Mitigating Cross-Site Form History Spamming Attacks with Domain-Based Ranking

Chuan Yue

University of Colorado at Colorado Springs
Department of Computer Science, Colorado Springs, CO 80918, USA
cyue@eas.uccs.edu

Abstract. Modern Web browsers often provide a very useful form autocomplete feature to help users conveniently speed up their form filling process. However, browsers are generally too permissive in both saving form history data and suggesting them to users. Attackers can take advantage of this permissiveness and use malicious webpages to inject a large amount of junk or spam data into the form history database of a browser, performing invasive advertising or simply making this useful form autocomplete feature almost useless to users. In this paper, we illustrate that this type of *cross-site form history spamming attacks* can be feasibly achieved at least on the recent versions of Mozilla Firefox and Google Chrome browsers. We inspect the autocomplete feature implementations in open source Firefox and Chromium browsers to analyze how basic and advanced cross-site form history spamming attacks can be successful. Browser vendors are apparently taking active measures to protect against these attacks, but we explore a different approach and propose a domain-based ranking mechanism to address the problem. Our mechanism is simple, transparent to users, and easily adoptable by different browsers to complement their existing protection mechanisms. We have implemented this mechanism in Firefox 3 and verified its effectiveness. We make our Firefox 3 build available for download and verification.

1 Introduction

Modern Web browsers often provide a very useful form autocomplete feature to help users speed up their form filling process. This feature keeps remembering a user's form filling history, predicts what a user wants to type in a new form field, and pops up a list of candidate history input values for a user to reuse. By directly selecting a matching history input value, a user can effectively reduce the keyboard typing time and effort.

However, browsers are generally too permissive in both saving form history data and suggesting them to users. When browsers save form history data, they disregard the origin of the form data. No matter on which website the form is submitted, the form field data are simply saved to the history database as name-value pairs. When browsers suggest a list of candidate history input values to a user, they again disregard the origins of both the current Web form and the history data. Normally, all the form field history values that have the matching field name and matching field value pattern with the current input field will be selected from the history database and displayed in the form autocomplete dropdown list. Such a permissive behavior is common to the three most popular browsers Internet Explorer, Mozilla Firefox, and Google Chrome.

A direct consequence of this permissiveness is that attackers can use malicious webpages to inject a large amount of junk or spam data into the form history database of a browser. These data may contain advertisement words used for getting financial or political benefits, or they are simply meaningless words used for nullifying the form autocomplete feature. The malicious webpages can trigger automatic form submissions, can inject all kinds of name-value combinations into the form history database of a browser, and can use different tricks to have their submitted field values ranked as top candidates by browsers. Later on, when users need to fill out forms on other legitimate websites, a large number of spam input values will be prompted by browsers as top form autocomplete candidates. This result may force users to read advertisement spams, may force them to spend more time and effort in choosing the desired reusable input value, and may also confuse them by suggesting many irrelevant junk input values. We call this type of attacks *cross-site form history spamming attacks*, and we illustrate in Section 3 that these attacks can be feasibly achieved at least on the recent versions of two popular browsers Firefox and Google Chrome.

Generally speaking, there could be two approaches to defending against cross-site form history spamming attacks. The first approach is to prevent the insertion of automatically submitted spam data into the form history database; the second approach is to prevent the suggestion of spam data values to a user. The former restricts the permissiveness of browsers in saving form history data, and the latter restricts the permissiveness of browsers in suggesting candidate form data. Both approaches have their own merits, and ideally browsers should leverage the advantages of both approaches to provide a strong protection against cross-site form history spamming attacks.

Browser vendors such as Google and Microsoft are apparently taking active measures to protect against cross-site form history spamming attacks [12,13]. However, they mainly took the first approach to fix the form autocomplete bugs in their browsers. Their efforts reduced the chances for attackers to inject junk form data, but still any spam that does make it into the form history database is persistent and stays until the user cleans it up. Moreover, the *cross-site autocomplete* phenomenon (Section 3) still exists in the latest versions of the three most popular browsers Internet Explorer 8.0.7600.16385, Google Chrome 8.0.552.237, and Firefox 3.6.13. Therefore, the form autocomplete feature in those browsers may still not able to ensure the availability of the most relevant form history data to a user (Section 4).

In this paper, we take the second approach and propose a domain-based ranking mechanism to restrict the suggestion of irrelevant form history field values to a user. We choose to take the second approach because no serious attention has been given to it yet, and we believe a solution with this approach could be beneficial to all popular browsers. We choose to explore the domain-based ranking mechanism because it is simple, transparent to users, and easily adoptable by different browsers. Meanwhile, our domain-based design philosophy is consistent with those of other efforts in securing browsers [1,3,5,6,10,14].

The basic idea of our domain-based ranking mechanism is to first ask a browser to also remember the origin domain of the form data when it saves a new name-value pair into the form history database. Later on, whenever a user needs to fill out a form on a webpage, the domain of the current website will also be inspected so that matching form history data with the matching domain will always be ranked higher than other matching form history data without the matching domain. This mechanism always allows users to conveniently select the most relevant form history data, thus rendering the cross-site form history spamming effort of attackers futile. We have implemented this defense mechanism in Firefox 3 and verified its effectiveness (Section 5). We make our Firefox 3 build on openSUSE Linux accessible to researchers for verifying its correctness and usefulness. Our source code is also available upon request.

The main contributions of this paper are as follows: (1) We illustrate that cross-site form history spamming attacks are feasible and can make the very useful form autocomplete feature almost useless to users. (2) We inspect the autocomplete feature implementations in open source Firefox and Chromium browsers to analyze how basic and advanced cross-site form history spamming attacks can be successful. (3) We propose a domain-based ranking mechanism for protecting browsers from those attacks. This mechanism is simple, so its correct implementation can be ensured with high confidence. It is completely transparent to users, so they need not to be trained or informed of its use. It is not bound to browsers' any specific support of DOM, event model, and JavaScript, so it can be easily adopted by different browsers as a defense-in-depth mechanism to complement their existing protections to the autocomplete feature.

2 Background

The form autocomplete feature is supported by the four most popular browsers: Internet Explorer, Firefox, Google Chrome, and Safari. Another popular browser Opera does not have a form autocomplete feature. The basic functioning of the form autocomplete feature is simple: it saves what a user has previously submitted in the *text* type input fields of a form into a history database, and makes the saved history data available to reuse to help speed up a user's future form filling tasks. A user is allowed to turn on or off this feature by configuring a preference option in the browser.

Some browsers also support another related feature called form autofill. Strictly speaking, form autofill is different from form autocomplete. Form autofill requires a user to explicitly save personal information such as name and address through the browser's form autofill user preference interface, and then allows the user to reuse the saved information to fill the corresponding forms. Table 1 lists the support of the form autocomplete and form autofill features in the recent versions of different browsers; it also lists the locations for configuring these two features.

Table 1. The support of form autocomplete and form autofill features in browsers

Browser	Auto-complete	Auto-fill	Configuration Location
Internet Explorer (8.0.7600.16385)	Yes	No	Tools → Internet Options → Content → AutoComplete → Settings
Firefox (3.6.10 to 3.6.13)	Yes	No	Tools → Options → Privacy → History → Remember search and form history
Google Chrome (6.0.472.63, 7.0.517.41, 8.0.552.237)	Yes	Yes	Options → Personal Stuff → Autofill options → Enable Autofill to fill out web forms in a single click
Safari (5.0.1 7533.17.8, 5.0.2 7533.18.5)	Yes	Yes	Preferences → AutoFill → AutoFill web forms
Opera (10.62, 10.63)	No	Yes	Settings → Preferences → Forms

From this table, we can see that the feature support, feature configuration locations, and even the usage of terms are inconsistent among different browsers. For one example, Internet Explorer and Firefox only have the form autocomplete feature. For another example, Google Chrome uses the same term autofill to represent both the autocomplete and autofill features, although internally they have different interfaces and implementations. Therefore, we want to make it clear that our work only focuses on the form autocomplete feature. The key reason is because attackers can simply use malicious webpages to abuse this feature (next section), but it is much harder for them to directly inject junk form data through a browser's form autofill user preference interface.

Two more autocomplete related features in modern browsers are also out of the scope of this paper. One is the website password remembering feature. This feature is normally controlled by the password manager of a browser, and similarly it is very difficult for attackers to directly manipulate it via malicious webpages. The other feature is the location bar autocomplete. This feature guesses what a URL address a user wants to type in the address bar based on what websites the user has visited before. Attackers can also inject junk data to a browser's URL location history database, but the severity of such attacks is limited. This is because browsers such as Firefox and Google Chrome do not save URLs with invalid domains and meanwhile it is difficult to load a large number of URLs from the address bar without being perceived by a user.

3 Attacks

In this section, we illustrate and analyze the cross-site form history spamming attacks. First, we use some concrete examples to show that modern browsers are generally too permissive in both saving form history data and suggesting them to users. Second, we construct basic attacks to demonstrate that cross-site form history spamming is feasible. Third, we construct advanced attacks by inspecting the implementation of the form autocomplete feature in open source Firefox and Chromium (which provides source code for Google Chrome) Web browsers; we illustrate that such advanced attacks can make the very useful form autocomplete feature almost useless to users.

3.1 The Permissiveness in Autocomplete

First, let us open either a Firefox or a Google Chrome browser window, and make sure its form autocomplete feature is not turned off (Table 1 has the configuration location information). Then, we visit the following three groups of example websites listed in Table 2 and submit forms on them.

Table 2. Three groups of example websites

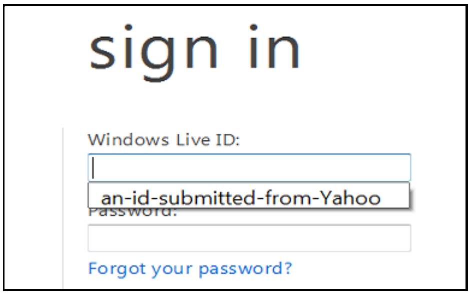
Website	Webpage Form	Field Label / Name
login.yahoo.com	Sign in to Yahoo!	Yahoo! ID / login
login.live.com	sign in	Window Live ID / login
amazon.com	Sign in	My e-mail address is / email
facebook.com	login	Email / email
usps.com	Search USPS.com	*empty* / q
flickr.com	SEARCH	*empty* / q

For the first group, we submit the "sign in" form on either login.yahoo.com or login.live.com. We can randomly type in some dummy ID and password combinations in our login attempts, and we do not need to ensure those attempts to be successful. It is easy for us to observe that a submitted Yahoo! ID on login.yahoo.com will appear in the autocomplete dropdown list when we plan to type in a Windows Live ID on login.live.com as shown in Figure 1. Similarly, a submitted Windows Live ID on login.live.com will also appear in the autocomplete dropdown list when we plan to type in a Yahoo! ID on login.yahoo.com. We refer to this phenomenon as *cross-site autocomplete.*

We can observe the same phenomenon between the login forms of the two websites listed in the second group of Table 2. We can also observe this phenomenon on many other types of form submissions. For example, in the third group, the search data submitted on usps.com will be suggested to us in the autocomplete dropdown list when we plan to type in the search form on flickr.com, and vice versa. All these observations are common to the latest versions of Google Chrome (6.0.472.63 to 8.0.552.237) and Firefox (3.6.10 to 3.6.13).

We also observed the cross-site autocomplete phenomenon on these three groups of websites in Internet Explorer 8 before a security update was automatically applied on October 12, 2010 [13]. With that update, Internet Explorer 8.0.7600.16385 disables its form autocomplete feature on https webpages. However, the cross-site autocomplete phenomenon still exists in Internet Explorer on http webpages such as the ones listed in the third group of Table 2. We did not observe the cross-site autocomplete phenomenon on the Safari Web browser. Indeed, we found that the autocomplete feature on Safari 5.0.1 7533.17.8 and the latest version 5.0.2 7533.18.5 does not work at all, and form data are not saved even for the same form of the same website.

The key reason why such a cross-site autocomplete phenomenon occurs is because modern browsers are generally too permissive in both saving form history


sign in

Windows Live ID:

an-id-submitted-from-Yahoo

Password:

Forgot your password?

Fig. 1. The cross-site autocomplete phenomenon

data and suggesting them to users. Modern browsers simply disregard the important origin domain information in their autocomplete feature design. When browsers save form data, they save the form field name-value pairs without saving the corresponding domain information. When browsers suggest a list of candidate form history input values to a user, they select values that have the matching field name and matching field value pattern[1] with the current input field regardless of the current domain name. For example, the two input fields Yahoo! ID and Windows Live ID in the first group have the same input field *name="login"*, the two input fields in the second group have the same input field *name="email"*, and the two input fields in the third group have the same input field *name="q"*. This is why the cross-site autocomplete phenomenon can be observed between the two websites in each group.

It is reasonable to assume that the original autocomplete feature designers had the intentions to promote cross-site autocomplete so that form data submitted from different websites can be ubiquitously reused. However, considering the widespread growth of Web-based threats in recent years [7,17], we believe it is time now to revise the design of this feature to restrict its permissiveness. Otherwise, as we show below, attackers can launch cross-site form history spamming attacks to perform invasive advertising or simply make the very useful form autocomplete feature almost useless to users.

3.2 Basic Spamming Attacks

Cross-site form history spamming attacks take advantage of browsers' permissiveness in autocomplete to inject a large amount of junk or spam data into the form history database of a browser. They may force users to read advertisement spams, may force them to spend more time and effort in choosing the desired reusable input value, and may also confuse them by suggesting many irrelevant junk input values. To be successful, such an attack needs to satisfy two basic conditions: the first condition is to attract a user to visit a malicious webpage,

[1] The field name is used to select the initial candidate list; the field value pattern is used to refine that initial list once a user begins to type something in the input field.

and the second one is to trigger automatic form submissions on the malicious webpage to spam the form history database.

There is no silver bullet for preventing the satisfaction of the first condition. For example, attackers can host malicious webpages on their own websites, or they can inject malicious scripts to compromise webpages hosted on legitimate websites. They can also use phishing attacks or malicious advertisements to further allure users to visit malicious webpages.

```
<html>
<head>
<script language="javascript" type="text/javascript">
function submitJunkData() {
  var charSet = "ABCDEFGHIJKLMNOPQRSTUVWXTZabcdefghiklmnopqrstuvwxyz0123456789._";
  var newValue = "";
  for (var i=0; i<8; i++) {
    var index = Math.floor(Math.random() * charSet.length);
    newValue += charSet.charAt(index);
  }
  document.getElementById('fieldID').value = newValue;
  document.getElementById('formID').submit();
}
</script>
</head>

<body onload="setInterval('submitJunkData()', 300)">
<p>Contents that attract a user to stay on this webpage.</p>
<form id="formID" method="post" action="" target="iframe-hole" style="visibility:hidden">
<input type="text" name="login" id="fieldID" value="" size=100>
</form>
<iframe name="iframe-hole" width=0 height=0 style="visibility:hidden"></iframe>
<input type="text" name="somethingelse" value="" size=100>
</body>
</html>
```

Fig. 2. Malicious example one (attack1.htm)

Now, we focus on illustrating that the second condition for a successful cross-site form history spamming attack can be feasibly achieved at least on the recent versions of two popular browsers Mozilla Firefox (3.6.10 to 3.6.13) and Google Chrome (6.0.472.63). Figure 2 illustrates one malicious webpage example attack1.htm. Basically, once this page is loaded on a user's browser, it will automatically trigger the form submission action every 300 milliseconds. In each form submission, a randomly generated junk value for the input field *name="login"* will be submitted, and that value will be saved by the browser to its form history database. Later on, when a user fills an input form field with *name="login"* such as the Yahoo! ID field or the Windows Live ID field listed in Table 2, those saved junk data will be suggested to the user in the browser's autocomplete dropdown list.

Looking at the content of this webpage, we can identify that it uses the JavaScript setInterval function to periodically call the submitJunkData function to submit a special form. The form itself is invisible and it employs a "hole in the page" technique [16] to set its *target* attribute pointing to an invisible iframe. Using these techniques, the malicious webpage can submit the form in a hidden manner without reloading the page, so that a user's interactions with other

```
<html>
<head>
<script language="javascript" type="text/javascript">
function submitJunkData() {
  //This function is the same as that in attack1.htm
}
</script>
</head>

<body onload="setInterval('submitJunkData()', 300)">
<p>Contents that attract a user to stay on this webpage.</p>
<form id="formID" method="post" action="" target="_self" style="visibility:hidden">
<input type="text" name="login" id="fieldID" value="" size=100>
</form>
</body>
</html>
```

Fig. 3. Malicious example two (attack2.htm)

```
<html>
<head>
</head>
<frameset cols="1%,*" name="frameset_01">
  <frame src="attack2.htm" name="frame_01" noresize>
  <frame src="http://www.amazon.com" name="frame_02" noresize>
</frameset>
</html>
```

Fig. 4. Malicious example two (frameset.htm)

elements such as the "somethingelse" input field will not be affected. Therefore, it is hard for a user to notice the behind the scenes automatic form submissions.

Even without using the "hole in the page" technique, a malicious webpage that reloads itself can also be very stealthy. Figure 3 and Figure 4 illustrate such an example. The malicious webpage attack2.htm is almost identical to attack1.htm, except for (1) removing the hidden iframe and the "somethingelse" input field, and (2) setting the form *target="_self"* to reload the page itself. To hide the automatic form submissions from users, an attacker can use a framed webpage frameset.htm to include attack2.htm as a frame element. Therefore, when a user interacts with other frame pages of frameset.htm such as amazon.com, attack2.htm can continuously inject junk data to the browser's form history database.

In both attack1.htm and attack2.htm example webpages, the *target* attribute of the form is used to specify where to open the URL once the form is submitted. The *target* attribute of *<form>* is deprecated in HTML 4.01, but it is still supported by all the five popular browsers (Table 1) to maintain backward compatibility. However, attack2.htm can also be successful without using the deprecated *target* attribute. In such a case, attack2.htm can be modified by removing *target="_self"* and setting *action="attack2.htm"*. In these two example webpages, the intervals on how often to submit the form can also be dynamically adjusted by changing the setInterval function call. In addition, the *<body>* onload event handler in attack2.htm can be optionally changed from setInterval to setTimeout.

Last but not least, what we have introduced so far are just two examples of basic cross-site form history spamming attacks. We verified these attacks on the recent versions of Mozilla Firefox (3.6.10 to 3.6.13) and Google Chrome (6.0.472.63). There could exist many other techniques for performing the same or similar spamming attacks; however, the objective of our work is not to enumerate all the attacking techniques, but to protect the form autocomplete feature of browsers no matter what techniques are used by attackers to inject junk data into browsers' form history database. We also verified that these two example malicious webpages cannot successfully attack Internet Explorer 8 even before applying its security update [13]. They can still trigger periodical form submissions, but the submitted form data are not saved by Internet Explorer to its form history database. It seems that Internet Explorer 8 and the latest versions of Google Chrome (7.0.517.41 to 8.0.552.237) require a user initiated action to enter data into the autocomplete database, thus mitigating the attacks.

3.3 Advanced Spamming Attacks

Advanced cross-site form history spamming attacks improve over basic spamming attacks to make them more effective. Here, from different perspectives, we illustrate three types of techniques that could be used in advanced spamming attacks: the first one is *target selection techniques* that can increase the possibility for a browser to select junk data from its form history database to the autocomplete dropdown list, the second one is *ranking promotion techniques* that can improve the ranking of junk data in a browser's autocomplete dropdown list, and the third one is *value generation techniques* that can help keep junk data in the autocomplete dropdown list even when a user begins to type something in an input field.

Target selection techniques. focus on properly choosing form input field names to increase the possibility for a browser to select the submitted junk data. Recall that when browsers suggest a list of candidate history input values to a user, they will first check the *name* property of the *text* type input field and will only select history input data that have the matching field name with the current input field. This practice is common to the three most popular browsers Internet Explorer, Firefox, and Google Chrome[2]. Therefore, to make their attacks effective, attackers can select certain targeted field names that may match the field names in users' potential future form filling activities.

The targeted field names can be selected from different sources. To attack popular websites such as the ones listed in Table 2, attackers can visit those sites and directly extract their form input field names. To attack as many websites as possible, attackers can use tools to crawl the Web and identify those most commonly used form input field names. After selecting the targeted field names, attackers can use those names on a malicious webpage in different ways. For one example, they can simply put a list of *text* type input fields with different

[2] A small difference: Firefox and Google Chrome perform case sensitive field name comparison, but Internet Explorer performs case insensitive field name comparison.

```
<script language="javascript" type="text/javascript">
function submitJunkData() {
  //The code that constructs new newName and newValue
  //......
  document.getElementById('fieldID').name = newName;
  document.getElementById('fieldID').value = newValue;
  document.getElementById('formID').submit();
}
</script>
```

Fig. 5. Example JavaScript code that dynamically changes the name of a *text* type input field

names inside of a single form. Note that browsers often set an upper limit on the maximum number of input field values that can be saved to the history database in a single form submission. This limit is 100 in Firefox and 256 in Chromium. For another example, attackers can also dynamically change the names of existing input fields using JavaScript as shown in Figure 5. These techniques work on recent versions of Firefox and Google Chrome as listed in Table 1.

Ranking promotion techniques. focus on improving the ranking of junk data in a browser's autocomplete dropdown list. Normally, after selecting the initial candidate form history values based on the current input field name, browsers will rank those values with the expectation that one of the top-ranked values could be a user's desired input value. Browsers often only show those top-ranked values in the autocomplete dropdown list. For example, both Firefox and Google Chrome limit that at any time at most six suggested values can appear in the autocomplete dropdown list window for a *text* type input field. Firefox enables a scrollbar for the dropdown list to allow a user to view more suggested values, but Google Chrome does not enable a scrollbar for its autocomplete dropdown list. Therefore, to make their junk data occupy the limited positions in the autocomplete dropdown list and out-compete a user's desired history data, attackers have to raise the ranking of their junk data.

By inspecting the source code of Firefox and Chromium, we can identify exactly how these two browsers rank matching input field values. In Firefox, three main factors are considered and values that are used more times, used more recently, or used more frequently over a long period of time will be ranked higher based on a complex formula. In Chromium, the main factor is the total number of times used, and the larger this number the higher the ranking of a value. While there are some differences between these two browsers in ranking form history values, a common and reasonable criterion is the number of times that a value is used. Indeed, in the form history databases of both browsers, such a timesUsed number is associated with each unique *text* type name-value pair. When a form is submitted and if a submitted name-value pair already exists in the form history database, both browsers will simply increase the timesUsed number of this existing record by one.

Based on these analyses, we can see that attackers may raise the ranking of their injected junk form data by submitting the same name-value pairs multiple times. We provide such an example in Figure 6. Basically, in a malicious

```
<script language="javascript" type="text/javascript">
var months=['January','February','March','April','May','June',
            'July','August','September','October','November','December'];

function submitJunkData() {
  var index = Math.floor(Math.random() * 12);
  var newValue = months[index];
  document.getElementById('fieldID').value = newValue;
  document.getElementById('formID').submit();
}
</script>
```

Fig. 6. Example JavaScript code that repeatedly submits some name-value pairs in a form

webpage such as attack1.htm or attack2.htm, the submitJunkData function will be periodically invoked to repeatedly submit the names of the twelve months as junk data. If their numbers of submitted times become much larger than those of a user's desired values, these junk data can occupy the limited positions in the autocomplete dropdown list for a long time. We have also verified this example on recent versions of Firefox and Google Chrome as listed in Table 1.

Value generation techniques. can help keep junk data in the autocomplete dropdown list even when a user begins to type something in an input field. Normally, once a user begins to type something in an input field, browsers will immediately refine the values suggested in the dropdown list. Internet Explorer and Google Chrome only use prefix matching, i.e., those suggested values must begin with what a user has typed into the input field. Firefox uses both prefix matching and substring matching at the same time. When a user has typed more than one character into the input field, substring matching will include those values that contain (not necessarily prefix) this partial input into the suggested list. In all these three browsers, case insensitive comparison is used to match input field values.

Based on these practices in browsers, attackers may want to generate junk values that start with a user's partial input. While it is very difficult to predict a user's input, attackers can simply generate junk values that start with the permutation of two alphabets or other unique characters according to the purpose of an input field such as a username field or a zip code field. Therefore, the junk data could still be selected into the autocomplete dropdown list even after a user types in two characters. We should note that with each keystroke the user is willing to type, the attack work required increases exponentially. Another important consideration in value generation is that attackers may put advertisement words in junk values to get financial or political benefits. This could be the strongest incentive for attackers to perform the cross-site form history spamming attacks.

4 Defenses

In general, there could be two approaches to defending against cross-site form history spamming attacks. The first approach is to prevent the insertion of automatically submitted junk or spam data into the form history database. The second approach is to prevent the suggestion of spam data values to a user. Some efforts have been made very recently in browsers such as Google Chrome [12] and Internet Explorer [13] to protect the form autocomplete feature, and actually those efforts followed the first approach by requiring a user initiated action to enter data into the autocomplete database. However, taking the first approach alone does not necessarily solve the whole problem. On the one hand, security bugs and flaws in modern browsers are inevitable [2,3,8,9,11]; therefore, attackers may still able to inject junk data to the form history database by exploiting some vulnerabilities in browsers. Any spam that does make it into the form history database is persistent and stays until the user cleans it up. On the other hand, the cross-site autocomplete phenomenon illustrated in the last section still exists in all the three most popular browsers; therefore, the most relevant site-specific form history data may still not be selected as top autocomplete candidates.

Therefore, we argue that modern browsers should also take the second approach to better protect the autocomplete feature. We explore this second approach by proposing a domain-based ranking mechanism. This mechanism is simple, transparent to users, and easily adoptable by different browsers as a defense-in-depth mechanism to complement their existing protection mechanisms. In this section, we present and analyze our domain-based ranking mechanism.

4.1 The Domain-Based Ranking Mechanism

The key idea of this mechanism is very simple. When a browser saves a new name-value pair into the form history database, it will also remember the corresponding origin domain of the submitted form data. Later on, when the browser selects candidate form history values, it will inspect the domain of the current website and will always rank matching values that have the corresponding domain higher than other matching values that do not have the corresponding domain. For example, a user has submitted the “sign in” forms before on login.yahoo.com (see Table 2) using the *login* value “yahoo2010” and on login.live.com using the *login* value “live2010”. When the user visits login.yahoo.com again, the value “yahoo2010” will always be ranked higher than the value “live2010” and any *login* values submitted from other (potentially malicious) domains.

To integrate such a domain-based ranking mechanism into a browser’s form autocomplete feature, we need to make three modifications: updating the form history database, updating the form data saving functionality, and updating the autocomplete suggestion functionality. Using our implementation of this mechanism in Firefox 3 as the example, we present the details of these modifications.

Form history database:Browsers usually save form history data into a database. For example, both Firefox and Chromium use a zero-configuration transactional SQLite [15] database engine to store all the form history data. In the SQLite database of Firefox 3, the original form history table consists of six columns as shown in Table 3.

To save the origin domain information of a fieldname-value pair, we need to add one new column to this form history table. We name this new column as *baseDomain* and specify its data type as TEXT. Adding this new column to the

Table 3. Form history table modification in Firefox

The original version with the six columns						
id	fieldname	value	timesUsed	firstUsed	lastUsed	–
The new version with an added column baseDomain						
id	fieldname	value	timesUsed	firstUsed	lastUsed	**baseDomain**

form history table is straightforward and we simply modified the table creation SQL statement. However, considering the scenario that a *.sqlite* database file with the old version database schema may exist when a browser is upgraded to include our domain-based ranking mechanism, we also need to instruct the browser to perform a database migration to migrate old data records to the new form history table. In such a case, a default domain name will be used to indicate that those records came from the old version of browser. Actually in Firefox 3, timesUsed, firstUsed, and lastUsed were new columns added to the database schema version one. We strictly followed their migration steps to ensure the correctness of our migration implementation.

We should also note that originally, each record in the form history table is uniquely identified by a fieldname-value pair, and a unique id is generated to represent that pair. This practice is common to both Firefox and Chromium. With our modification to the form history table, each record is now uniquely identified by a triplet (fieldname, value, baseDomain), and a unique id is generated to represent that triplet. Thus, baseDomain will be taken into account in all the related database operations such as insertion and updating.

Form data saving: When a form is submitted and its data need to be saved to the form history database, a browser will notify its autocomplete feature to perform the task. We need to make modifications to this form data saving functionality so that the domain name information of the corresponding form data will also be saved.

To extract the domain name information, we consider two main factors: the source of the domain name and the parts of the domain name. In terms of the source, we extract the domain name of a form's owner document. In other words, we are interested in where exactly a form comes from, instead of what is the domain name of the top-level document. This design choice is reasonable because the owner document contains the most relevant information of a form. For example, on mashup websites, if a form is submitted from a sub-frame document, the domain name of the sub-frame document will be extracted and used by our mechanism. Therefore, the saved form history data can always be available no matter a owner document (e.g., www.flickr.com) is included in mashup websites as a sub-frame document (Figure 7(a)) or is directly loaded as a top-level document (Figure 7(b)) from a browser's URL address bar.

In terms of the parts of the domain name, we extract the base domain name and save it as baseDomain into the form history database (Table 3). For example, if a form's owner document domain name is login.yahoo.com, the extracted base domain name will be yahoo.com; if a form's owner document domain name is www.bbc.co.uk, the extracted base domain name will be bbc.co.uk. Once the

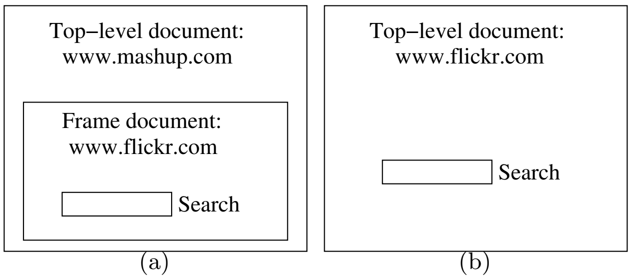


Fig. 7. Flickr.com (a) as a sub-frame document, (b) as a top-level document

base domain name is extracted, it will be used in the triplet (fieldname, value, baseDomain) for inserting or updating each unique form history record.

Autocomplete suggestion: When a user plans to fill out a form, a browser will search its form history database to check if any previously submitted *text* type input field values can be suggested to the user. In our mechanism, the initial search criterion is unchanged, i.e., all the records that have the matching field name with the current input field will be selected. However, with our modification, the values of the new added baseDomain column will also be retrieved for all the selected form history records. Then, our mechanism will mainly focus on adjusting the ranking of these selected records based on the domain name.

The first step is to extract the domain name of the current input form. Similar to what we modified in the form data saving functionality, here we also extract and use the domain name of the form's owner document. The second step is to compare the domain names. If the extracted owner document domain name contains the baseDomain of a selected form history record as the suffix, this record will be identified as a domain-matching record; otherwise, this record will be identified as a domain-mismatching record. For example, if the extracted owner document domain name is news.bbc.co.uk and a selected form history record has the baseDomain bbc.co.uk, this record will be identified as a domain-matching record. The way we use baseDomain and suffix matching in our mechanism is reasonable and it is consistent with the way HTTP cookie service of Firefox 3 checks whether a domain is a foreign domain for a cookie.

The ranking adjustment is purely based on the final ranking result of a browser's original algorithm. In other words, our mechanism does not manipulate the original formula used by a browser for ranking matching input field values; it simply adds one more adjustment step just before a browser suggests the ranked input field values to a user. In this final adjustment step, all the values of the domain-matching records will be moved ahead of all the values of domain-mismatching records. However, the relative ranking relations within the values of the domain-matching records will be preserved, and the relative ranking relations within the values of the domain-mismatching records will also be preserved. Meanwhile, if a value has appeared in a domain-matching record, that value will no longer be selected from any domain-mismatching record. Therefore, no duplicate values will be suggested to a user.

Table 4. An original ranking of “login” values

Rank	value	baseDomain
1	Eve	malicious.com
2	Chuck	malicious.com
3	Dave	yahoo.com
4	Carol	live.com
5	Fran	live.com
6	Bob	yahoo.com
7	Fran	yahoo.com
8	Alice	live.com

Table 5. The adjusted ranking on *.yahoo.com

Rank	value
1	Dave
2	Bob
3	Fran
4	Eve
5	Chuck
6	Carol
7	Alice

Table 6. The adjusted ranking on *.live.com

Rank	value
1	Carol
2	Fran
3	Alice
4	Eve
5	Chuck
6	Dave
7	Bob

For example, we assume that eight input values for the input field *name=“login”* are currently stored in our modified form history database. Two values were submitted from attack.malicious.com, and three values were submitted from login.yahoo.com, and another three values were submitted from login.live.com. Table 4 lists these eight values and their ranking calculated by a browser’s original algorithm. Our domain-based ranking adjustment is based on this original ranking. If a user plans to fill a “login” form field on any yahoo.com webpage, the three history values submitted from yahoo.com will be ranked higher than any other values as shown in Table 5. Similarly, if a user plans to fill a “login” form field on any live.com webpage, the three history values submitted from live.com will be ranked higher than any other values as shown in Table 6.

As we discussed in the last section, when a user begins to type something in an input field, browsers will immediately refine the values suggested in the dropdown list. In such partial input cases, our mechanism still simply performs the same domain-based ranking adjustment step based on the original ranking calculated by a browser’s original algorithm. Finally, the adjusted ranking results will be suggested to a user in the autocomplete dropdown list.

4.2 Security and Usability Analysis

The design philosophy of our domain-based ranking mechanism is consistent with those of other domain-based efforts in securing Web browsers [1,3,5,6,10,14].

In general, no matter attackers use what types of basic or advanced attacking techniques presented in Section 3, our domain-based ranking mechanism can always ensure that form history data previously submitted on the same domain can have the highest ranking in the autocomplete dropdown list. Note that newly visited websites will still be subject to the attack because there will be no previously submitted data to serve as good suggestions in this case. By ensuring the availability of those previously submitted most relevant form history data, our mechanism can prevent attackers from using junk data to make the autocomplete feature completely useless to users.

One rare exception is that attackers directly compromise the webpages of a vulnerable legitimate website to inject junk form data. In such a case, injected junk data may still be ranked higher than the form history data submitted by a user. However, compromising a legitimate website especially a high-security website is much more difficult than setting up a malicious website. Moreover, our mechanism can still successfully limit the influence of those junk data to the compromised website, and form autocomplete on other legitimate websites are still protected. On the other hand, we demonstrated in the last section that cross-site form history spamming attacks work in the absence of any vulnerability in websites; therefore, our mechanism is especially valuable in more general cases. As emphasized at the beginning of this section, the objective of our mechanism is not to replace browser vendors' efforts in securing the form autocomplete feature, but to complement those efforts to better protect this useful feature.

Even if spamming attacks do not happen and there is no junk data injected into the form history database, our domain-based ranking mechanism can still provide usability advantages over existing methods used in browsers. For example, a user may have visited a few other legitimate websites that have the "login" form field (Table 4), and may have multiple "login" IDs on each website. Even without injected junk data, current browsers cannot guarantee that the relevant "login" values on a legitimate website can be ranked high and directly suggested in the autocomplete dropdown list. This usability problem could be further aggravated considering that browsers such as Firefox and Google Chrome only allow at most six suggested values to appear at the same time in the autocomplete dropdown list. Integrating our domain-based ranking mechanism into the autocomplete feature of a browser can definitely help increase the relevancy of this feature's suggestions and improve its usability.

There is another usability consideration. In the current design, our domain-based ranking mechanism simply adjusts the ranking of all the selected values that have the matching field name. It moves the values of the domain-matching records ahead of the values of domain-mismatching records, but it does not remove those values of domain-mismatching records. We made such a design decision because sometimes users may want to reuse the values submitted from other domains. With such a design choice, injected junk data will still stay in the autocomplete dropdown list as shown in Table 5 and Table 6. If necessary, one potential enhancement is to further rank those values of domain-mismatching records. We leave this topic to our future research. One caveat is that if a user enters a typo on one website, the typo will be ranked above frequently used values entered on other websites. Enabling a user to easily remove a typo data record is desirable in general cases, and we expect browser vendors to provide such a user interface in the future.

4.3 Deployment Analysis

Our domain-based ranking mechanism can be easily deployed in popular browsers. First, the mechanism itself is very simple. It does not mandate underlying logic or algorithm changes to the autocomplete feature of browsers. Thus, the correct implementation of this mechanism can be ensured with high confidence.

Second, this mechanism is not bound to browsers' any specific support of DOM, event model, and JavaScript. We mentioned at the beginning of this section that some efforts have been made recently in Internet Explorer and Google Chrome to restrict the permissiveness of browsers in saving form history data. Normally, those efforts are quite browser-specific and they have to deal with the low-level details of how the form autocomplete feature is supported by browsers' specific DOM, event model, and JavaScript implementations. Our mechanism simply focuses on the upper-level form history data suggestion, so it can be easily adopted by different browsers to complement their existing protection mechanisms.

Finally, our mechanism is completely transparent to users. There is no need to train users, and there is no need to inform them of the deployment of this mechanism. Indeed, a user may not even notice the integration of our mechanism into a browser because nothing will be disabled by our mechanism.

5 Implementation and Evaluation

We have implemented our domain-based ranking mechanism in Firefox 3.6.10. The form autocomplete feature in Firefox 3 includes both C++ components and JavaScript components. The C++ components are responsible for the form filling control, form history database maintenance, as well as form data inserting, updating, and retrieving. The JavaScript components are responsible for preparing SQL search statements and ranking calculation. Overall, our modifications in the C++ and JavaScript components are within 180 lines of code. Our mechanism is easily implemented in Firefox, and we believe this simple mechanism can also be easily implemented in other modern browsers.

We have built the modified Firefox 3.6.10 on an openSUSE 11.3 (x86_64) Linux system and performed different types of experiments on it. We verified the correctness of our implementation and its integration with the browser. Basically, our implementation does not cause any runtime error or logic problem. We verified the effectiveness of our mechanism in ranking adjustment on over 30 various legitimate websites. Basically, the ranking adjustment step presented in the last section was properly executed; for example, all the values of the domain-matching records were moved ahead of all the values of domain-mismatching

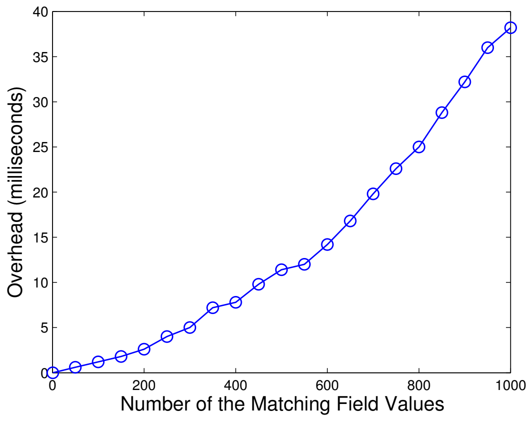


Fig. 8. Overhead of the ranking adjustment

records. We also used attack examples presented in Section 3 to inject a large amount of junk data and verified the usefulness of our protection mechanism. Basically, whenever a legitimate website was visited, junk data injected from an attack website were always ranked behind the data previously submitted on that legitimate website. We make our Firefox 3.6.10 openSUSE build available `http://www.cs.uccs.edu/~cyue/auto/ff3` for download and verification. Our source code is also available upon request.

We also measured the performance overhead of our mechanism in the Firefox 3.6.10 Linux build on a laptop with a 2.67GHz CPU. Our modifications made in the C++ components cause negligible overhead on data inserting, updating, and retrieving because they do not introduce new database operations. The ranking adjustment implemented in the JavaScript components can incur some overhead, but the overhead is very low. Figure 8 presents the ranking adjustment overhead (in milliseconds) versus the total number of the selected matching values, i.e., those values with the matching field name. We can see that the overhead increases approximately linearly with the increasing number of the matching values. The overhead is still less than 40 milliseconds even when the ranking of 1,000 matching values needs to be adjusted. These overhead results are averaged over five times of execution. If necessary, we can further reduce the overhead by implementing the ranking adjustment into an XPCOM (Cross-Platform Component Object Mode) C++ component in Firefox.

6 Related Work

In the BlackHat USA 2010 conference, Grossman [4] presented a few autocomplete and autofill vulnerabilities in popular browsers. One of the vulnerabilities reported in Grossman's talk is that attackers can automatically inject junk data into the form history database of a browser. Grossman described that automatically injected form data could be annoying, and perhaps browser vendors should fix that bug. However, we analyzed that injected junk data can make the useful autocomplete feature almost useless. Moreover, the problem is not just about fixing a bug in specific browsers; it is also about introducing new defense-in-depth mechanisms to the autocomplete feature. The *cross-site autocomplete* phenomenon is also what we need to pay attention to. Our domain-based ranking mechanism is a solution that can restrict the permissiveness of cross-site autocomplete, but still preserve its potential usability benefits.

Browser vendors are apparently taking active measures to protect against the cross-site form history spamming attacks. Just very recently, when we were finalizing this paper, we observed that Google Chrome 7.0.517.41 released on October 19, 2010 [12] has a security fix for its form autocomplete feature. Starting from this Google Chrome 7, the two malicious examples (attack1.htm and attack2.htm) presented in Section 3 are no longer successful. Security update for Internet Explorer released on October 12, 2010 also has a fix for the form autocomplete feature [13]. With that security update, Internet Explorer 8.0.7600.16385 disables its form autocomplete feature on https webpages. Firefox 3.6.11, 3.6.12, 3.6.13 were released on October 19, 2010, October 27, 2010, and December 09, 2010 respectively, but we verified that all the attacks presented in Section 3 can still be successful. The key point is that the *cross-site autocomplete* phenomenon still exists in the latest versions of the three most popular browsers. For Google Chrome and Firefox, this phenomenon exists on both http and https webpages. For Internet Explorer, it still exists on http webpages. So far, the security fixes made to these browsers focus on restricting the permissiveness of browsers in saving form history data. Our domain-based ranking mechanism complements browser vendors' efforts by focusing on restricting the permissiveness of browsers in suggesting candidate form history data. Such a domain-based design philosophy is also consistent with those of other efforts in securing Web browsers.

7 Conclusion

In this paper, we illustrated that modern browsers are generally too permissive in both saving form history data and suggesting them to users. A direct consequence of this permissiveness is that attackers can inject a large amount of junk or spam data into the form history database of a browser. These data may contain advertisement words used for getting financial or political benefits, or they are simply meaningless words used for nullifying the form autocomplete feature. We exemplified that both basic and advanced cross-site form history spamming attacks can be constructed by attackers. Browser vendors are apparently taking active measures to fix the security bugs in their browsers' form autocomplete feature, but we emphasized that they should also pay attention to the *cross-site autocomplete* phenomenon itself. Based on our analysis, we explored a different approach and proposed a domain-based ranking mechanism for protecting browsers from cross-site form history spamming attacks. Our mechanism restricts the permissiveness of cross-site autocomplete but still preserves its potential usability benefits. This mechanism is simple, transparent to users, and easily adoptable by different browsers as an in-depth defense to complement their existing protection mechanisms. We implemented our mechanism in Firefox 3.6.10 and verified its effectiveness. We believe that this simple mechanism can also be easily implemented in other popular browsers.

References

1. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. In: Proc. of the CCS, pp. 75–88 (2008)
2. Chen, S., Meseguer, J., Sasse, R., Wang, H.J., Wang, Y.-M.: A systematic approach to uncover security flaws in gui logic. In: Proc. of the IEEE Symposium on Security and Privacy, pp. 71–85 (2007)
3. Chen, S., Ross, D., Wang, Y.-M.: An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In: Proc. of the CCS (2007)
4. Grossman, J.: Breaking Browsers: Hacking Auto-Complete. In: Proc. of the BlackHat USA Technical Security Conference (2010), http://jeremiahgrossman.blogspot.com/2010/08/breaking-browsers-hacking-auto-complete.html
5. Jackson, C., Barth, A.: Beware of finer-grained origins. In: Proc. of the Web 2.0 Security and Privacy, W2SP (2008)
6. Jackson, C., Bortz, A., Boneh, D., Mitchell, J.C.: Protecting browser state from web privacy attacks. In: Proc. of the WWW, pp. 737–744 (2006)
7. Provos, N., Rajab, M.A., Mavrommatis, P.: Cybercrime 2.0: when the cloud turns dark. Commun. ACM 52(4), 42–47 (2009)
8. Reis, C., Barth, A., Pizano, C.: Browser security: lessons from google chrome. Commun. ACM 52(8), 45–49 (2009)
9. Reis, C., Dunagan, J., Wang, H.J., Dubrovsky, O., Esmeir, S.: Browsershield: vulnerability-driven filtering of dynamic html. In: Proc. of the OSDI (2006)
10. Ross, B., Jackson, C., Miyake, N., Boneh, D., Mitchell, J.C.: Stronger password authentication using browser extensions. In: Proc. of the USENIX Security Symposium, pp. 17–32 (2005)
11. Singh, K., Moshchuk, A., Wang, H.J., Lee, W.: On the incoherencies in web browser access control policies. In: Proc. of the IEEE Symposium on Security and Privacy, pp. 463–478 (2010)
12. Google Chrome Releases: Stable Channel Update, http://googlechromereleases.blogspot.com/2010/10/stable-channel-update.html
13. Microsoft Security Bulletin MS10-071 - Critical, http://www.microsoft.com/technet/security/bulletin/ms10-071.mspx
14. Same origin policy, http://en.wikipedia.org/wiki/Same_origin_policy
15. SQLite Home Page, http://www.sqlite.org
16. Submit form without reloading page, http://codingforums.com/showthread.php?t=61444.
17. Symantec Internet Security Threat Report Volume XV (April 2010), http://www.symantec.com/business/theme.jsp?themeid=threatreport