Protecting Users Against XSS-based Password Manager Abuse

Ben Stock
FAU Erlangen-Nuremberg
ben.stock@cs.fau.de

Martin Johns
SAP AG
martin.johns@sap.com

Abstract

To ease the burden of repeated password authentication on multiple sites, modern Web browsers provide password managers, which offer to automatically complete password fields on Web pages, after the password has been stored once. Unfortunately, these managers operate by simply inserting the clear-text password into the document's DOM, where it is accessible by JavaScript. Thus, a successful Cross-site Scripting attack can be leveraged by the attacker to read and leak password data which has been provided by the password manager. In this paper, we assess this potential threat through a thorough survey of the current password manager generation and observable characteristics of password fields in popular Web sites. Furthermore, we propose an alternative password manager design, which robustly prevents the identified attacks, while maintaining compatibility with the established functionality of the existing approaches.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access; H.4.3 [**Communications Applications**]: Information browsers

Keywords

Cross-site Scripting, XSS, Passwords, Password Managers, Countermeasure, Web Security

1. INTRODUCTION

In this section, we present the motivation behind our work and give a short outlook on the remainder of the paper.

1.1 Motivation

Passwords are the primary authentication method of the Web. With the growing set of Web applications that enter our life, the number of utilized passwords rises continuously. Security's best practices dictate, that each password should be sufficiently hard to guess and unique for the respective site, to limit dangers that arise from password reuse and guessing attacks. However, due to the traits of human nature, our capacities for remembering large sets of good passwords are limited, which in turn evidently leads to violation of the aforementioned password discipline [18, 13].

To provide assistance in this matter, password managers were introduced, that store passwords for the user and offer to (semi)automatically enter them into the matching password dialogues. Nowadays, all popular Web browser provide built-in password managers. Unfortunately, these managers operate by simply inserting the clear-text password into the document's DOM, where it is accessible by JavaScript. Thus, a successful Cross-site Scripting (XSS) attack can be leveraged by the attacker to read and leak password data which has been provided by the program.

Cross-Site Scripting is one of the most common security issues on the Web. The Open Web Application Security Project (OWASP) regularly lists XSS as one of the top three security vulnerability problems on the Web [24] and in its yearly analysis report [27], Whitehat Security lists XSS vulnerabilities documents that 43% of all discovered serious vulnerabilities can be accounted to XSS.

Thus, XSS-driven, automated password stealing attacks appear to be a realistic threat. In this paper, we assess the susceptibility of the current password (manager) landscape and propose a lightweight countermeasure, that robustly prevents the identified attacks, while maintaining compatibility with the established functionality of the existing approaches.

1.2 Contribution and Paper Outline

After providing the required technical background (Sec. 2), we make the following contributions:

- We give a comprehensive overview on potential XSS-based attack patterns on browser-provided password managers (Sec. 3) and explore potential mitigation strategies, that can be realized with the currently available technology (Sec. 4).
- We present two systematic studies: For one, we examine the current generation of existing password managers and show their susceptibility to the outlined attacks. Furthermore, we report on a large-scale study on the Alexa Top 4000 site, in which we studied how password fields are used by popular Web sites (Sec. 5).
- Motivated by the gained insights, that the vast majority of password scenarios are indeed vulnerable to the

ASIA CCS '14 June 03 - 06 2014, Kyoto, Japan

ACM ACM 978-1-4503-2800-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2590296.2590336 .

identified attacks, and that the currently available mitigation strategies are insufficient, we propose a client-side countermeasure, which robustly protects against XSS-based password theft without changing the general interaction scheme between user, Web page and password manager. Using a prototypical implementation as a Firefox extension, we practically evaluate our solution's security and functionality characteristics (Sec. 6).

We end the paper with a discussion of related work (Sec. 7) and a conclusion (Sec. 8).

2. TECHNICAL BACKGROUND

In the following, we give a brief technical background on password managers and the concepts of Cross-Site Scripting.

2.1 Password Managers

As studies [18, 13] have shown, users tend to choose bad passwords and/or reuse passwords over multiple sites, therefore undermining the security of their login credentials. To support users in employing a more secure password strategy, browser vendors as well as third party programmers have implemented password managers capable of storing these secret credentials for the users. This allows users to choose more complex and possibly random passwords by lifting the burden of remembering numerous complicated passwords. Hence, password managers can be beneficial for supporting better security practices in password handling.

Current implementations of password managers in browsers all work in a similar manner. Just before a form is submitted, the form is checked for password fields. If any such field exists, the username and password fields are determined and their values are subsequently extracted. These extracted credentials are then – along with the domain they were entered into – passed to the password manager. The password manager's database is subsequently checked for a matching entry, whereas no action is taken if the extracted credentials already match the stored ones. If, however, no matching entry is found, the user is prompted to approve storing of the password data. Analogously to that, if an entry for the same username but different password is found, the user is prompted to consent to updating the stored data. This process only works with forms that are submitted, either by the user clicking a submit button or by JavaScript invocation of the `submit()` method of that form. According to Mozilla [6], storing passwords which are sent using JavaScript XMLHttpRequests is not supported since no actual submission of the form takes place.

For each page containing a username and password field, the password manager is queried for entries matching the URL (or domain, depending on the implementation). If an entry is found, the fields on that page are automatically filled with the previously persisted credentials. Hence, the user then only has to submit the form to log into the application.

2.2 Cross-site Scripting

In the Web, access from a document to another document's content is governed by the Same-Origin Policy [2]. This policy makes sure that interaction between two documents may only occur if their origins match. An origin in this sense is the combination of protocol, domain and port of the interacting resources. Hence, an attacker, hosting his code on his own page, cannot directly access the content of, e.g., Google Mail even if the visitor of the page is logged in to that service. In the Web context, the term Cross-Site Scripting (*XSS*) is used for a class of attacks that allow an attacker to inject HTML or script code into a vulnerable Web application. This code is then executed in the context and, thus in the origin of the vulnerable application. In our example, this would mean that the code would be executed in the origin of Google Mail, thus allowing it to read arbitrary content from the site. Hence, Cross-Site Scripting can be seen as a way of circumventing the Same-Origin Policy. Cross-Site Scripting attacks are typically classified into three categories, namely *persistent*, *reflected* and *DOM-based* XSS. The first term adheres to the fact that the attackers code is persistently stored in an application's database, where in the second case, data provided to the application – e.g. via the URL – is reflected back into the response, allowing for malicious code to be executed. The third kind, which was first described by Amit Klein [15] in 2005 and – in contrast to the previously outlined attacks – abuses client-side code vulnerabilities. Although it was first discussed in 2005, DOM-based XSS still appears to be a major threat as Lekies et al. [16] recently showed by finding DOM-based XSS vulnerabilities on 9.6% of the Alexa top 5000 sites.

3. ATTACKS

In this section, we discuss the general attack pattern usable for stealing passwords via Cross-Site Scripting vulnerabilities and follow up with means of leveraging password managers to automate these kinds of pattacks. Afterwards, we give an overview of specific attack scenarios aiming to extract credentials from password managers and conclude the section with a related, active network-based attacker model.

3.1 Stealing passwords with XSS

Cross-Site Scripting gives an attacker ample opportunity to steal secret data from his victim. Typically, login forms for Web applications are realized using two input fields, which the user fills with his username and password, respectively. By design, JavaScript may interact with the document and thus is also capable of accessing the username and password field. This feature is often used by applications to verify that certain criteria are met – such as checking for e-mail addresses. However, this functionality also allows an attacker to retrieve the credentials utilizing Cross-Site Scripting. If the attacker can successfully inject his own JavaScript code into the login page, that code can extract the credentials entered by the user and subsequently leak them back to the attacker. This kind of vulnerability obviously only works if the user is not yet logged in when clicking on the crafted link. However, this is where password managers come to the aid of the attacker, as we will discuss in the following.

3.2 Leveraging Password Managers to Automate Attacks

Password managers provide a convenient way for users to automate parts of logins into Web applications. To make the login as simple and comfortable as possible, they automatically pre-fill forms for which the user stored the password beforehand. This feature can be exploited by a Cross-Site Scripting attacker. If a site is susceptible to XSS attacks,

the adversary can inject his own code into the application's login form in a similar manner as described earlier.

However, the attacker no longer needs to wait for the form filled by the user, since the password manager auto-fills the required fields. The attacker's code can then automatically retrieve the information and leak it back to the attacker. The biggest advantage in comparison to the aforementioned attack is the fact that the user does not need to be involved at all. This process can – depending on the browser – be fully automated in a hidden frame while the user is looking at a seemingly innocent page.

Figure 1 shows this process. First, when the login page is initially loaded, the fields are both empty. In the next step, the password manager automatically fills in the username and password, which can be subsequently retrieved by the code the attacker injected. The password is then automatically leaked back to the attacker as depicted in the lower part of the figure.

3.3 Specific Attack Patterns

In terms of Cross-Site Scripting, an attack targeting password managers specifically aims at extracting the stored user credentials in an automated way. The attacker therefore tries to embed a form into a vulnerable application which is then filled in by the password manager. This form can afterwards be read by the attacker's JavaScript code to retrieve the data that was inserted by the victim's browser and eventually leak the information back to the attacker.

In our study of current password manager implementation, which we will discuss in further detail in Section 5.1, we found that their behavior could be distinguished in four dimensions. In the following, we will discuss the discriminating factors and the attack patterns associated with them.

Matching requirements for the URL and form.

The first factor we examined was the way in which password managers react to changes both to the URL and the form itself. Password managers often fill passwords regardless of the context, as long as the domain matches, and, potentially, other easily fabricated indicators such as field names and types or form action are present.

When a password manager does not explicitly store the complete form the credentials were stored for, but rather only the origin, an attacker can easily extract the credentials. To achieve this, he can abuse a Cross-Site Scripting vulnerability on an arbitrary part of the application to inject his form and corresponding JavaScript code. This form is then filled by the password manager and the stolen data can be sent to the attacker. In cases where a password manager also does not store the names of the fields the data was stored from, the attack is even easier since an attacker does not need to craft a form specifically mimicking the login page of the target application, but may use a generic form. This allows him to automate the attack for multiple vulnerable pages in a very simple and almost effortless manner.

Viewports.

If a password manager explicitly checks the URL rather than the origin, the attacker has to force the victim's browser to load the original login page to make the password manager fill out all the relevant fields. Hence, the second criterion we found is the difference in handling viewports. In our notion, a viewport can either be a top frame, a sub frame or a popup window. With respect to that, the interesting question is whether a password manager still fills out forms if they are not located in the top frame of a page.

If login field data is inserted regardless of the viewport, an adversary can place a hidden frame pointing to the login page on the vulnerable page. As enforced by the Same-Origin Policy, any page may only access another document's content and resources if the protocol, the domain and the port of both involved documents match. As we assume the attacker has control over some page inside the vulnerable Web application, he can therefore access the aforementioned frame's content, thus enabling extraction of the credentials that were filled in by the password manager. If a password manager does not automatically fill in the values of interest to the attacker, or the application itself forces not to be framed using the `X-Frame-Options` header [19], the login page can be opened in a popup window. Still operating under the assumption that vulnerable and login page are of the same origin, the attacker's code can retrieve the data from the opened popup.

User interaction.

As a third distinguishing feature of the examined password managers, we identified user interaction – i.e. whether the user has to somehow interact with the password manager before it fills out the forms, e.g., via clicking or typing into the password field. If a given password manager requires such interaction, fully automated XSS password stealing attacks are not feasible.

However, in such cases, an attacker can attempt to conduct a ClickJacking [26] attack. ClickJacking attacks work by tricking the user to interact with a security sensitive Web UI without his knowledge. In the general attack case, the adversary loads the document which contains the security sensitive UI into an iframe and hides the frame from the user's eyes via CSS properties, such as `opacity`. Subsequently, he overlays the targeted (and now invisible) UI elements with unsuspicious elements and motivates the user to click them, for instance in the context of a game or a competition. If the user falls for the adversary's bait, he involuntary interacts with the hidden UI.

Using this attack, the adversary can trick the victim to interact with the password field in the required fashion, thus, causing the password manager to fill the field with the stored value.

Adherence to the autocomplete attribute.

The fourth and last dimension we found was the adherence to the `autocomplete` attribute for fields. According to the W3C standard [12], a browser must not store data that is inserted into input fields which have `autocomplete` set to `off`.

From the attacker's point of view, this feature is very interesting. If a password manager does not respect the `autocomplete` value when *storing* the credentials but only when later *filling out* the input fields, it is still susceptible to attacks. In order to extract password data from clients, the adversary can simply add a second form with the same names and input types to the document, this time without the `autocomplete` attribute, which is then filled with the persisted credentials.

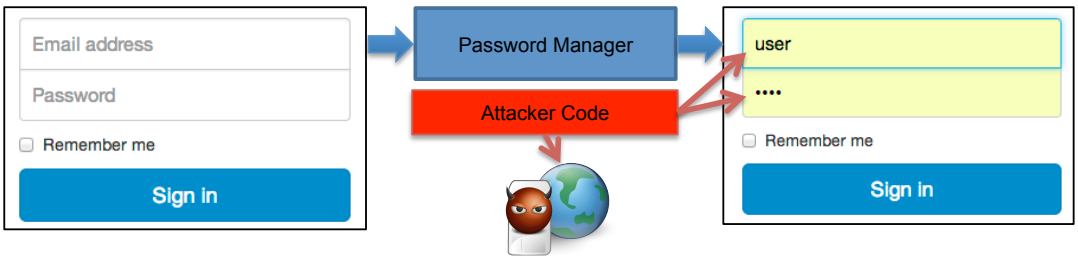

Figure 1: Process of leveraging the password manager to steal stored credentials

In Section 5, we will discuss in detail how the browsers that we examined behaved with respect to the aforementioned four dimensions.

3.4 Network-based attacks

In concurrent and independent work, Gonzalez et al. [9] discovered a related attack. However, in contrast to our attacker model, their attacker is located in the network and can thus introduce content into every site of his choosing. They developed a tool called Lupin which automatically injects iframes for each site they want to steal credentials from and then extracts the inserted data. However, if we assume that the attacker cannot produce a valid TLS certificate for any domain he wants to extract data from, this attacker can only steal login data from sites that do not use HTTPS to serve their login form. Thus, this attack scenario can be thwarted if all pages use strict HTTPS for both the login form and the URL the data is then sent to. Although this attacker model is not the focus of our work, we evaluate the suspecibility of current Web applications to these attacks in Section 5.

4. MITIGATION STRATEGIES

Before presenting our solution approach in Section 6, we briefly discuss existing remedies and potential techniques, which could be applied by application providers and end users.

4.1 Server-side Mitigation

Several approaches exist, which could be adopted by Web application providers to protect their users against the presented attacks. However, as we will discuss, each of these techniques comes with certain drawbacks.

Switching to alternative authentication methods.

Instead of relying on form-based authentication, which is susceptible to XSS attacks due to the fact that injected JavaScript can access the value of the password field, Web application providers could switch to authentication processes that utilize credentials which are out of the adversary's reach:

- *HTTP authentication:* HTML forms are not the only available user interface component for password entry offered by Web browsers. If a Web application's server side requires HTTP authentication, signaled through a 401 HTTP response [7], the user is presented with a dedicated authentication dialog, which is realized with UI components that are outside of the current document's DOM, and hence, out of reach for a JavaScript attacker. Unfortunately, HTTP authentication is cumbersome to integrate the general workflow of Web applications: Form data is *pushed* by the browser to the server on form submit, while HTTP authentication is *pulled*, after the server notices an attempt to access restricted information. Mixing the two paradigms potentially leads to incoherent and confusing end user interfaces.
- *Client-side SSL authentication:* Instead of using passwords, the application could switch to public key authentication. Authenticating via client-side SSL certificates [5] is well supported by the browser's SSL/TLS implementation. However, such certificates have significant disadvantages over passwords in respect to deployment (the application needs to outfit all users with valid certificates), usability (handling browser-based certificates is well out of the expertise of the average Web end user), and portability (the certificates are enrolled in the browser, switching browsers or computers requires certificate migration).

Moving password forms to a dedicated sub domain.

Furthermore, to mitigate the outlined threat, sites could leverage the Same-Origin Policy's protection capabilities. If all password handling forms, and only these forms, are hosted on a dedicated (sub)domain, XSS vulnerabilities of the main application won't compromise the password's security. The injected JavaScript is executed under an origin that differs from the origin of the password-hosting document, hence, the JavaScript has no access to the field. Furthermore, the password manager stores the password for the domain value of the dedicated (sub)domain, thus, injecting further password fields into the document of the injected JavaScript has no effect, as no password has been stored for this document's origin in the first place. For apparent reasons, choosing this path comes with the price of elevated setup, deployment, and maintenance costs for the application provider (e.g., valid SSL certificates for the subdomain need to be acquired).

Disabling the password manager.

As covered earlier, using the HTML attribute `autocomplete="off"` [12], a Web application can prevent the browser from storing the password in the first place. The reasoning behind this is that the data, which is entered into fields which have this attribute set to `off`, is particularly sensitive and should therefore neither be stored nor inserted later. While being secure against the outlined attacks, this technique causes the loss of the usability advantage of the

password manager, which in turn (see Sec. 2.1) could lead to situations in which end users potentially choose less secure passwords.

4.2 End-user Protection

Currently, end-users have two options to protect themselves against XSS password theft:

For one, they can simply turn their browser's password manager off through switching the corresponding setting in the browser's configuration [21]. While this is certainly the safest choice, the user looses both the functionality gain and the potential security advantage (see Sec. 2.1) of using a password manager.

The second option is to switch to a third-party password manager that requires explicit user interaction. If explicit user interaction, such as clicking a button, is a prerequisite for the system to fill the password value, fully automatic XSS-driven attacks are thwarted. Please note: It is crucial to mention, that the UI component which triggers the required user interaction is positioned outside of the DOM of the attacked page. Otherwise, the adversary can resort to the ClickJacking attack variant (see Sec. 3.3) and undermine the provided mitigation.

The primary focus of most third-party password managers is on security and not on ease-of-use or transparency. For instance, in it's default configuration, 1Password[1] requires the pressing of a predefined key combination, followed with the entry of the tool's master password. Hence, they might be not a viable option for end users that utilize password managers foremost as a convenience feature.

5. EXPLORING THE PASSWORD (MANAGER) LANDSCAPE

As explained above, several potential XSS attack patterns on password managers exist. To examine the degree to which these theoretic attacks are applicable with the currently deployed password managers and Web sites, we conducted two comprehensive studies. For one, we systematically examined the built-in password managers of the current browser generation (see Sec. 5.1). Furthermore, we conducted a large scale study on how password fields are used by existing applications (see Sec. 5.2).

5.1 Password managers

In this section we present the results of our experiments on the behavior of different modern browsers. Our tests were aimed in the four different dimensions previously discussed in Section 3.3.

To ensure a broad coverage of internet users, we opted to examine Google Chrome (version 31), Mozilla Firefox (version 25), Opera (version 18), Safari (version 7), Internet Explorer (version 11) and the Maxthon Cloud Browser (version 3). Although the latter one might not be as well-known as the other candidates, it is one of the options that is shown to users installing the latest Windows versions. Hence, we looked at the behavior of this browser along with the previously named.

Before investigating the behavioral changes when tampering with the form or the URLs the form was located in, we first analyzed the general fill-in behavior of our test subjects according to the specific attacks discussed in Section 3.3.

[1]1Password: `https://agilebits.com/onepassword`

Filling only in the top frame.

To assess whether password managers would fill out forms only in top frames, we created a page that framed the original, unchanged login page we had initially stored our credentials for. Apart from Internet Explorer, which refused to insert any data, all browsers filled in the username and password field.

Explicit user interaction.

Next, we investigated whether a browser would actually need any form of interaction from the user to fill in passwords. Again, Internet Explorer was the (albeit positive) outlier, being the only browsing engine that required any form of interaction. In Internet Explorer, the user has to manually put the focus to the username field and is then presented with a dropdown menu allowing him to select which credentials he wants to insert. The user then has explicitly click on an entry to trigger the browsers fill-in action. Also, this is done properly outside of the DOM, thus the ClickJacking attacker discussed in Section 3.3 can also not force the filling of password fields.

URL matching.

We assume that the attacker wants to steal the credentials from his victim in a stealthy manner. We consider the following example: an application hosts its login at `/login`. The attacker has found a XSS vulnerability at `/otherpage` which he wants to abuse to steal the stored credentials. Hence, if a password manager only supplies the password to the exact URL it stored the passwords for, the attacker would have to open a popup window or embed a frame to the login page to steal the secret data. However, opening a popup window is very suspicious and therefore not desirable. Also, framing the login page in an invisible frame might not work due to `X-Frame-Options` headers. In our study, which we discuss in Section 6.3, we found that only 8.9% of login pages make use of this header to ensure that they are not framed. Thus, in our work, we wanted to determine how easy it was to make password managers fill in the stored credentials into forms if the URL did not match the one the password was originally stored for. To examine the browsers' behaviours, we created a simple Web application with a login form. We visited this login and let the password manager under investigation save the credentials that we entered. We then created multiple other pages running under different protocol (HTTP vs. HTTPS), different ports, different (sub-)domains as well as changing paths to determine what the implemented matching criteria were for all our test subjects. In the following, we discuss the results of the analysis of the aforementioned browsers.

- *Google Chrome:* Our tests showed that changing the protocol, sub domain or port lead to the password to not be filled in anymore. In contrast, when visiting a form running under a different path, Chrome still inserted the stored credentials. This leads us to reason that Chrome stores the password alongside their origin in the sense of the Same-Origin Policy, namely the triple protocol, domain and port.
- Our second candidate was *Firefox*. Similar to the behaviour Chrome exhibited, Firefox also refused to fill out login fields if either protocol, (sub-)domain or port were changed. It also behaved in a similar manner to

Chrome with respect to a change in the path – still automatically setting the username and password fields to the stored values.

- Both, *Opera and Safari* behaved in a similar manner. With changed origins, they refused to fill out forms, whereas the path was not taken into consideration in the decision whether to insert the stored credentials or not.

- *Internet Explorer:* In contrast to all the aforementioned, Microsoft's Internet Explorer apparently stores the complete URL of the form it saved the password data for. In our tests, it showed to be the only browser that did not insert stored credentials even if only the path changed.

- *Maxthon Cloud Browser:* Alongside all the well-known browsers we examined thus far, we also looked at the password manager of the Maxthon Cloud Browser. Most interestingly, the passwords were apparently only stored coupled with the second-level domain they stemmed from. In our tests, the browser would still fill in password fields even if the protocol, sub domain, port or path changed.

Summarizing, our tests showed that out of the most commonly used browsers on the Web, all but Internet Explorer gladly fill forms on any part of the same Web application, whereas the application borders are determined by the Same-Origin Policy. The Maxthon Cloud Browser even fills in credentials if only the same second-level domain is visited – ignoring both the protocol and the port of resource – making it even easier for an attacker to extract the passwords from its storage.

Form matching.

After having examined how browsers treat changes in the URL with respect to their password managers, we analyzed what kind of information on the actual form browsers would store. To gain insight into this, we built another set of test pages – this time with different modifications to the login form itself. Our test pages were different from the original form in different aspects, which we discuss briefly in the following.

For the first test case, we removed the action and the method of the form. Our second modification was the removal of the names of all fields in the form, whereas the third change was to only change the names of all fields rather than removing them. For the next part of our analysis, we removed the types from all fields, essentially resetting them all to `type=text`. We then derived a minimal form as shown in Listing 1, only consisting of two input fields with random names, no action or method as well as no additional submit buttons. After these changes to the form fields, we build a final testing page, setting the `autocomplete` attribute for the *password* field to off. According to the W3C specification [12], this value indicates the browser should neither store the data inserted into that field nor automatically fill it in later.

Utilizing these, we now discuss the matching criteria with respect to the structure of the form presented to the password manager.

- *Google Chrome:* We observed that neither action nor method of the form were criteria in the decision, whereas the same held true for changes to the names of the fields we provided. However, if we presented Chrome with fields without any name, it would not provide the credentials to the form. Chrome did not strictly adhere to the autocomplete setting of the password field, prompting the user to save the password nonetheless. It did however adhere to the setting when inserting the password into the form – nevertheless, we could extract secret data by adding a second form, as described in Section 3.3. Since the matching is done on a structural basis, the minimal form shown in Listing 1 was sufficient for this attack.

- *Firefox* also only performed matching against a form's structure, not the content itself. In contrast to what we had seen with Chrome, Firefox did however also insert credentials into forms that only contained input fields without names. Also unlike Chrome, Firefox adhered to the `autocomplete` attribute – if either field had this set to off, Firefox would not store any data. Due to these factors, injecting the minimal form would still trigger the auto-fill functionality of Firefox's password manager.

- *Opera and Safari* again behaved alike, filling in passwords into the minimal form but not into forms containing only input fields without names. On our test machine, a Macbook running OS X Mavericks, we discovered that both Opera and Safari also use the OS X keychain to store their passwords. Thus, after having stored a password in Opera, Safari automatically filled out our fields although we had not previously stored a password in its database. While Opera – similar to Chrome – also offered to store passwords if at least one of the posted fields did not have `autocomplete` set to off, Safari behaved like Firefox and did not save any data in that case. Again, the test subjects only performed structural rather than content matching, leading to both of them also auto-filling the minimal form. Contrary to Firefox, both browsers would not fill input fields without names.

- *Internet Explorer:* As explained in Section 5.1, Internet Explorer was the only browser that required any form of user interaction to fill in the passwords. To nevertheless check the functionality, we manually interacted with the browser to ensure that it would fill in the username and password. In that, we discovered that Internet Explorer applies matching criteria in the same manner as Firefox, namely inserting passwords even into forms containing only input fields without any name. In terms of adhering to the autocomplete attribute, Internet Explorer did respect the value by not saving any information if either field had the autocomplete value set to off.

- *Maxthon Cloud Browser:* Not unlike the insecure behaviour it showed regarding matching the URL, the Maxthon Cloud Browser was not at all strict in matching the form, even filling in input fields that had no name and – most notably – that had autocomplete set to off.

To sum up: Most browsers are very relaxed in terms of matching criteria. All but Internet Explorer would still fill in passwords if only the origins matched, whereas the Maxthon Cloud Browser even only took the second-level domain into consideration for its decision. Similar to that, matching against a form was mostly performed on a structural level, i.e. meaning that any two fields were filled out if the latter was a password. According to Mozilla[6], this is done by design as a convenience feature. Looking at the results, the tests with different forms showed that the attacker only has to create a minimal form as shown in Listing 1 to trick the browser's password managers into providing the stored passwords from any site that uses two input fields for its login dialogue.

All the previously discussed results are depicted in Table 1, whereas *'Yes'* denotes that the criterion must match. For the minimal form, *'Yes'* denotes that the minimal form was sufficient, whereas *'Yes'* for autocomplete means that the browsers would not save passwords if the autocomplete attribute was set to off.

Listing 1: Minimal HTML form used in our tests

```
<form>
<input name="random1">
<input name="random2" type="password">
</form>
```

5.2 Password fields

To obtain a realistic picture on how password fields are currently used in practice and to which degree real-world password dialogs are susceptible to the attacks discussed in Section 3, we conducted a survey on the password fields of the top ranked Web sites according to the Alexa index [1].

5.2.1 Methodology

To locate and analyze password fields in real-world Web sites, we conducted a lightweight crawl of the top 4000 Alexa sites. As many modern sites rely on client-side markup creation and DOM manipulation via JavaScript, we chose a full fledged browser engine as the technical foundation of our crawling infrastructure: We implemented an extension for the Chrome browser, that pulls starting URLs from a backend component, which are subsequently visited by the browser. This way, we not only can examine the same final DOM structure that is also presented to the browser, it also gives us the opportunity to observe client-side actions after a password has been entered (more on this below). Our Chrome extension consists of the following JavaScript components [10]:

- A single *background script*, which is able to monitor network traffic and distribute the crawling process over multiple browser tabs,
- and multiple *content script* instances, one for each Web document that is rendered by the browser. A content script is instantiated as soon as a new document is created by the browser engine. This script has direct access to this document's DOM tree. However, the script's execution context is strictly isolated from the scripts running in the document.
- Thus, the content script injects a *user script* directly into the documents' DOM. Unlike the content script, which is cleanly separated from the document's script environment, the user script runs directly in the same global context as the document's own script content. This in turn grants us the ability to wrap and intercept native JavaScript functions [17], such as `XMLHttpRequest` or getter/setter properties of HTML objects.

Criteria	# Sites	% rel.	% abs.
Password found	2143	100 %	53,6 %
PW on HTTPS page	821	38,31 %	20,5 %
Secure action[1]	1197	55,9 %	29,9 %
Autocomplete off	293	13,6 %	7,3 %
X-Frame-Options	189	8,9 %	4,7 %
JavaScript access	325	15,1 %	8,1 %

[1]: Password form submitted to an HTTPS URL

Table 2: Recorded characteristics of the Alex Top 4K password fields

Using this infrastructure, our extension conducted the following steps:

The homepage URL of the next examination candidate is pulled from the backend and loaded into one of the browsers tabs. After the rendering process has terminated, the DOM tree is traversed to find password fields. However, most sites do not immediately contain the login dialog (if they have one at all) on their homepages. Instead, it is usually contained in a dedicated subpage, linked from the homepage. Hence, in case no password field could be found on the homepage, all hyperlinks on this page are examined, if they contain indicators that the linked subpage leads to the site's login functionality. This is done via a list of indicative keywords, consisting of e.g., "sign in", "login", or "logon". If such a link was found, the browser tab is directed to the corresponding URL and this document is examined for password fields. While this methodology is apparently incomplete, e.g., due to the keyword list only containing terms derived from the English language, turned out to be sufficient to find a representative number of password fields (see Sec. 5.2).

If a password field was found, important characteristics of the document were recorded, including the hosting document's URL, the corresponding HTML form's `action` attribute, as well as the presence of `autocomplete` attributes and `X-Frame-Option` headers.

Furthermore, to observe potential client-side processing after a password has been entered, we instrumented the `get`-property of the password field using JavaScript's `Object.defineProperty` API [22], after the page's rendering process has terminated, but before the page's own scripts executed.

Subsequently, after the page's scripts have been run, the user script simulates user interaction with the password field to potentially activate JavaScripts that access the password value legitimately. More precisely, our script triggers JavaScript events, that would occur if a user clicks into the field, changes its values, and leaves the password field, i.e., moves the focus to a different field. Finally, the script submits the form, in order to activate any JavaScript that is tied to the `onsubmit` event.

5.2.2 Results

During our crawl, we could successfully detect a login form on 2,143 of the 4,000 domains. In the following, we will

	Port	Path	sub domain	any name req.	name match	input type match	min form	autocomp.
Chrome 31	Yes	No	Yes	Yes	No	Yes	Yes	No
Internet Expl. 11	Yes	Yes	Yes	No	No	Yes	Yes	Yes
Firefox 25	Yes	No	Yes	No	No	Yes	Yes	Yes
Opera 18	Yes	No	Yes	Yes	No	Yes	Yes	No
Safari 7	Yes	No	Yes	Yes	No	Yes	Yes	Yes
Maxthon 3	No	No	No	No	No	Yes	Yes	No

Table 1: Overview of tested browsers and their matching criteria

outline the analysis of the data we gathered from these fields with respect to different, security-relevant questions.

As discussed in Section 4.1, the use of the `autocomplete` attribute on input fields allows an application to ensure that no data is persisted into the password manager's storage. We therefore investigated how often this was explicitly set to `off`, essentially instructing the browser to neither store login data nor auto-filling these forms. Out of the 2,143 domains we examined, a total of 293 domains prohibited password managers from storing the credentials this way.

In respect to the ClickJacking attack on a password manager that requires user interaction (see Sec 3.3), the applicable remedy is the usage of the HTTP's `X-Frame-Options` response header [26]. By using this header an application can explicitly tell the browser that a page may not be rendered inside a frame. However, this only helps against the discussed attacks if the header is set to `DENY`, since we must assume that the XSS attacker is capable of positioning an iframe containing the login form on a page located in the same application, thus running under the same origin. In our investigation, we found that only 189 domains set the header to `DENY`, while another 173 had set it to the `SAMEORIGIN`, which is useless in the context of the discussed attacks.

Furthermore, to gain insight on the extent of legitimate client-side functionality that uses JavaScript to read password fields, we instrumented the password field, such that we were able to record read access (see above). For a total of 325 password fields, we were able to witness read operations via JavaScript.

Finally, we examined to which degree the sites were potentially susceptible to network attackers. To do so, we checked how many forms containing password fields are delivered via plain HTTP rather than HTTPS. In the attacks scenario described by Gonzalez et al. [9], enabling HTTPS effectively blocks their attacker. In our study, we found that only 821 domains utilize HTTPS when transmitting the password field itself. The remaining 1,289 domains are hence susceptible to the network-based attacker who directly inserts JavaScript into the server's response to retrieve the password data from the victim.

Additionally, a network-based attacker may also retrieve passwords from users once they log in to an application if the credentials are sent to the server using HTTP and not HTTPS. Investigating how many applications send out secret login data in an unencrypted manner, we found that in total, 1197 sites used HTTPS to send the password data to the server, leaving 946 sites with unencrypted communication.

5.3 Assessment

As shown in Section 5.1, most browsers only store the origin of a password and not the complete URL of the form it was initially stored from. Thus, placing a form on an arbitrary page with the same origin as the login form is sufficient to extract credentials from the victim.

The Cross-Site Scripting attacker, which we discussed in the previous sections, is capable of injecting his malicious payload into applications that are delivered via HTTP as well as over HTTPS. Thus, the only line of defense in this case is the `autocomplete` feature. As discussed earlier, this is only used in 293 login pages, thus resulting in a total number of 1,850 out of 2,143 domains which are potentially vulnerable to password stealing by an XSS attacker. This amounts to a total of 86.3% of analyzed pages which are susceptible to the attack scenario we outlined. Apart from Microsoft's Internet Explorer, the built-in password managers of all browsers we examined automatically filled out forms presented to them and would also behave in the same manner if the login page was put into a frame. In order to successfully conduct an attack on Internet Explorer, the attacker would have to have found a vulnerability on the exact login page and would also have to rely on the victim actively selecting the credentials to insert.

The network-based attacker, who is only capable of injecting his malicious payload into login pages which are not served using HTTPS, can only successfully attack 1,029 different domains, summing up to 48% of all applications we analyzed.

These observations lead us to the conclusion that the current implementation of browsers' password managers is highly vulnerable with respect to password stealing – both by a network and a XSS attacker. Also, the server-side measures we discussed in Section 4.1 are not employed in prevailing web applications in a satisfactory manner. Therefore, in the following section, we will discuss a new approach to the concept and implementation of a password manager capable of tackling these issues.

6. CLIENT-SIDE PROTECTION

Our analysis has shown that popular browsers implement password managers in a way that is susceptible to Cross-Site Scripting attacks. We have shown that most of the browsers neither save information on the URL the password was initially stored for nor do they require user interaction to fill out forms. This allows the attacker to retrieve passwords in the presence of an XSS vulnerability. In the following, we propose a simple yet effective solution to counter these types of attacks on password managers.

6.1 Concept

The common enabling factor of the documented attack types (see Sec. 3) is the fact that the secret data is directly inserted into the forms when the page is loaded, and can subsequently be retrieved by JavaScript.

The underlying problem is that concept and implementation of password managers are not aligned. Abstracting

what a password manager's task is, we see that it should aid users in the login process to Web applications. This process can be seen as the credentials being sent to the login page. The implementation of that paradigm however aims at filling out forms before the actual, clear-text login data is required. A XSS attacker aims specifically at this conceptual flaw, extracting the credentials from the auto-filled form. In our notion, a password manager should ensure that only once the login data is sent to the server, the plain-text password is contained in the request. Hence, in the following, the propose an enhanced password manager which tackles this conceptual flaw.

Our proposal is that a password manager should only insert placeholding nonces into a form. Once the user then submits the form towards the application, the password manager replaces the nonce with the original, clear-text password. Thus, if an attacker can extract the content of this form utilizing a XSS vulnerability, he is nevertheless unable to retrieve the real password of the targeted user.

Furthermore, our mechanism requires strict matching of the password field `name` attribute and the corresponding POST value. For better understanding of the rationale behind this, consider the following scenario: The attacker is able to inject a new field called `query` into the form. Once the password manager has filled in the placeholder into the `password` field, the attacker's code copies the value of that field into the newly added `query` field. He then changes the action of the form to the application's search functionality. If the password manager now replaced all occurrences of the placeholder in the request, the `query` parameter would also contain the clear-text password. Under the assumption that a search page will in some manner reflect the search term back to the user, the attacker could then extract the password from this response. Therefore, making sure that the password manager only exchanges the correct field is essential.

6.2 Implementation

To investigate the soundness of our proposal, we implemented a proof-of-concept password manager. Since completely changing the implementation of one of the built-in password managers in the modern browsers would have been to complex, we opted to instead build an extension for Firefox. The extension is built upon the original, built-in password manager which is used to only store the placeholder values. The clear-text passwords in turn are stored in a separate storage located inside the extension. For our prototype, we did not implement any form of encryption for these values, since securely storing the passwords inside the browser is out of scope for this work.

Figure 2 shows how our approach works when a new username and password combination is saved. First, the user is prompted to have the login manager remember the recently sent password. In the second step, once the user has agreed to do so, the login manager stores the username and password combination. Firefox provides all plugins with a means of being notified when credentials are stored in the password manager [20]. The notification message contains – along with the recently stored username and password – the origin of the site the password was posted to as well as the names for both the username and password field in the submitted form. Our extension saves all this information in its own storage and replaces the password in the built-in storage with a random placeholder value (nonce). This placeholder value is subsequently also stored inside the extension's database alongside the previously persisted data to ensure that the matching credentials can be extracted later on.

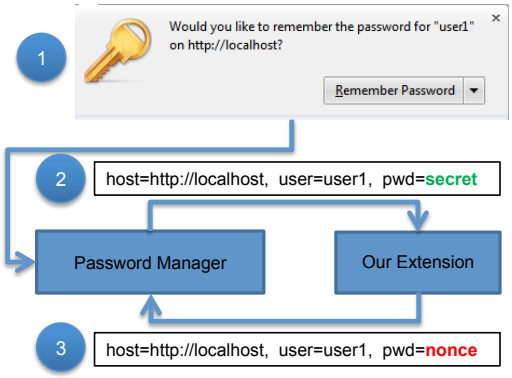

Figure 2: Initial login and credential storing

Figure 3 outlines how the placeholder is later restored in a normal login. When opening the login page, the built-in password manager inserts the username and the placeholder into the form. Similar to the internal password manager, the extension is notified of a password form being submitted. [6] Subsequently, the next outgoing POST request is scanned by our extension for the easily discernible placeholder value. If the nonce is found, the extension searches its own database for the corresponding entry. Next, the entry's origin is checked against the origin of the page the data is being sent to. If the origins match, the placeholder is replaced with the actual passwords adhering to the aforementioned constraint that only the password field (whose name is stored in the extension's data storage) should be changed. On the lower half of the figure, this is shown. Although the nonce is contained in the HTTP request, it is not replaced with the actual secret data since the name of the POST parameter does not match. Thus, the attacker cannot utilize the search functionality to extract the secret password data from our password manager.

We also evaluated the option of exchanging GET parameters in a request. In our empirical study we found that none of the sites use a form in combination with a GET request. An attacker could however easily exchange the method of a form from POST to GET. If our proposed password manager would then exchange the nonce with the secret password, the adversary could easily read the complete URL of the newly loaded page and thus retrieve the password. Therefore, we explicitly disable the replacement of our nonces in GET parameters and only exchange them for the real credentials in POST requests.

6.3 Evaluation

In this section, we discuss both the security and the functional evaluation of our approach, showing that it adds security while not causing incompatibility with existing applications.

6.3.1 Security Evaluation

After the password value has initially been stored by the password manager, it is never again inserted into Web docu-

Figure 3: Replacement of login credentials by our enhanced password manager

ments. Hence, it is kept out of reach of potentially malicious JavaScript.

Furthermore, our implementation enforces strict matching constraints, before the replacement process executes: Only password nonces for which the combination of *target origin* and *password parameter name* matches the recorded values are substituted with the actual password value in the outgoing request. This requirement effectively thwarts attack attempts in which the adversary tries to leak the password via tampering with the password field's form element in the timespan between the autofill process and the form submission. Thus, our proposed implementation of a secure password manager effectively hinders an attacker who utilizes XSS attacks against his victim.

However, the attacker model discussed by Gonzalez et al. [9] – positioned at the network layer – could still be successful if password data is transmitted in clear-text to the server. In our study of the Alexa Top 4000 sites, we found that 44,1% of the examined sites utilize HTTP instead of HTTPS to transmit the credentials to the server. In these cases, the network-based attacker could wait for the form to be submitted and subsequently retrieve the secret login data from the traffic capture. Nevertheless, this kind of attack does not specifically target password managers and can therefore not be fully prevented by a secure password manager in any case.

6.3.2 *Functional Evaluation*

From the user's point of view, nothing changes compared to the behavior of the current generation of deployed password managers: After page load, the password field is automatically filled with characters, which are presented to the user with masquerading asterisks. After form submit, the browser exchanges the password nonce with the actual values, before it is sent to the server.

Our approach aims at only putting the real password of a user in the outgoing request to the server and not into the password field. This however leads to potential problems with Web applications that do some transformation on the original field's value before submitting it. For instance, an application might derive the hash sum of the user-provided password on the client-side before submitting it.

In the evaluation of the top 4000 Alexa sites, we detected 325 JavaScript accesses to password data (see Sec. 5.2). We then manually analyzed the snippets responsible for these accesses and detected that a total 96 domains used client-side functionality such as XmlHttpRequests to transmit the password data to the server. Out of these 96 cases, 24 pages transformed the provided password before forwarding it to the server, whereas 23 employed hashing functions like MD5 and SHA1 and the remaining case encoded the password as Base64. Of the remaining 72 pages that did not post the form directly to server, only 6 pages employed HTTP GET requests to transmit the credentials, whereas the rest used HTTP POST in their XmlHttpRequests. Our proposed approach would obviously not work in these 30 cases, since our extension neither exchanges passwords directly in the input field nor does it modify HTTP GET requests. However, the current implementations of the password managers do not support storing passwords that are not sent via submitting HTML forms – thus our approach is in no way inferior to the currently deployed concepts [6]. Also, if the built-in password manager stored these credentials, there is no way of detecting whether access to a given password field is done by the legitimate page or is a Cross-Site Scripting attack. Hence, we deliberately fail in the aforementioned scenario by not replacing the nonce in the input field with the real password. Therefore, our approach is secure by default and can also not be undermined by an unknowing user.

The purpose of the remaining 229 scripts was verify was that certain criteria had been met in filling the user and password field, e.g. the username being an e-mail address or the password consisting of at least a certain amount of chars.

7. RELATED WORK

Most research in the area of password managers focussed mainly on three different aspects: generating pseudo-random and unique passwords for each single Web application based one some master secret [11, 25, 4], storing passwords in a secure manner [31, 3, 14, 8] and protecting users from phishing attacks [29, 30].

The problem of weak password manager implementations with respect to their vulnerability towards Cross-Site Scripting attacks has been discussed by browser vendors since 2006 [23]. However, researchers did not re-evaluate possibilities in terms of adopting new concepts to protect users from these kinds of attacks. In a recent blog post, Ben Toews again brought up the issue of password managers that were prone to XSS attacks [28]. However, the question on how to improve the security of password managers remained unanswered.

Furthermore, in 2013, Gonzalez et al. [9] discovered a related attack. They described a network-based attacker that can inject code of his own choosing into any unencrypted HTTP connection. To leverage this, they injected multiple invisible frames into pages loaded by the victim and iterated through the login pages of different domains. They automated their attack using a self-developed tool called Lupin and were able to extract 1000 passwords in 35 seconds from a victim's machine. They propose a first set of countermeasures, which are either in line with the mitigation strategies covered in Sec. 4, and thus, share the same drawbacks, or

are targeted at strict network layer security, which is not applicable to our XSS attacker.

8. CONCLUSION

In this paper, we have demonstrated that current implementations of built-in password managers in browsers are vulnerable to XSS attacks targeting the stored passwords. We have identified the root cause of these problems, namely the fact that password managers automatically fill out password fields with the clear-text password which are subsequently accessibly by client-side code.

Our approach thwarts this class of attacks through keeping the actual password value out of the reach of potentially malicious JavaScript. It works by filling the password fields with placeholder values which are only replaced later, once the request is sent to the server. For robust protection, our prototype enforces strict integrity constraint in respect to the password form's context and only exchanges the placeholder for the credentials if the origins and names of both login URL and saved password match.

Our solution robustly protects auto-saved passwords from XSS-based theft attempts, and, in the majority of the examined cases, mitigates related, network-level attacks uncovered by Gonzalez et al [9], while maintaining a high degree of compatibility with the currently established behavior of password managers and real-world practices in handling password fields.

Acknowledgements

The authors would like to thank Eric Schmall and Armin Stock for their work pertaining to the implementational parts of this work as well as the anonymous reviewers for their helpful comments. Martin Johns' work was partially funded by the EU projects WebSand (FP7-256964, `http://websand.eu`) and STREWS (FP7-318097, `http://strews.eu`). The support is gratefully acknowledged.

9. REFERENCES

[1] Alexa Internet, Inc. Alexa Top 500 Global Sites. Website, `http://www.alexa.com/`, accessed in March 2010.

[2] Barth, A. The web origin concept, November 2009.

[3] Bojinov, H., Bursztein, E., Boyen, X., and Boneh, D. Kamouflage: Loss-resistant password management. In *Computer Security–ESORICS 2010*. Springer, 2010, pp. 286–302.

[4] Chiasson, S., van Oorschot, P. C., and Biddle, R. A usability study and critique of two password managers. In *15th USENIX Security Symposium* (2006), pp. 1–16.

[5] Dierks, T., and Allen, C. The TLS Protocol Version 1.0. RFC 2246, `http://www.ietf.org/rfc/rfc2246.txt`, January 1999.

[6] Dolske, J. On firefox's password manager. [online] `https://blog.mozilla.org/dolske/2013/08/20/on-firefoxs-password-manager/`, August 2013.

[7] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617, `http://www.ietf.org/rfc/rfc2617.txt`, June 1999.

[8] Gasti, P., and Rasmussen, K. B. On the security of password manager database formats. In *Computer Security–ESORICS 2012*. Springer, 2012, pp. 770–787.

[9] Gonzalez, R., Chen, E. Y., and Jackson, C. Automated password extraction attack on modern password managers. *arXiv preprint arXiv:1309.1416* (2013).

[10] Google Developers. Chrome Extensions - Developer's Guide. [online], `http://developer.chrome.com/extensions/devguide.html`, last access 06/05/13, 2012.

[11] Halderman, J. A., Waters, B., and Felten, E. W. A convenient method for securely managing passwords. In *Proceedings of the 14th international conference on World Wide Web* (2005), ACM, pp. 471–479.

[12] Hickson, I. Web forms 2.0, Apri 2005.

[13] Ives, B., Walsh, K. R., and Schneider, H. The domino effect of password reuse. *Communications of the ACM 47*, 4 (2004), 75–78.

[14] Karole, A., Saxena, N., and Christin, N. A comparative usability evaluation of traditional password managers. In *Information Security and Cryptology-ICISC 2010*. Springer, 2011, pp. 233–251.

[15] Klein, A. Dom based cross site scripting or xss of the third kind. *Web Application Security Consortium, Articles 4* (2005).

[16] Lekies, S., Stock, B., and Johns, M. 25 million flows later: large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 1193–1204.

[17] Magazinius, J., Phung, P. H., and Sands, D. Safe wrappers and sane policies for self protecting JavaScript. In *The 15th Nordic Conference in Secure IT Systems* (October 2010), T. Aura, Ed., LNCS, Springer Verlag. (Selected papers from AppSec 2010).

[18] Mazurek, M. L., Komanduri, S., Vidas, T., Bauer, L., Christin, N., Cranor, L. F., Kelley, P. G., Shay, R., and Ur, B. Measuring password guessability for an entire university. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 173–186.

[19] Microsoft. Ie8 security part vii: Clickjacking defenses, 2009.

[20] Mozilla. *Firefox Add-On SDK - Passwords*.

[21] Mozilla Developer Network. How to Turn Off Form Autocompletion. [online], `https://developer.mozilla.org/en-US/docs/Mozilla/How_to_Turn_Off_Form_Autocompletion`, May 2013.

[22] Mozilla Developer Network. Object.defineProperty(). [online], `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty`, November 2013.

[23] O'Shannessy, P. Bug 359675 - provide an option to manually fill forms and log in.

[24] OWASP. Cross-site scripting (xss), September 2013.

[25] Ross, B., Jackson, C., Miyake, N., Boneh, D., and Mitchell, J. C. Stronger password

authentication using browser extensions. In *Proceedings of the 14th Usenix Security Symposium* (2005), vol. 1998.

[26] Rydstedt, G., Bursztein, E., Boneh, D., and Jackson, C. Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites. In *Web 2.0 Security and Privacy (W2SP 2010)* (2010).

[27] Security, W. Website security statistics report, May 2013.

[28] Toews, B. Abusing password managers with xss. online, 04 2012.

[29] Wu, M., Miller, R. C., and Little, G. Web wallet: preventing phishing attacks by revealing user intentions. In *Proceedings of the second symposium on Usable privacy and security* (2006), ACM, pp. 102–113.

[30] Ye, Z. E., Smith, S., and Anthony, D. Trusted paths for browsers. *ACM Transactions on Information and System Security (TISSEC) 8*, 2 (2005), 153–186.

[31] Zhao, R., and Yue, C. All your browser-saved passwords could belong to us: A security analysis and a cloud-based new design. In *Proceedings of the third ACM conference on Data and application security and privacy* (2013), ACM, pp. 333–340.