Password Managers: Attacks and Defenses

David Silver[1], Suman Jana[1], Eric Chen[2], Collin Jackson[2], and Dan Boneh[1]

[1]Stanford University
[2]Carnegie Mellon University

Abstract

We study the security of popular password managers and their policies on automatically filling in Web passwords. We examine browser built-in password managers, mobile password managers, and 3rd party managers. We observe significant differences in autofill policies among password managers. Several autofill policies can lead to disastrous consequences where a remote network attacker can extract multiple passwords from the user's password manager without any interaction with the user. We experiment with these attacks and with techniques to enhance the security of password managers. We show that our enhancements can be adopted by existing managers.

1 Introduction

With the proliferation of Web services, ordinary users are setting up authentication credentials with a large number of sites. As a result, users who want to setup different passwords at different sites are driven to use a password manager. Many password managers are available: some are provided by browser vendors as part of the browser, some are provided by third parties, and many are network based where passwords are backed up to the cloud and synced across the user's devices (such as Apple's iCloud Keychain). Given the sensitivity of the data they manage, it is natural to study their security.

All the password managers (PMs) we examined do not expect users to manually enter managed passwords on login pages. Instead they automatically fill-in the username and password fields when the user visits a login page. Third party password managers use browser extensions to support autofill.

In this paper we study the autofill policies of ten popular password managers across four platforms and show that all are too loose in their autofill policies: they autofill the user's password in situations where they should not thereby exposing the user's password to potential attackers. The results can be disastrous: an attacker can extract many passwords from the user's password manager without the user's knowledge or consent as soon as the user connects to a rogue WiFi network such as a rogue router at a coffee shop. Cloud-based password syncing further exacerbates the problem because the attacker can potentially extract user passwords that were never used on the device being attacked.

Our results. We study the security of password managers and propose ways to improve their security.

- We begin with a survey of how ten popular password managers decide when to autofill passwords. Different password managers employ very different autofill policies, exposing their users to different risks.
- Next, we show that many corner cases in autofill policies can lead to significant attacks that enable remote password extraction without the user's knowledge, simply by having the user connect to a rogue router at a coffee shop.
- We believe that password managers can help strengthen credential security rather than harm it. In Section 5 we propose ways to strengthen password managers so that users who use them are more secure than users who type in passwords manually. We implemented the modifications in the Chrome browser and report on their effectiveness.

We conclude with a discussion of related work on password managers.

An example. We give many examples of password extraction in the paper, but as a warm-up we present one example here. Consider web sites that serve a login page over HTTP, but submit the user's password over HTTPS (a setup intended to prevent an eavesdropper from reading the password but actually leaves the site vulnerable). As we show in Section 4, about 17% of the Alexa Top 500 websites use this setup. Suppose a user, Alice, uses a password manager to save her passwords for these sites

At some point later, Alice connects to a rogue WiFi router at a coffee shop. Her browser is directed to a landing page that asks her to agree to the terms of service, as is common in free WiFi hotspots. Unbeknownst to Alice, the landing page (as shown in Figure 1) contains

multiple invisible iFrames pointing to the login pages of the websites for which Alice has saved passwords. When the browser loads these iFrames, the rogue router injects JavaScript into each page and extracts the passwords autofilled by the password manager.

This simple attack, without any interaction with the user, can automatically extract passwords from the password manager at a rate of about ten passwords per second. Six of the ten password managers we examined were vulnerable to this attack. From the user's point of view, she simply visited the landing page of a free WiFi hotspot. There is no visual indication that password extraction is taking place.

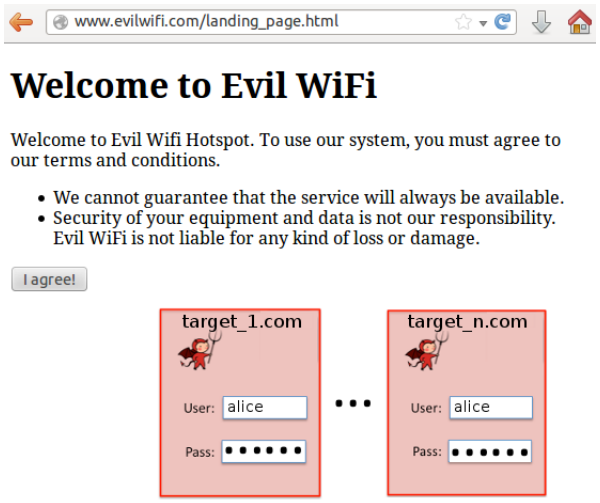

Figure 1: A sample landing page of a rogue WiFi hotspot containing invisible iFrames to the target sites. Note that the iFrames are actually invisible to the user and shown here only for clarity.

2 Password managers: a survey

We begin with a detailed survey of the autofill policies implemented in widely deployed password managers. The password managers we survey include:

- **Desktop Browser PMs**: Google Chrome 34, Microsoft Internet Explorer 11, Mozilla Firefox 29, and Apple Safari 7.

- **3rd Party PMs**: 1Password [1], LastPass [29], Keeper [28], Norton IdentitySafe [26], PasswordSafe [32], and KeePass [27]. All of these besides PasswordSafe and KeePass provide browser extensions that support password field autofill.

- **iOS PMs**: Mobile Safari's password manager syncs with the desktop version of Safari through Apple's iCloud Keychain synchronization service. Since mobile Safari does not support extensions, 3rd Party PMs are separate applications with their own built-in web browser. In addition to Mobile Safari, we survey password managers in Google Chrome, 1Password, and LastPass Tab.

- **Android PMs**: the default Android browser and Chrome.

All these password managers offer an "autofill" functionality, wherein the password manager automatically populates the username and password fields within the user's web browser. We divide autofill strategies into two broad categories:

- **Automatic autofill:** populate username and password fields as soon as the login page is loaded without requiring any user interaction. Password managers that support automatic autofill include Chrome (all platforms), Firefox, Safari, LastPass, Norton IdentitySafe, and LastPass Tab.

- **Manual autofill:** require some user interaction before autofilling. Types of interaction include clicking on or typing into the username field, pressing a keyboard shortcut, or pressing a button in the browser. Password managers that always require manual interaction include 1Password, Keeper, Password Safe, and KeePass.

Internet Explorer 11 uses a hybrid approach: it automatically autofills passwords on pages loaded over HTTPS, but requires user interaction on pages loaded over HTTP. We show in Section 4 that even this conservative behavior still enables some attacks.

Some password managers require manual interaction for autofill in specific situations:

- Chrome requires manual interaction if the password field is in an iFrame.

- Chrome can read passwords stored in Mac OS X's system-wide keychain, but will not automatically autofill them until they have been manually selected by the user at least once.

- The first time Safari or Chrome on Mac OS X access a password in the system keychain, a system dialog requests permission from the user. If the user chooses "Always Allow", this dialog will not be shown again and the password will automatically

autofill in the future. This dialog does not appear if the password was synchronized from another device using iCloud Keychain.

- LastPass and Norton IdentitySafe provide non-default configuration options to disable automatic autofill. In this paper we only discuss the default configurations for these password managers.

2.1 Autofill policies

Next, we ask what happens when the PM is presented with a login page that is slightly different from the login page at the time the password was saved. Should the PM apply autofill or not? Different PMs behave differently and we survey the different policies we found. Table 1 summarizes some of our findings.

The domain and path. All password managers we tested allow passwords to be autofilled on any page within the same domain as the page from which the password was originally saved. For example, a password originally saved on `https://www.example.com/aaa.php` would be filled on `https://www.example.com/bbb.php`. This allows autofill to function on sites that display the login form on multiple pages, such as in a page header visible on all pages. It also allows autofill after a site redesign that moves the login form.

This feature means that an attacker can attack the password manager (as in Section 4) on the *least-secure* page within the domain. It also means that two sites hosted on the same domain (ie, `example.edu/~jdoe` and `example.edu/~jsmith`) are treated as a single site by the password manager.

Protocol: HTTP vs. HTTPS. Suppose the password was saved on a login page loaded over one protocol (say, HTTPS), but the current login page is loaded over a different protocol (say, HTTP)? All other elements of the URL are the same, including the domain and path. Should the password manager autofill the password on the current login page?

Chrome, Safari, Firefox, and Internet Explorer all refuse to autofill if the protocol on the current login page is different from the protocol at the time the password was saved. However, 1Password, Keeper, and LastPass all allow autofill after user interaction in this case. Note that LastPass normally uses automatic autofill, so this downgrade to manual autofill on a different protocol was implemented as a conscious security measure. Norton IdentitySafe does not pay attention to the protocol. It automatically autofills a password saved under HTTPS on a page served by HTTP. As we show later on, any form of autofilling, manual or not, is dangerous on a protocol change.

Modified form action. A form's action attribute specifies where the form's contents will be sent to upon submission.

```
<form action="example.com" method="post">
```

One way an attacker can steal a user's password is to change the action on the login form to a URL under the attacker's control. Therefore, one would expect password managers to not autofill a login form if the form's action differs from the action when the password was first saved.

We consider two different cases. First, suppose that at the time the login page is loaded the form's action field points to a different URL than when the password was first saved. Safari, Norton IdentitySafe and IE (on HTTPS pages) nevertheless automatically autofill the password field. Desktop Chrome and IE (on HTTP pages) autofill after some manual interaction with the user. LastPass asks for user confirmation before filling a form whose action points to a different origin than the current page.

Second, suppose that at the time the login page is loaded the form's action field points to the correct URL. However, JavaScript on the page modifies the form action field so that when the form is submitted the data is sent to a different URL. All of the password managers we tested allow an autofilled form to be submitted in this case even though the password is being sent to the wrong location. We discuss the implications of this in Section 4 and discuss mitigations in Section 5.

Password managers without automatic autofill require user interaction before filling the form, but none give any indication to the user that the form's action does not match the action when the credentials were first saved. Since a form's action is normally not visible to the user, there is no way for the user to be sure that the form was submitting to the place the user intended.

The effects of the action attribute on autofill behavior is captured in the third and fourth columns of Table 1.

Autocomplete attribute A website can use the *autocomplete* attribute to suggest that autocompletion be disabled for a form input [44]:

```
<input autocomplete="off" ... >
```

We find that Firefox, Mobile Safari, the default Android Browser, and the iOS version of Chrome respect the autocomplete attribute when it is applied to a password input. If a password field has its autocomplete attribute set to "off", these password managers will neither fill it nor offer to save new passwords entered into it. All

Platform	Password manager	Same protocol and action	Different protocol	Different form action on load	Different form action on submit	auto-complete = "off"	Broken HTTPS
Mac OS X 10.9.3	Chrome 34.0.1847.137	Auto	No Fill	Manual	Auto	Auto	No Fill
	Firefox 29.0.1	Auto	No Fill	None	Auto	No Fill	Auto
	Safari 7.0.3	Auto	No Fill	Auto	Auto	Auto	Auto
Safari ext.	1Password 4.4	Manual	Manual	Manual	Manual	Manual	Manual
Safari ext.	LastPass 3.1.21	Auto	Manual	Warning	Auto	Auto	Auto
Safari ext.	Keeper 7.5.26	Manual	Manual	Manual	Manual	Manual	Manual
Windows 8.1 Pro	IE 11.0.9600.16531	Auto/Man	No Fill	Auto/Man	Auto/Man	Auto/Man	Manual
	KeePass 2.24	Manual	Manual	Manual	Manual	Manual	Manual
IE addon	IdentitySafe 2014.7.0.43	Auto	Auto	Auto	Auto	Auto	Auto
iOS 7.1.1	Mobile Safari	Auto	No Fill	Auto	Auto	No Fill	Auto
	1Password 4.5.1	Manual	Manual	Manual	Manual	Manual	Manual
	LastPass Tab 2.0.7	Auto	Manual	Auto	Auto	Auto	Auto
	Chrome 34.0.1847.18	Auto	No Fill	No Fill	Auto	No Fill	Auto
Android 4.3	Chrome 34.0.1847.114	Auto	No Fill	No Fill	Auto	Auto	No Fill
	Android Browser	Auto	No Fill	Auto	Auto	No Fill	Auto

Table 1: Password Manager autofill behavior (automatic autofill, manual autofill, or no fill), depending on the protocol (http/https), autocomplete attribute, form action used on the current page relative to the protocol, and form action used when the password was saved. *Manual autofilling* refers to autofilling a password after some user interaction, such as a click or tap on one of the form fields. *No fill* means that no autofilling of passwords takes place. The second to last column refers to autofill behavior when the password field's autocomplete attribute is set to "off". The last column refers to autofill behavior for a login page loaded over a bad HTTPS connection.

of the other password managers we tested fill the password anyway, ignoring the value of the autocomplete attribute. LastPass ignores the attribute by default, but provides an option to respect it.

Once the password manager contains a password for a site, the autocomplete attribute does not affect its vulnerability to the attacks presented in this paper. As described in Section 4, in our setting, the attacker controls the network and can modify the login form to turn the password input's autocomplete attribute on even if the victim website turns it off.

In supporting browsers, the autocomplete attribute can be used to prevent the password from being saved at all. This trivially defends against our attacks, as they require a saved password. However, it is not a suitable defense in general due to usability concerns. A password manager that doesn't save or fill passwords will not be popular amongst users.

Broken HTTPS behavior. Suppose the password was saved on a login page loaded over a valid HTTPS connection, but when visiting this login page at a later time the resulting HTTPS session is broken, say due to a bad certificate. The user may choose to ignore the certificate warning and visit the login page regardless. Should the password manager automatically autofill passwords in this case? The desktop and Android versions of Chrome refuse to autofill passwords in this situation. IE downgrades from automatic to manual autofill. All other password managers we tested autofill passwords as normal when the user clicks through HTTPS warnings. As we will see, this can lead to significant attacks.

Modified password field name. All autofilling password managers, except for LastPass, autofill passwords even when the password element on the login page has a `name` that differs from the name present when the password was first saved. Autofilling in such situations can lead to "self-exfiltration" attacks, as discussed in Section 5.2.1. LastPass requires manual interaction before autofilling a password in a field whose name is different from when the password was saved.

2.2 Additional PM Features

Several password managers have the following security features worth mentioning:

iFrame autofill. Norton IdentitySafe, Mobile Safari and LastPass Tab do not autofill a form in an iFrame that is not same-origin to its parent page. Desktop Chrome requires manual interaction to autofill a form in an iFrame regardless of origin. Chrome for iOS and the Android browser will never autofill an iFrame. Firefox, Safari,

and Chrome for Android automatically autofill forms in iFrames regardless of origin.

Safari and Mobile Safari will only autofill a single login form per top-level page load. If a page, combined with all of its iFrames, has more than one login form, only the first will be autofilled.

We discuss the impact of these policies on security in Section 4.

Visibility. Norton IdentitySafe does not automatically autofill a form that is invisible because its CSS display attribute is set to "none" (either directly or inherited from a parent). However, it *will* automatically autofill a form with an opacity of 0. Therefore, this defense does not enhance security.

Autofill method. KeePass is unique amongst desktop password managers in that it does not integrate directly with the browser. Instead, it can "autotype" a sequence of keystrokes into whatever text field is active. For most login forms, this means it will type the username, the Tab key, the password, then the Enter key to populate and submit the form.

Autofill and Submit. 1Password, LastPass, Norton IdentitySafe, and KeePass provide variants of "autofill and submit" functionality, in which the password managers not only autofills a login form but also automatically submits it. This frees the user from interacting with the submit button of a login form and thus makes autofill more convenient for the user.

3 Threat Model

In the next section we present a number of attacks against password managers that extract passwords from all the managers we examined. First, we define the attackers capabilities and goals. We only consider active man-in-the-middle network attackers i.e. we assume that the adversary can interpose and modify arbitrary network traffic originating from or destined to the user's machine. However, unlike standard man-in-the-middle attacks, we do not require the user to log into any target websites in the presence of the attacker. Instead, the setup consists of two phases:

First, the user logs in to a number of sites and the attacker cannot observe or interfere with these logins. The user's password manager records the passwords used for these logins. For password managers that support syncing of stored passwords across multiple machines (e.g., Apple's iCloud KeyChain), users may even carry out this step on an altogether different device from the eventual victim device.

At a later time the user connects to a malicious network controlled by the attacker, such as a rogue WiFi router in a coffee shop. The attacker can inject, block, and modify packets and its goal is to extract the passwords stored in the user's password manager without any action from the user.

We call this type of attacker the evil coffee shop attacker. These attacks require only temporary control of a network router and are much easier and thus more likely to happen in practice. We show that even such weak man in the middle attackers can leverage design flaws in password managers to remotely extract stored passwords without the user logging into any website.

The attacker has no software (malware) installed on the user's machine. We only assume the presence of a password manager acting in the context of a web browser.

4 Remote extraction of passwords from password managers

We show that an evil coffee shop attacker can extract passwords stored in the user's password manager. In many of our attacks the user need not interact with the victim web site and is unaware that password extraction is taking place. We discuss defenses in Section 5.

4.1 Sweep attacks

Sweep attacks take advantage of automatic password autofill to steal the credentials for multiple sites at once without the user visiting any of the victim sites. For password managers backed by a syncing service (such as Apple's iCloud Keychain) the attacker can extract site passwords even if the user never visited the site on that device. These attacks work in password managers that support automatic autofill, highlighting the fundamental danger of this feature.

Sweep attacks consist of three steps. First, the attacker makes the user's browser visit an arbitrary vulnerable webpage at the target site without the user's knowledge. Next, by tampering with network traffic the attacker injects JavaScript code into the vulnerable webpage as it is fetched over the network using one of the methods described in Section 4.2. Finally, the JavaScript code exfiltrates passwords to the attacker using the techniques in Section 4.3.

In the sweep attacks we implemented, the user connects to a WiFi hotspot controlled by the attacker. When the user launches the browser, the browser is redirected to a standard hotspot landing page asking for user consent to standard terms of use. This is common behavior for public hotspots. Unbeknownst to the user, however,

the landing page contains invisible elements that implement the attack.

iFrame sweep attack. Here the innocuous hotspot landing page contains invisible iFrames pointing to the arbitrary pages at multiple target sites. When the browser loads these iFrames, the attacker uses his control of the router to inject a login form and JavaScript into each iFrame using the methods described in Section 4.2. As we will see, injecting a login form and JavaScript is not difficult and can be done in several different ways. All that is needed is *some* vulnerable page on the target site. It is especially easy for sites that serve their login page over HTTP (but submit passwords over HTTPS), which is a common setup discussed in the next section.

As each iFrame loads, the password manager will automatically populate the corresponding password field with the user's password. The injected JavaScript in each iFrame can then steal and exfiltrate these credentials.

Our experiments show that this method can extract passwords, unbeknownst to the user, at a rate of about ten passwords per second. To prevent the user from clicking through the landing page before the attacks are done, the landing page includes a JavaScript animated progress bar that forces the user to wait until the attacks complete.

We also find that the password extraction process can be made more efficient by arranging the iFrames in a hierarchical structure instead of adding one iFrame to the top-level page for each target website. Adding all the iFrames to the top-level page would create large increases in both the amount of traffic on the network and the amount of memory used by the victim's browser. Hierarchical arrangement of the iFrames can avoid such issues. The top-level iFrame contains most of the code for the attack and dynamically spawns child frames and navigates them to the target pages. This technique allows the iFrames to load asynchronously and thus ensures that network and memory usage remain reasonable for the duration of the attack.

Chrome (all platforms) is the only automatic autofill password manager that is not vulnerable to the iFrame-based attack, because they never automatically autofill passwords in iFrames. All the other automatic autofill password managers are vulnerable to this attack. Even though the autofill policies of Norton IdentitySafe, Safari, Mobile Safari, and LastPass Tab described in Section 2.2 restrict the number of passwords that can be stolen in a single sweep to 1, they remain vulnerable.

Window sweep attack. A variant of the sweep attack uses windows instead of invisible iFrames. If the attacker can trick users into disabling their popup blocker (e.g., by requiring a window to open before the user can gain access to the WiFi network), the landing page can open each of the victim pages in a separate window. This is more noticeable than the iFrame-based approach, but the JavaScript injected into each victim page can disguise these windows to minimize the chances of detection. Techniques for disguising the windows include minimizing their size, moving them to the edge of the screen, hiding the pages' contents so that they appear to the user as blank windows, and closing them as soon as the password has been stolen.

Nearly all automatic autofill password managers, including desktop Chrome, are vulnerable to the window-based attack. Only LastPass Tab is not vulnerable, as it does not support popup windows at all. Hence, although iFrames make the sweep attack easier, they are not required.

Redirect sweep attack. A redirect sweep attack enables password extraction without any iFrames or separate windows. In our implementation, once the user connects to a network controlled by the attacker and requests an arbitrary page (say, a.com), the network attacker responds with an HTTP redirect to some vulnerable page on the target site (say, b.com). The user's browser receives the redirect and issues a request for the page at b.com. The attacker allows the page to load, but injects a login form and JavaScript into the page, as described in Section 4.2. The injected JavaScript disguises the page (for example, by hiding its body) so that the user does not see that b.com is being visited.

When the user's browser loads the page from b.com, the vulnerable password manager will automatically autofill the login form with the credentials for b.com, which the injected JavaScript can then exfiltrate. Once done, the injected JavaScript redirects the user's browser to the next victim site, (say c.com) and exfiltrates the user's password at c.com in the same way. When sufficiently many passwords have been exfiltrated the attacker redirects the user's browser to the original page requested by the user (a.com).

This attack leaves small indications that password extraction took place. While the attack is underway the user's address bar will display the address of the attacked site, and the attacked site will remain in the user's history. However, as long as the body of the page itself is disguised, most users will not notice these small visual clues.

All of the automatic autofill password managers we tested were vulnerable to this attack.

Summary. Table 2 describes which password managers are vulnerable to these sweep attacks.

Platform	Password Manager	iFrame sweep	Window sweep	Redirect sweep
Mac OS X 10.9.3	Chrome 34.0.1847.137		+	+
	Firefox 29.0.1	+	+	+
	Safari 7.0.3	Single	+	+
Safari ext.	1Password 4.4			
Safari ext.	LastPass 3.1.21	+	+	+
Safari ext.	Keeper 7.5.26			
Windows 8.1 Pro	Internet Explorer 11.0.9600.16531	HTTPS, SO	HTTPS	HTTPS
	KeePass 2.24			
IE addon	Norton IdentitySafe 2014.7.0.43	SO	+	+
iOS 7.1.1	Mobile Safari	Single, SO	+	+
	1Password 4.5.1			
	LastPass Tab 2.0.7	SO		+
	Chrome 34.0.1847.18		+	+
Android 4.3	Chrome 34.0.1847.114		+	+
	Android Browser		+	+

Table 2: Vulnerability to sweep attacks. **+** indicates vulnerability without restriction. **HTTPS** indicates vulnerability only on pages served over HTTPS. **Single** indicates a single site is vulnerable per top-level page load. **SO** indicates vulnerability when the page containing the iFrame is same-origin with the target page in the iFrame.

Attack amplification via password sync. Most password managers offer services that synchronize users' passwords between different devices. These password synchronization services can potentially result in password extraction from devices without them ever having visited the victim site.

Suppose the user's password manager syncs between their desktop and tablet, and will automatically autofill a password synced from another device without user interaction. Suppose further that the site c.com is vulnerable to network attacks and thus to the attacks described above. The user is careful and only ever visits c.com on their desktop, which never leaves the user's safe home network. However, when the user connects their tablet to the attacker's WiFi network at a coffee shop, the attacker can launch a sweep attack on the user's tablet and extract the user's password for c.com even though the user has never visited c.com on their tablet.

We tested Apple's iCloud Keychain, Google Chrome Sync, Firefox Sync, and LastPass Tab, and found all of them to be vulnerable to this attack. In general, any password manager that automatically autofills a password synced from another device will be vulnerable to this type of attack amplification. Therefore, the security of any password manager is only as strong as the security of the weakest password manager it syncs with.

4.2 Injection Techniques

Sweep attacks rely on the attacker's ability to modify a page on the victim site by tampering with network traffic. The attacks are simplest when the vulnerable page is the login page itself. However, any page that is same-origin with login page is sufficient, as all password managers associate saved passwords with domains and ignore the login page's path. The attacker can inject a login form into any page in the origin of the actual login page and launch a password extraction attack against that page. We list a few viable injections techniques.

HTTP login page. Consider a web site that serves its login page over HTTP, but submits the login form over HTTPS. While this setup protects the user's password from eavesdropping when the form is submitted, a coffee shop attacker can easily inject the required JavaScript into the login form at the router and mount all the sweep attacks discussed in the previous section.

Clearly serving a login form over HTTP is bad practice because it exposes the site to SSLstrip attacks [31]. However extracting passwords via SSLstrip requires users to actively enter their passwords while connected to the attacker's network and visiting the victim page. In contrast, the sweep attacks in the previous section extract passwords without any user interaction.

To test the prevalence of this setup — a login page loaded over HTTP, but login form submitted over HTTPS — we surveyed Alexa Top 500 sites (as of October 2013) by manually visiting them and examining their login procedures. Of the 500 sites surveyed, 408 had login forms. 71 of these 408 sites, or 17.40%, use HTTP for loading the login page, but HTTPS for submitting it. Some well known names are on this list of 71 sites, including `ask.com`, `godaddy.com`, `reddit.com`,

huffingtonpost.com, and att.com.

Additionally, 123 (or 30.15%) of the sites used HTTP both for loading the login page and for submitting it. This setup is trivially vulnerable to eavesdropping, but a vulnerable password manager increases this vulnerability by removing the need for a human to enter their password. For the purposes of our attacks, these sites can be thought of as an especially vulnerable subclass of sites with a login form served over HTTP.

Passwords for all these vulnerable websites can be easily extracted from an autofilling password manager using the sweep attacks in the previous section. One could argue that all these sites need to be redesigned to load and submit the login page over HTTPS. However, until that is done there is a need to strengthen password managers to prevent these attacks. We discuss defenses in Section 5.

Embedded devices I. Many embedded devices serve their login pages over HTTP by default because the channel is assumed to be protected by a WiFi encryption protocol such as WPA2. Indeed, Gourdin et al. report that the majority of the embedded web interfaces still use HTTP [20]. Similarly, internal servers in a corporate network may also serve web login pages over HTTP because access to these servers can only be done over a Virtual Private Network (VPN).

Sweep attacks are very effective against these devices: the password manager autofills the password even when the underlying network connection is insecure. By injecting JavaScript into the HTTP login page as above, a coffee shop attacker can extract passwords for embedded devices and corporate servers that the user has previously interacted with.

Embedded Devices II. Some home routers serve their login pages over HTTPS, but use are self-signed certificates. An attacker can purchase a valid certificate for the same common name as the router's [37] or generate its own self signed certificate. When the user's machine connects to the attacker's network, the attacker can spoof the user's home router by presenting a valid certificate for the router's web site. This allows the attacker to mount the sweep attack and extract the user's home router password.

Broken HTTPS. Consider a public site whose login page is served over HTTPS. In Section 2 we noted that many password managers that autofill passwords automatically do so even when the login page is loaded over a broken HTTPS connection, say due to a bad certificate. This can be exploited in our redirect sweep attack: when the browser is redirected to the victim site, the attacker serves the modified login page using a self signed cert for that site. This modified login page contains a login form and the JavaScript needed to exfiltrate the user's password once it is autofilled by the password manager.

These self signed certs will generate HTTPS warning in the browser, but if the redirect sweep attack happens as part of the process of logging on to the hotspot, the user is motivated to click through the resulting HTTPS warning messages. As a result the attacker can extract user passwords from the password manager, even for sites where the login page is served over HTTPS.

Indeed, several prior works have found that users often tend to click through HTTPS warnings [42, 3]. The user may decide to click through the warning and visit the site anyway, but not enter any sensitive information. Nevertheless, the user's password manager autofills the password resulting in password extraction by the attacker, regardless of the user's caution. All of the password managers we tested fill passwords even when the user has clicked through an SSL warning, with the exception of the desktop and Android versions of Chrome.

Active Mixed Content. Any HTTPS webpage containing active content (e.g., scripts) that is fetched over HTTP is also a potential vector. If rendering active mixed content is enabled in the user's browser, any HTTPS page containing active mixed content is vulnerable to injection. Chrome, Firefox, and IE block active mixed content by default but provide a user option to enable it. Safari, Mobile Safari, and the Android stock browser allow active mixed content to be fetched and executed without any warnings. Several types of active mixed content, especially those processed by browser plugins, are harder to block. For example, embedding a Shockwave Flash (SWF) file over HTTP if not blocked correctly can be used by a network attacker to inject arbitrary scripts [24].

XSS Injection. A cross-site scripting vulnerability in a page allows the attacker to inject JavaScript to modify the page as needed [18]. XSS vulnerabilities are listed as one of the most common web vulnerabilities in 2013 internet security threat report by Symantec [14]. If an XSS vulnerability is present on *any* page of the victim site, the sweep attacks will work even if the site's login page is served over HTTPS. For example, the attacker simply includes an iFrame or a redirect on the malicious hotspot landing page that links to the XSS page. The link uses the XSS vulnerability to inject the required login form and JavaScript into the page.

Furthermore, an XSS vulnerability allows for a weaker threat model than our coffee shop attacker. An ordinary web attacker can trick the user into visiting his site, then launch the attack through the XSS vulnerability. This style of attack requires no access to the user's network

and has been suggested previously by RSnake [36] and Saltzman et al. [39].

Leftover Passwords. The user's password manager may contain leftover passwords from older, less secure versions of a site. An attacker could spoof the old site to steal the leftover password. Unless the user is proactive about removing older passwords, updating the security of the site does not protect the domain from this type of attack. For example, if a user's password manager contained a password for Facebook from before its switch to HTTPS, an attacker could spoof an HTTP Facebook login page to steal the password.

4.3 Password Exfiltration

In the previous section we referred to JavaScript that exfiltrates the user's password once it is autofilled by the password manager. Once the password manager has autofilled the login form, the attacker must be able to access the filled-in credentials and send them to a server under its control. We briefly describe two methods for accomplishing this.

4.3.1 Method #1: Stealth

Using stealth exfiltration, the attacker waits until the login form is populated with the user's credentials automatically by a password manager, then steals the password by loading an attacker controlled page in an invisible iFrame and passing the credentials as parameters. The following simple JavaScript does just that and works with all password managers we tested:

```
function testPassword() {
 var password =
        document.forms[0].password.value;
 if(password != "") {
  var temp = document.createElement("div");
  temp.innerHTML +=
        "<iFrame src=\""+ attacker_addr +
        "?password=" + password +
        "\" style=\"display:none;\" />";
  document.body.appendChild(temp);
  clearInterval(interval);
}}
interval = setInterval(testPassword, 50);
```

4.3.2 Method #2: Action

An HTML form's "action" is the URL to which the form's data will be submitted. The attacker can modify a login form's action attribute so that it submits to an attacker-controlled site, thereby leaking the user's credentials to the attacker. If the attacker redirects the user's browser back to the real action, the user will not notice the change.

Automatic autofill password managers populate password forms when the page first loads. The attacker can then use injected JavaScript to change the action, submit the form and steal the password. If the login page is loaded in an iFrame or if it is rendered invisible, the users will not even realize that a login form was submitted. The following simple code does just that:

```
changer = function() {
  document.forms[0].action = attacker_addr;
  document.forms[0].submit();   }
setTimeout(changer, 1000);
```

In section 2.1 we showed that password managers that automatically autofill passwords do so on page load and show no warning to the user when the submitted form action differs from the action when the password was first saved. Thus, all password managers with automatic autofill are vulnerable to this exfiltration method.

4.4 Attacks that need user interaction

All of the attacks described thus far take advantage of automatic autofill password managers to work when the user does not interact with the login form. However, the exfiltration techniques we described work regardless of how the login form was filled. If the user's password manager requires user input to fill passwords and an attacker can trick the user to interact with the login form without them realizing it, the same exfiltration techniques can be used to steal the password as soon as the password form is filled.

We created a simple "clickjacking" attack [23, 38, 25]. The attacker presents the user with a benign form seemingly unrelated to the target site. Overlaying the benign form is an invisible iFrame pointing to the target site's login page. The iFrame is positioned such that when a user interacts with the benign form, they actually interact with the invisible iFrame — in this case, when the user thinks they are filling a form on a benign site, they are actually filling the password in the target site. Once filled, any of the exfiltration techniques described previously can be used to steal the password. This attack steals a password for one site at a time, but could be repeated to steal passwords for multiple sites.

We confirmed this attack works against both Chrome and Internet Explorer 11, as both required manual interaction before filling in at least some situations.

5 Strengthening password managers

In this section we present two complementary solutions to the attacks presented earlier. Before describing the details of our solutions, we first describe why some of the obvious solutions do not work. For example, as all

our attacks require JavaScript injection, a potential solution is to prevent password managers from autofilling passwords on a page that is vulnerable to JavaScript injection. This solution is hard to implement in practice as some JavaScript injection vectors (e.g., XSS bugs) are extremely hard for the browser to detect. Another possible solution is to completely block autofill inside iFrames. However, this solution does not prevent the window or redirect sweep attacks described in Section 4. Moreover, blocking autofill inside iFrames will inconvenience users of benign websites that include login forms inside iFrames.

5.1 Forcing user interaction

Our ultimate goal is to ensure that using a password manager results in better security than when users manually enter passwords in a password field. This is certainly not the case with password managers today, as the attacks of the previous section demonstrate. We begin with the simplest defense that makes password managers no worse than manual user entry.

Our most powerful attacks exploit the automatic autofill of the password field. An obvious defense is to *always* require some user interaction before autofilling a form. This will prevent sweep attacks where multiple passwords are extracted without any user interaction. Interaction can come in the form of a keyboard shortcut, clicking a button, selecting an entry from a menu, or typing into the username field. Regardless of the type of interaction, it must be protected against clickjacking attacks as described in Section 4.4. The user interaction should occur through trusted browser UI that JavaScript cannot interact with, preventing malicious JavaScript from spoofing user interaction and triggering an autofill.

Furthermore, the password manager should show the domain name being autofilled before the filling occurs, so that users know which site is being autofilled. This reduces the chances of the user filling a form without meaning to. For example, if a login page for one site contains an invisible iFrame pointing to the login page of another site, the user must explicitly choose which domain they want filled.

In some settings, such as broken HTTPS, the password manager should simply refuse to autofill passwords.

Implementation. Always forcing user interaction was easy to prototype in Chrome[1] because Chrome already requires user input in certain situations, such as when the action on the current page is different from the action when the password was saved. Since the UI implementation already existed we simply had to always trigger it. We did so by hardcoding the `wait_for_username` variable to `true` in the constructor of the `PasswordFormFillData` object. Note that this does not protect against the clickjacking attacks described in Section 4.4 but can be extended to do so.

Minimizing user inconvenience. As always forcing user interaction before autofilling may cause inconvenience to the user, password managers could provide a "autofill-and-submit" functionality that once triggered by user interaction will autofill the login form and submit it. We found that variants of autofill-and-submit are already supported by 1Password, LastPass, Norton IdentitySafe, and KeePass.

With this feature, the user's total interaction will remain similar to the current manual autofill password managers. Instead of interacting with the submit button after the password managers autofill the login form, the user will interact with the password manager to trigger autofill-and-submit. As long as the conditions stated earlier in this section are satisfied, the use of such a feature will be as secure as manually entering a password.

5.2 Secure Filling

Our main defense, called *secure filling*, is intended to make the use of password managers more secure than typing in passwords manually. Simply requiring user interaction is not sufficient. Indeed, if a login page is loaded over HTTP but submitted over HTTPS, no browser or password manager implementation provides security once the login form has been filled with the user's password: JavaScript can read the password directly from the form or change the form's action so that it submits to a password stealing page hosted by the attacker.

The goal of secure filling is that even if an attacker injects malicious JavaScript into the login page, passwords autofilled by the password manager will remain secure so long as the form is submitted over HTTPS. This defense is somewhat akin to `HttpOnly` cookies [4], but applied to autofilled passwords: they can be submitted to the web server, but cannot be accessed by JavaScript. We discuss compatibility issues at the end of the section.

Our proposed defense works as follows:

1. Along with the username and password, the password manager stores the action present in the login form when the username and password were first saved.

2. When a login form is autofilled by the password manager, the password field becomes unreadable by JavaScript. We say that the autofill is now "in progress".

[1] Chromium build 231333

3. If the username or password fields are modified (by the user or by JavaScript) while an autofill is in progress, the autofill aborts. The password is cleared from the password field, and password field becomes readable by JavaScript once more.

4. Once a form with an autofill in progress is submitted, and *after* all JavaScript code that is going to be run has run, the browser checks that the form's action matches the domain of the action it has stored. If the domains do not match, the password field is erased and the form submission fails. If the domains do match, the form is allowed to submit as normal.

Making the password field unreadable by JavaScript prevents stealth exfiltration, as the malicious JavaScript is unable to read the password field and thus unable to steal the password. Checking the action before allowing the form to submit ensures that the action has not been changed to point to a potentially malicious site. The password is guaranteed to only be filled into a form that submits to the same place as when the password was originally saved. For this to work, it is essential that the check be performed after JavaScript's (and thus the attacker's) last opportunity to modify the form's action.

In the case where the form's action does not match what is stored, it may be desirable to give the user the option to submit the form (and password) anyway. However, the browser should allow the user to make an educated decision by showing the user both the new and original actions and explaining how their password may be leaked. This will weaken security, as the user may chose to submit the form when they should not, but it would improve compatibility when sites undergo a redesign and the login page changes.

Implementation. We implemented a prototype of this defense in Chrome[2] by modifying the `PasswordAutofillAgent` class. In the `FillUserNameAndPassword` method, we fill the password field with a dummy value (a sequence of unprintable characters), then store the real password and the form's action in a `PasswordInfo` object associated with the form. In the `WillSendSubmitEvent` method, we check if the dummy value is still present in the password field; if it is, and if the form's action matches the action we had stored, we replace the dummy value with the real password and allow the form to submit. While our implementation is only a prototype, it shows that implementing this defense is reasonably straightforward, at least in Chrome.

[2]Chromium build 231333

Although browsers vendors will need to implement this functionality in their own password managers, they may consider providing a mechanism for external password manager extensions to implement the same functionality. An API could allow the password manager extension to fill a form and designate it as autofilled, as well as designate the expected action on the form. The behavior would then be the same as with the internal password manager: the password field would become unreadable by JavaScript, and the browser checks that the action has not changed before submitting the form.

5.2.1 Limitations of secure filling

The secure filling approach will cause compatibility issues with existing sites whose login process relies on the ability to read the password field using JavaScript.

AJAX-based login. Some sites submit their login forms using AJAX instead of standard form submission. When the login form's submit button is pressed, these sites use JavaScript to read the form fields, then construct and submit an XMLHttpRequest object. This approach is not compatible with our solution, as JavaScript would not be able to read the filled password field and therefore be unable to construct the XMLHttpRequest. Furthermore, this does not use the form's action field, and therefore the password manager cannot detect when the password is being submitted to a different site than when it was first saved.

To study the impact our proposal would have on existing popular sites, we looked for the use of AJAX for login on the Alexa Top 50 sites, as of October 26, 2013. 10 of the these 50 sites used AJAX to submit logins. 8 of 10 sites were based in China, with only one Chinese site on the list not using AJAX. The remaining two sites were based in Russia and the U.S., with other sites from both countries using ordinary form submission. This suggests the use of AJAX to submit passwords is popular in China but not common elsewhere in the world, and overall AJAX is used by a significant minority of popular sites.

We propose two workarounds that will allow our solution to work with AJAX. First, sites could place the login form in an iFrame instead of using XMLHttpRequest. The iFrame would submit using standard form submission. Using this approach, there is no need for JavaScript to read the form fields and the form's action behaves normally. Therefore, it is fully compatible with our secure filling recommendation, but still allows the user to login asynchronously.

Second, for sites that must use XMLHttpRequest, the browser could provide an additional API that allows JavaScript to submit the password without being able to

read it. The existing XMLHttpRequest API uses a send() method to send data. We propose an additional method, sendPassword(). The sendPassword() method accepts a form as a parameter, and sends the contents of the form's password fields without ever making them readable to other JavaScript. To prevent an attacker from exfiltrating a password using AJAX, the password manager should check that whenever a filled password is sent using sendPassword(), the destination URL matches the destination URL from the first time the filled password was sent.

Although these workarounds will require modifications to a few existing sites, the security benefits are well worth the effort. The only downside for sites that do not make the required modifications is that their users will not be able to use some password managers.

Preventing self exfiltration attacks. Chen et al. [11] point out that in some cases an attacker can extract data using what they call "self-exfiltration." In our setting this translates to the following potential attack: if any page on the victim site supports a public discussion forum, an attacker can cause the secure filling mechanism to submit the password to the forum page and have the password posted publicly. The attacker can later visit the public forum and retrieve the posted passwords. Since the attacker is changing the login form's action to another page in the *same domain* our secure filling mechanism will allow the password to be sent. In this discussion, the public forum can be replaced by any public form-posted data on the victim site

For this attack to work, the `name` of the password field on the login page must be the same as the name of the text field on the public forum page. An attacker can easily accomplish this by sending to the browser a login page with the desired name.

Fortunately, it is straight forward to defend against this issue: our secure filling mechanism should only fill a password field whose `name` matches the `name` of the field when the password was saved. Furthermore, dynamically changing the `name` attribute using JavaScript should cause a fill to abort. This defense prevents the attacker from submitting the password using any field with a name`name` other than the one chosen by the site itself for the login page. This prevents the self exfiltration attack, except for the extremely unlikely event where a public forum page on the victim site has a text field whose `name` happens to be identical to the password field `name` on the login page.

User registration pages. An additional limitation of our secure filling proposal is that it cannot improve the security of manually entered passwords. HTML does not provide a way to distinguish between password fields on user registration pages and password fields in login forms. Registration pages frequently use JavaScript to evaluate passwords before submission — for example, to check password strength or to verify two passwords match. Therefore, JavaScript on registration pages must have access to the password.

There are two solutions to this problem. One option is to forbid JavaScript from reading any password field, and require that registration pages use regular text fields programmatically made to behave like password fields. On every key stroke JavaScript on the page replaces the character with an asterisk, as in a password field. To the user the text field will behave as a password field, yet JavaScript on the registration page will be able to access the password.

Alternatively, HTML can be slightly extended to support two types of password fields, one for login and one for registration. For login, the Password field allows no JavaScript access to its contents as needed for *secure fill*. The PasswordRegistration field used for registration allows JavaScript access to its contents but is never autofilled with a saved password (separate password manager features such as a password generator can continue to work).

5.3 Server-side defenses

How can a site defend itself without support from password managers? As the attacks rely on decisions made client-side by the user's password manager, a complete server-side defense is not possible. However, a few existing best-practices can be used to greatly reduce the attack area:

1. Use HTTPS on both the login page and page it submits to. Ideally, use HTTPS everywhere on the site and enable HSTS (HTTP Strict Transport Security) to prevent pages from ever loading under HTTP.

2. Use CSP (Content Security Policy) to prevent the execution of inline scripts, making the injection of JavaScript directly into the login page ineffective.

3. Host the login page in a different subdomain that the rest of the site (i.e., login.site.com instead of site.com). This limits the number of pages considered same-origin with the login page, reducing the attack surface.

None of these defenses are unique to the attacks we described, but are best-practices that will make our attacks more difficult. Even with these defenses, attacks are still possible — attacks that take advantage of broken HTTPS, for example, will still be feasible. Therefore,

it remains important that password managers implement the fixes we described to fully defend against the attacks.

6 Related work

There have been several prior works about finding vulnerabilities in existing password managers as well as building stronger password authentication systems. We summarize them below.

Vulnerabilities in password managers: Belekno et al. [5] and Gasti et al. [19] surveyed several password managers and found that most of them save passwords to device storage in an insecure manner. However, these attacks have a very different threat model than the attacks described in this paper. They require the attacker to have physical access to a user's device. By contrast, for our attacks we only consider network attackers which is a weaker threat model than the ones requiring physical access.

Besides autofilling of passwords, several password managers also support autofilling of forms with information like name, phone no etc. Prior works [15, 34, 21] have shown that an attacker can steal autofilled information by using specially crafted forms. This is a different class of attack than the attacks on login forms as unlike login passwords, information filled into these forms is not tied to any particular origin. However, for completeness, we summarize our findings about attacks against autofilling of regular forms in Appendix A.

Some existing works [17, 43] have demonstrated how an attacker can use injected JavaScript to steal user's stored passwords in a password manager for login pages that are either vulnerable to XSS attacks or are fetched over HTTP. However, unlike our attacks, these attacks require that users willingly visit the vulnerable website at the presence of the attacker. Reverse Cross-Site Request (RCSR) [7] vulnerabilities perform phishing attacks by leveraging the fact that several password managers will fill in passwords to login forms even if the form's action differs from the action when the password was first saved. These attacks require that the user clicks the submit button. By contrast, our attacks are completely automated and transparent to the user.

The most closely related works to the attacks we present in this paper are by RSnake [36] and Saltzman et al. [39]. RSnake [36] speculated that an attacker can exploit form autofilling tools that fills forms without any user input in sites vulnerable to XSS attacks to extract the autofillable information without users' notice. The basic idea is to inject JavaScript using the XSS attack and exfiltrate the autofilled information. Saltzman et al. [39] suggested that active network attackers can inject iFrames to login forms of websites vulnerable to script injection either through XSS attacks or through pages loaded over HTTP, make the password managers fill those login forms, and steal those passwords without users noticing anything wrong. However, none of these works tested the attacks. We performed a comprehensive study of vulnerabilities and presented several new and different attack vectors (mixed content, broken SSL, embedded device admin pages etc.) and attack techniques (such as the redirect attack).

Using XSS attacks for stealing autofilled passwords has also been explored by Stock et al. [41]. They suggested that the password managers can prevent such attacks by using a placeholder dummy password for autofilling and replacing it with the original one just before submitting the login form to the remote server. In this work, unlike Stock at al., we explore several different vectors for stealing autofilled passwords besides XSS attacks. We also investigate several different third-party password managers together with the builtin password managers that were analyzed by Stock et al.

Blanchou et al. [6] describe several weaknesses of password manager browser extensions and implement a phishing attack that demonstrates the danger of automatic autofill. They do not examine any built-in browser password managers or consider how passwords from multiple sites could be stolen in one attack. They suggest that password managers prevent the cross-domain submission of passwords (what we called action exfiltration in this paper), but do not consider stealth exfiltration.

Fahl et al. [16] demonstrate attacks against Android password managers. However, their attacks were specific to the Android operating system, and most relied upon a malicious Android app, not a network attacker.

Li et al. [30] survey a variety of vulnerabilities specific to third-party web-based password managers and a web attacker, then discuss mitigation strategies. They do not discuss browser or native code password managers, nor a network attacker.

Both the Chromium and Firefox bug databases have bugs filed to prevent autofilling of login information inside an iFrame [12, 10]. However, preventing autofilling of passwords inside iFrames will not prevent the window sweep or the redirect attacks described in Section 4. At the time of this writing, only the Chromium bug has been fixed.

Another Chromium bug [13] seeks to only autofill forms after the user interacts with the login page, but not necessarily the login form. This is not yet implemented, however, increasing the scope of interaction to the entire page will make it easier for the attackers to launch click-

jacking attacks. In contrast, autofilling only after explicit user interaction with the login form as suggested in Section 5 is robust against such attacks.

A Firefox bug [8] discusses man-in-the-middle attacks against the password manager similar to our redirect attack. Another bug [9] suggests that filled passwords should not be readable by JavaScript. Their approach is similar to our secure filling, but remains vulnerable to exfiltration using the action attribute. Although both bugs are several years old, neither has been acted upon.

Password manager features: Aris [2] discusses the autocomplete attribute and why setting autocomplete=off results in poor security in addition to a bad user experience.

Secure password authentication systems: Another related line of research investigated designing secure password authentication systems that can choose strong domain-specific passwords with minimal user intervention [35, 22]. The main motivation behind these works is to minimize the damage caused by users mistakenly revealing their passwords through phishing websites or social engineering These solutions also protect against an attacker leveraging reused passwords that were stolen from a low security website on a high security website. None of these works focus on autofilling of passwords and thus do not help in preventing against the attacks we presented in this paper.

There are also several research works that built password authentication systems that supported autofilling [46, 45]. However, their primary goal was to prevent phishing attacks. In this paper, we focus on existing password managers and thus do not evaluate how vulnerable these systems are against our attacks.

Sandler et al. proposed the 'password booth', a new secure browser-controlled mechanism to let users securely enter passwords that are not unaccessible from JavaScript running as part of the host page's origin [40]. Their solution is similar to our secure filling defense, but does not take password managers into account. Secure filling takes advantage of password managers to provide guarantees the password booth cannot, namely that an autofilled password is submitted to the same origin it was saved from. Furthermore, their proposal requires a dramatic UI change for all users, whereas ours requires only a very minimal UI change from automatic to manual autofill. They suggest that a dramatic change is a feature because it makes security more visible to the user, yet at the same time a dramatic change will reduce adoption from browser developers unwilling to upset their users with change. Ultimately, our two ideas are compatible as the password booth could be extended to work with password managers as we describe in this paper.

An early unpublished version of this paper, containing only a subset of the results, appears as a technical report in [33].

7 Conclusions

In this paper we surveyed a wide variety of password managers and found that they follow very different and inconsistent autofill policies. We showed how an evil coffee shop attacker can leverage these policies to steal the user's stored passwords without any user interaction. We also demonstrated that password managers can prevent these attacks by simply following two steps - never autofilling under certain conditions like in the presence of HTTPS certificate validation errors and requiring user interaction through some form of trusted browser UI, that untrusted JavaScript cannot affect, before autofilling any passwords. Finally, we presented *secure filling*, a defense that makes autofilling password managers more secure than manually entering a password under certain circumstances (e.g., a login page fetched over HTTP but submitted over HTTPS). We hope that this work will improve the security of password managers and encourage developers to adopt our enhancements.

We disclosed our results to the password manager vendors, prompting several changes to autofill policies. Due to our findings, LastPass will no longer automatically autofill password fields in iFrames, and 1Password will no longer offer to fill passwords from HTTPS pages on HTTP pages.

Acknowledgments

This work was supported by NSF, the DARPA SAFER program, and a Google PhD Fellowship to Suman Jana. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF, DARPA, or Google.

References

[1] 1password - agilebits. `https://agilebits.com/onepassword`.

[2] A. Adamantiadis. The war against autocomplete=off, 2013. http://blog.0xbadc0de.be/archives/124.

[3] D. Akhawe and A. P. Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *USENIX Security Symposium*, 2013.

[4] A. Barth. Http state management mechanism. RFC 2965, 2011.

[5] A. Belenko and D. Sklyarov. secure password managers and military-grade encryption on smartphones: Oh, really? *Blackhat Europe*, 2012.

[6] M. Blanchou and P. Youn. Password managers: Exposing passwords everywhere, 2013. https://isecpartners.github.io/whitepapers/passwords/2013/11/05/Browser-Extension-Password-Managers.html.

[7] Bugzilla@Mozilla. Bug 360493 - (cve-2006-6077) cross-site forms + password manager = security failure. https://bugzilla.mozilla.org/show_bug.cgi?id=360493.

[8] Bugzilla@Mozilla. Bug 534541 - passwords from login manager can be intercepted by mitm attacker (e.g. evil wifi hotspot or dns poisoning). https://bugzilla.mozilla.org/show_bug.cgi?id=534541.

[9] Bugzilla@Mozilla. Bug 653132 - auto-filled password fields should not have their values available to javascript). https://bugzilla.mozilla.org/show_bug.cgi?id=653132.

[10] Bugzilla@Mozilla. Bug 786276 - don't autofill passwords in frames that are not same-origin with top-level page. https://bugzilla.mozilla.org/show_bug.cgi?id=786276.

[11] E. Y. Chen, S. Gorbaty, A. Singhal, and C. Jackson. Self-exfiltration: The dangers of browser-enforced information flow control. In *W2SP*, 2012.

[12] Chromium. Issue 163072: Chrome should only fill in saved passwords after user action. https://code.google.com/p/chromium/issues/detail?id=163072.

[13] Chromium. Issue 257156: Don't autofill passwords on page load for iframed content. https://code.google.com/p/chromium/issues/detail?id=257156.

[14] S. Corp. 2013 internet security threat report, volume 18. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf.

[15] J. de Valk. Why you should not use autocomplete. https://yoast.com/autocomplete-security/.

[16] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith. Hey, you, get off of my clipboard. In *Financial Cryptography and Data Security*, pages 144–161. Springer, 2013.

[17] M. Felker. Password management concerns with ie and firefox, part one, 2010. http://www.symantec.com/connect/articles/password-management-concerns-ie-and-firefox-part-one.

[18] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. Xss exploits: Cross site scripting attacks and defense. *Syngress*, 2(3), 2007.

[19] P. Gasti and K. Rasmussen. On the security of password manager database formats. In *ESORICS*, LNCS. Springer, 2012.

[20] B. Gourdin, C. Soman, H. Bojinov, and E. Bursztein. Toward secure embedded web interfaces. In *USENIX Security Symposium*, 2011.

[21] J. Grossman. I know who your name, where you work, and live (safari v4 & v5). http://jeremiahgrossman.blogspot.com/2010/07/i-know-who-your-name-where-you-work-and.html.

[22] J. A. Halderman, B. Waters, and E. W. Felten. A convenient method for securely managing passwords. In *WWW*, 2005.

[23] R. Hansen. Clickjacking. http://ha.ckers.org/blog/20080915/clickjacking/.

[24] J. Hodges, C. Jackson, and A. Barth. Http strict transport security (hsts). http://www.hjp.at/doc/rfc/rfc6797.html.

[25] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: attacks and defenses. In *USENIX Security Symposium*, 2012.

[26] Norton identity safe: Password manager & online identity security. https://identitysafe.norton.com.

[27] KeePass password safe. http://keypass.info.

[28] Secure password manager - keeper password & data vault1password. https://keepersecurity.com.

[29] Lastpass — the last password you have to remember. https://lastpass.com.

[30] Z. Li, W. He, D. Akhawe, and D. Song. The emperor's new password manager: Security analysis of web-based password managers. In *23rd USENIX Security Symposium (USENIX Security 14)*, Aug. 2014.

[31] M. Marlinspike. New tricks for defeating ssl in practice. In *Blackhat DC*, 2009.

[32] PasswordSafe. www.schneier.com/passsafe.html.

[33] R. Gonzalez, E. Chen, and C. Jackson. Automated password extraction attack on modern password managers. Unpublished, Sep. 2013. arxiv.org/pdf/1309.1416v1.pdf.

[34] R. M. Rodriguez. How to take advantage of chrome autofill feature to get sensitive information. http://blog.elevenpaths.com/2013/10/how-to-take-advantage-of-chrome.html.

[35] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. Mitchell. Stronger password authentication using browser extensions. In *Usenix Security Symposium*, 2005.

[36] RSnake. Stealing user information via automatic form filling. http://ha.ckers.org/blog/20060821/stealing-user-information-via-automatic-form-filling.

[37] RunSSL. Ssl certificate for private internal ip address or local intranet server name. http://runssl.com/members/knowledgebase/9/SSL-Certificate-For-Private-Internal-IP-Address-or-Local-Intranet-Server-Name.html.

[38] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *W2SP*, 2010.

[39] R. Saltzman and A. Sharabani. Active man in the middle attacks. *OWASP AU*, 2009.

[40] D. Sandler and D. S. Wallach. input type=password must die. *W2SP*, pages 102–113, 2008.

[41] B. Stock and M. Johns. Protecting Users Against XSS-based Password Manager Abuse. In *AsiaCCS*, 2014.

[42] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. Crying wolf: An empirical study of ssl warning effectiveness. In *USENIX Security Symposium*, 2009.

[43] B. Toews. Abusing password managers with xss, 2012. `http://labs.neohapsis.com/2012/04/25/abusing-password-managers-with-xss/`.

[44] W3C. The autocomplete attribute. `http://www.w3.org/TR/2011/WD-html5-20110525/common-input-element-attributes.html#the-autocomplete-attribute`.

[45] M. Wu, R. C. Miller, and G. Little. Web wallet: preventing phishing attacks by revealing user intentions. In *SOUPS*, 2006.

[46] K.-P. Yee and K. Sitaker. Passpet: convenient password management and phishing protection. In *SOUPS*, 2006.

A Autofilling of forms

Several password managers (Chrome, Safari, LastPass and 1Password) that we studied in this paper also supported autofilling forms with different pieces of information like name, email address, phone no, credit card no, expiry date etc. Even though this is not directly related to autofilling of passwords we summarize our findings in this section for completeness.

Unlike login information, autofill information for forms is not tied to any origin. Therefore, forms from any domain can be autofilled with the same information. To make autofilling secure all the password managers we studied required user interaction to start autofilling of forms. However, several prior works have noticed that a malicious attacker can create specially crafted forms that only have certain innocuous fields visible (e.g. name) while other more sensitive fields (e.g. phone number) invisible to the user and once the user triggers autofilling, both the invisible and visible fields get filled and thus become accessible by the attacker [15, 34].

We found that while all the autofilling password managers we studied are to some extent vulnerable to this attack, the type of sensitive information that can be extracted depends on the nature of user interaction required to trigger autofill. Unlike the rest of the paper in this section we consider web attackers only as the autofill information is not tied by any origin.

- **Chrome & Safari:** Both Chrome and Safari separate the autofillable information into two categories - personal information (e.g., name, email address, phone no., physical address) and credit card information (e.g., credit card no, expiry date). To trigger autofill for each category the user needs to click a field in each category and select an entry from the available ones. Thus, even if an attacker makes a user click a visible field in the personal information category none of the hidden credit card fields will get autofilled. This makes stealing credit information much harder in these password managers without the users noticing it.

- **LastPass:** Unlike Chrome and Safari, for triggering autofilling, LastPass only requires that user click a button shown on top of the page. Once this button is clicked all fields in the form (both hidden and visible) gets filled. This makes it very easy for an attacker to create a crafted form showing only fields like name and email address while stealing additional information, such as credit cards, or a Social Security Number, through hidden fields.

- **1Password:** Unlike LastPass, 1Password requires that the users click different buttons depending on what information they want to fill. Thus, it is not possible to steal credit card information from 1Password by making all credit cards hidden. However, if a legitimate page that a user wants to fill credit card information into also contains an iFrame with hidden credit card fields from a third-party domain (e.g., advertisement), 1Password will fill the credit card information inside the iFrame as well as in the main page with a single click and no notification to the user.