Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[8] Opera. Opera Extensions: Quick Documentation Overview, 2010. http://dev.opera.com/articles/view/opera-extensions-quick-documentation-overview/.

[9] Taras Ivashchenko. Web Application Vulnerabilities in Context of Browser Extensions, 2011. http://oxdef.info/papers/ext/chrome.html.

[10] Lei Liu, Xinwen Zhang, Guanhua Yan and Songqing Chen. Chrome Extensions: Threat Analysis and Countermeasures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[11] Robert Hansen and Tom Stracener. Xploiting Google Gadgets: Gmalware and Beyond. In *Black Hat 2008 USA*, 2008.

[12] Jason Adriaan. Why Facebook should Police their API, 2011. http://www.bandwidthblog.com/2011/05/05/why-facebook-should-police-their-api/.

[13] AVG Community. Threat Report Q1 2012. http://www.avg.com.au/files/media/avg_threat_report_2012-q1.pdf.

[14] Emanuele Gentili, Alessandro Scoscia and Emanuele Acri. Cross Application Scripting. *Milan Security Summit 2010*.

[15] Mark Atwood, Dirk Balfanz and Darren Bounds, et al.. OAuth Core 1.0 Revision A, 2009. http://oauth.net/core /1.0a/.

[16] E. Hammer-Lahav. RFC 5849, The OAuth 1.0 Protocol, 2010. http:// tools.ietf.org/html/rfc5849.

[17] E. Hammer-Lahav. The OAuth 2.0 Authorization Protocol, 2011. http:// tools.ietf.org/html/draft-ietf-oauth-v2-22.

[18] XSS (Cross Site Scripting) Cheat Sheet, 2011. http://ha.ckers.org/xss.html.

[19] Mike Ter Louw, V.N. Venkatakrishnan. BLUEPRINT-Robust Prevention of Cross-Site Scripting Attacks for Existing Browsers. In *Proceedings of the 30th IEEE Symposium on Security & Privacy*, 2009.

[20] Yacin Nadji, Prateek Saxena and Dawn Song. Document Sructure Integrity: A Robust Basis for Cross-Site Scripting Defense. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2009.

[21] M. Van Gundy and H. Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2009.

[22] Prithvi Bisht, V.N. Venkatakrishnan. XSS-GUARD-Precise Dynamic Prevention of Cross-site scripting attacks. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2008.

[23] Jin-Cherng Lin and Jan-Min Chen. The Automatic Defense Mechanism for Malicious Injection Attack. In *Proceedings of 7th International Conference on Computer and Information Technology*, 2007.

[24] Martin Johns, Bjorn Engelmann, and Joachim Posegga. XSSDS: Server-Side Detection of Cross-Site Scripting Attacks. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, 2008.

[25] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks.In *Proceedings of the 21st Annual ACM Symposium on Applied Computing*. New York, USA: ACM, 2006: 330-337.

[26] Benjamin Livshits, Weidong Cui. Spectator: Detection and Containment of JavaScript Worms. In *Proceedings of USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Boston, USA: ACM, 2008: 335-348.

[27] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, et al. A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS)*, 2011.

[28] Theodoor Scholte, Davide Balzarotti and Engin Kirda. Quo Vadis? A Study of the Evolution of Input Validation Vulnerabilities in Web Applications, 2011. http://www.iseclab.org/papers/vuln_fcds.pdf.

[29] Balzarotti, D., Cova, M., Felmetsger, V., et al. Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[30] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

[31] Saxena, P., Hanna, S., Poosankam, P., Song, D.: FLAX: Systematic discovery of client-side validation vulnerabilities in rich Web applications. In *Proceedings of 17th Annual Network & Distributed System Security Symposium*, 2010.

[32] Mohammad Reza Faghani and Hossein Saidi. Social Networks' XSS Worms. In *Proceedings of the International Conference on Computational Science and Engineering*, 2009.

[33] Fangqi Sun, Liang Xu, and Zhendong Su. Client-Side Detection of XSS Worms by Monitoring Payload Propagation. In *Proceedings of the 14th European Conference on Research in Computer Security*. Saint-Malo, France: ACM, 2009: 539-554.

[34] Thomas, K., Grier, C., and Nicol, DM. unFriendly: Multi-Party Privacy Risks in Social Networks. In *Proceedings of the 10th International Conference on Privacy Rnhancing Technologies*, 2010, Springer- Verlag, pp. 236–252.

[35] Adrienne Felt and David Evans. Privacy Protection for Social network APIs. In *Proceedings of the IEEE Web 2.0 Security and Privacy Workshop (W2SP)*, 2008.

[36] Kapil Singh, Sumeer Bhola and Wenke Lee. xBook: Redesigning Privacy Control in Social network Platforms. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[37] Rui Wang, Shuo Chen, XiaoFeng Wang, Shaz Qadeer. How to Shop for Free Online: Security Analysis of Cashier-as-a-Service Based Web Stores. In *Proceedings of the 32nd IEEE Symposium on Security & Privacy*, 2011.

[38] Luyi Xing, Yangyi Chen, XiaoFeng Wang, Shuo Chen. InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations. In *Proceedings of 20th Annual Network & Distributed System Security Symposium*, 2013.