Not Brave Enough to Autofill: Security Analysis of Password Managers Autofill Feature

Mateo Pena Campos
mpenacam@kaist.ac.kr
School of Computer Sciences
KAIST

Guillaume Davy
gdavy@kaist.ac.kr
School of Information Security
KAIST

Abstract—**Password Manager could be the most used authentication mechanism on Internet. But, due to several security issues, it is actually used by few. Additionally, most people trust their passwords and do not see the interest of Password Managers. It is known that common users will create weak passwords, based on human biases. Unfortunately, for now, as precedent works showed, Password Manager lack security in storage, password creation and password autofill. That is what make it difficult to convince people to change their habits and use Password Manager. Anyway, it is unenviable as long as their are to much vulnerabilities in them. In this paper, we will review the main issues of Password Manager and focus on the autofill part. With our own server, we implemented clickjacking attacks when previous papers were only mentioning it. It turned out that Password Managers are more resilient to these attacks nowadays. We especially focus our researches on Brave, a privacy oriented browser. We also analyzed the behavior of Password Manager now compared to how they were doing before precedent works. We found out new default behaviors respecting the advices of previous papers, while, some points are still lacking security. Based on our researches and work, we print a picture of the actual state of the Password Manager domain.**

Keywords

Password Manager, Autofill, Security

1. Introduction

Passwords are known to combine multiple issues and vulnerabilities in the web security. But, as for now, we do not have better options. Several alternative like phone-based or hardware token proved to offer a better security, however password remain the most efficient in its deployability [6].

While some research are actively trying to find new alternatives, other try to find a way to improve the security provided by passwords. For humans, it is hard to use different and strong passwords for every website or application they log in, hence, they tend to use similar or same passwords for every account. Furthermore, human-found passwords lack randomness and therefore make brute force attack (trying multiple passwords until it find the right one) easier.

As a solution, Password Managers (PMs) propose to users the storage and creation of hard-to-guess passwords for them. Thanks to it, people do not need to remember passwords anymore. A different password for each website will be created and then stored. Additionally, we know that it is recommended to change passwords, on average, every 3 months. Which is easier and faster with a PM.

Notwithstanding, as all emerging systems, PM comes to light with its own vulnerabilities and generates new issues. We can summarize PM into three major parts : How does it create passwords ? How does it store passwords ? And, last but not least, how does it fill passwords ?

We will now briefly introduce issues from each parts :

Password generation. PMs need to generate strong and resilient to online and offline bruteforce attack passwords. Precedent work showed that it is not the case every time. By creating numerous random passwords, PM tend to generates some weak passwords among all the others. According to S. Oesch et al. [4], password's length has to be at least equal to 12 to protect every users (so even the ones with weak generated password) from online attacks. While, to defend every customer from offline attack as well, password's length need to be at least equal to 18. It is important for PM to assure a reliable resilience for every customer against online and offline attacks. Whereas, PMs usually do not generate passwords as long as 18. Therefore, this point is a main issue in the password generation of PM.

Password storage. Before PM, we needed to remember our passwords in our head or by writing them down on a paper. PM have to remember your passwords as well. However, on one hand stealing information from your mind is extremely difficult (for now). Whereas, on the other hand, obtaining data stored on servers, clouds or your device (computer, phone,...) is easier. S. Oesch et al. [4] analyses the encryption of different PMs storing password on your device. It appeared that some metadata are not encrypted, opening the door to password robberies. Server storage include all the attacks and lost issues that could face this kind of structure. Using the cloud can also be problematic. As a solution, some papers propose a "mixing storage" [7]. Concretely, on pair with encryption, your data are stored locally and on the cloud. In that case, even if the data on one end is compromised, there is no problem, as both local and cloud data are required to access the database.

Password autofill. When a user goes back to a web site where he has already registered, PMs act differently. Several will ask permission to the user before autofilling the connection forms while other will directly autofill it. This two behavior includes security risk that will be explain in the background (part 3). Based on this vulnerabilities, Silver et al. [2] implemented a threat model, named sweep attack, stealing the credentials autofilled by the PM.

Our project will be presented to you as follow in this paper : Firstly, we will replicate Silver. et al work [2] to see if PMs changed to cover the issues pointed out by this paper. Secondly, on the basis of the precedent step, we will extend out attack to credit card or other information like names, addresses, phone numbers... We will also do a deeper analysis on Brave, known as a privacy-oriented browser. During our research, we found out that it was proposing an integrated PM to its customers. Furthermore, by default, the ”autofilling password” parameter seems to be activated.

2. Motivation

2.1. Attacker Motivation

Attacks on passwords and PMs will increase in the future. Depending on the attack, it can be at the same time cheap and easy for an attacker to operate. Furthermore, obtaining passwords and, therefore, account victim access, open the way to other attacks like information collection, blackmailing, phishing, i.e. an attacker could, with the Instagram account of a student, ask money to his mother by saying that he lost his phone and by specifying that it is an urgent matter.

A little effort for a great reward will always convince new attackers.

2.2. Our Motivation

We want to achieve two objectives with this work. First, as explained before, the password domain has multiple issues since multiple years and PM could be a reliable way to fix several issues. So by finding vulnerabilities and way to protect PMs, we want to improve globally PMs. Doing so, we will, at the same time, improve the security of every Internet users.

Second, we find interesting to see the impact of precedent papers and related work on the subject on the PM field. Did the company listened to the searchers and fixed the problems they pointed out ?

3. Background

3.1. Password Manager

To get to the point, a password manager is a tool that allows a user to generate strong and unique passwords for each website, store the associated credentials and automatically fill them in when connecting to the website. Taking this into account 3 main functionalities stand out:

- **Password Generation.**
- **Password Storage.**
- **Password Autofill**

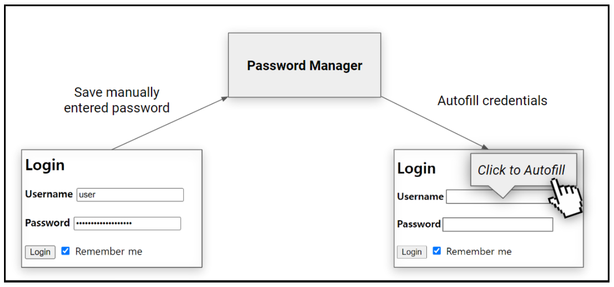


Figure 1. Simplified functioning of manual autofill. User has to click on the pop-up to autofill previously saved logins.

For Password Autofill, the auto-completion is triggered depending on the domain on which the form is detected. That is to say that on a web page `https://site.com`, if a password is saved for this site then it will be associated to each page of this domain. If on `https://site.com/login` there is a login form, then the PM will fill the fields with the credentials corresponding to `https://site.com`. This functionality protect the user from hacker attacks such as keylogging. Two type of autofill exist: automatic and manual autofill (Figure 1). The main difference is that in manual autofilling, user interaction is required in order to fill.

For PM solutions there are actually 3 options possible on the market, and each one have his own set of functionalities:

- **Apps/Desktop clients**
  These PM are application installed on your computer such as KeePass.
- **Browser extensions**
  They don’t rely on a desktop application but are directly integrated into the browser as extensions/add-ons. At the time of this paper the top 5 password manager extensions are 1Password, Keeper, Dashlane, LastPass and RoboForm.
- **Browser-based/Native component**
  Browser such as Chrome, Edge, Safari, Firefox or more privacy-oriented browser like Brave now integrate their own password manager or at least their own autofill feature.

Also the benefits of PM for the user are multiple (1) User don’t have to remember complicated passwords, (2) PM provides different passwords for each website, so it’s more secured in case of a db leakage so that others account are still protected and (3) Size of usual human created passwords are limited (1M) because of the tendency of humans to use words from the dictionary for instance. Whereas password generated by PM don’t follow these

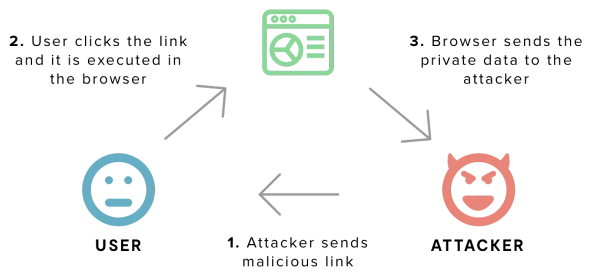

2. User clicks the link and it is executed in the browser
3. Browser sends the private data to the attacker
USER
1. Attacker sends malicious link
ATTACKER

Figure 2. Reflected XSS attack process

conventions which increase tremendously the number of possible passwords ($\approx 94^8$ 8-character passwords) and so decrease the risk of bruteforce.

3.2. XSS attack

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. 3 types of XSS attack exist that can be divided in 2 sides client-side and server-side :

Stored XSS attack. Also referred to as Type-II XSS, user input is stored on a target server, (in a db, forum message, comment on a publication, ...). Subsequently, the victim retrieves the stored data from a web application and sees the user input rendered on his browser. This input could be JavaScript code, which would be executed once the content has been rendered. For example injections of images containing scripts, so when the image will be showed the script will also be executed.

DOM-based XSS attack. Also referred to as Type-0, enables attackers to inject a malicious payload into a web page by manipulating the client's browser environment. This attack is operated on the client-side after loading the page. Since the malicious payload is stored within the client's browser environment, the attack cannot be detected using traditional traffic analysis tools. DOM-based XSS attacks can only be seen by checking the document object model and client-side scripts at runtime.

Reflected XSS attack. Also referred to as Type-I XSS, the workflow consist in making the user visit the attacker's webpage (using social engineering or whatever) and then exploits server-side web app vulnerabilities to execute a malicious script in the victim's browser. One of these vulnerabilities could be a search function that echoes the results. Figure 1. shows a simplified visual illustration of the process.

In this paper we'll focus on the autofill functionality of browser-based and browser extensions PM. We'll look at the security of this functionality and more precisely on how an attacker can retrieve confidential credentials from a malicious webpage using sweep attack.

Prior works showed that autofill functionality from the Firefox's built-in PM is extremely insecure and vulnerable to harvesting attack [3].

We'll replicate earlier work [2] [4] and see if PM autofill is still vulnerable to theft. We'll also look at the new built-in Firefox password manager to see if it follows recommendations and improved its security flaws. Additionally, we'll expand our study to Credit Card No/Expiry Date autofill retrieval and we'll see if the privacy-oriented browser Brave is vulnerable to these kind of attack.

3.3. Sweep Attacks

In the paper of Silver et al. [2], they created a new attack named Sweep Attack to exploit the vulnerabilities from the autofill feature. In there approach, they used XSS vulnerabilities and/or network injections (thanks to a hotspot) to inject login forms in legitimate websites. After, they sent this modified web page containing there malicious form to victims. If the victim is using a password manager autofilling his credentials, the form will be fill out and then send to the attacker. Hence, attackers can steal private information like passwords, email, nickname... But to avoid being detected by the victim, attackers need the attack to be stealthy, that is why searchers thought about sweep attacks. While stealing the credentials as explain before, a sweep attack is also using different methods like iFrame or pop-ups to hide itself.

3.3.1. iFrame. A web page can contain another web page in an iFrame. It is originally a benign JavaScript function to display the content of another web page or ads. However, as the opacity of an iFrame can be so reduced that it is invisible for human eyes. Attackers can use it to hide malicious content. In sweep attacks, it is used to hide the malicious form. With an XSS vulnerability on benignpage.com, the attacker can inject a form and then find a way for the victim to go to the attacker's web page (using phishing email for example) containing the modified benignpage in an iFrame. By doing so, when the victim browse to the attacker's web page, he won't see his credentials being stolen in the 'invisible' iFrame. Silver et al. [2] used multiple iFrame in their experiments to steal credentials from different websites. Their last objective was to make the victim stay long enough on the attacker's web page for all his credentials to be steal. They used loading bar as a reason for victims to wait.

3.3.2. Window Pop-ups. As an alternative to iFrames, Silver et al. [2] proposed to use pop-ups. A pop-up is, usually, a small window or banner that appears in the foreground while browsing a website. When implementing our attacker's website, and hence, our pop-ups, we tried to make them as stealthy as possible. To do so, several techniques exist. By changing the color, we can do a totally white pop-up, less visible, or a pop-up of the same color as our main web page, so it would blend into it. It is also possible to change the size, making it as little as possible. We can chose its place on the screen when it appear, placing the pop-ups at the edge could increase their stealthiness. In the previous paper, they used these methods for sweep attacks. They replaced iFrames by pop-ups, so the vulnerable web page containing malicious forms were displosed in the pop-ups.

3.4. Clickjacking

As defined by the website OWASP [8], a recognized company in the web security domain. Clickjacking, also known as a “UI redress attack”, is when an attacker uses multiple transparent or opaque layers to trick a user into clicking on a button or link on another page when they were intending to click on the top level page. Thus, the attacker is “hijacking” clicks meant for their page and routing them to another page, most likely owned by another application, domain, or both.

4. Attack Model

For our experimentation we assume an attacker that can inject JavaScript into the website (by Network Injection or XSS). More precisely, we assume that the website contains at least one vulnerable webpage (HTTP, Broken HTTPS or with an XSS vulnerability). We consider stolen credentials are extracted with whichever known method (GET request to attacker website, email, ...). We made these assumptions to help us to only focus on the behavior of PMs autofill and not about the context. These points have already been addressed in previous researches [1] [2] [3].

5. Vulnerabilities and Exploits

Previous researches [2] [3] have shown that when PMs autofill login forms automatically, passwords can be retrieve easily as long as JavaScript can be injected into the webpage. As autofill is based on the website URL and often no interaction is required to activate the process, this feature becomes a vulnerability for user passwords. Indeed, Silver et al. [2] presented the first attack exploiting this vulnerability. Called Sweep Attack, it exploits the fact that for most PMs, no interaction is required to autofill the form entries. By combining this with iFrames from websites that allow interactions between different origins (i.e. with bad Same Origin Policies), they were able to retrieve passwords from several websites while the user only visited one malicious webpage and did not even realise the operation. This vulnerability is very critical, as it allows to recover a large number of passwords in a very short time and in a very stealthy or even undetectable way.

In this section, we have analyzed the behavior of the autofill among the multitude of PMs available on the market to see if they are still vulnerable to the sweep attack presented in 2014 by Silver et al. [2]. We mainly focused on built-in PMs in Chrome, Firefox and Brave a security and privacy oriented web browser, as well as PMs in the form of web browser extensions like LastPass, RoboForm and BitWarden (All in the Chrome Web Store Top 5 most used password managers, at the time this paper was written).

5.1. Current state of autofill in PMs

To analyze each of these password managers, we went about it by hand, creating our own vulnerable website and pretending to be an attacker who could inject a script containing a form. All the results of our analysis are summarized in Table 1.

We first created a basic website containing one webpage with a login form inside. This webpage is supposed to be where the user saves his username and password in the PM. We host this website locally using the *http-server* npm package. This is a simple, zero-configuration command-line static HTTP server but hackable enough to be used for testing, local development and learning. We also used the *mkcert* npm package to create self signed TLS certificates without OpenSSL. This allowed us to set up HTTPS webpages in a simple and efficient way. Next, we created a webpage in the same domain as the previous containing a simple XSS vulnerability. We chose to make a reflected XSS attack possible by echoing the content of the URL parameter. For example with this URL: `http://notbraveenough.com/s?var=autofill`, the webpage will echo ”*autofill*” somewhere in the page. To make script injection possible by this way, no sanitization is done, of course this is a behavior to avoid but there are still a lot of developers who forget to secure user input like this, making this attack realistic. We remind you that our goal here is not to focus on how the attacker manages to inject the malicious script, but rather to analyse the behaviour of the autofill against an attacker who has already succeeded in this first step and to draw some conclusions to secure this feature. The exact script that is injected into this vulnerable page is a simple login form (Listing 1). The autofill feature will fill it in when the necessary conditions are met, which depend on the PMs used.

Our workflow was simple, first we went to the page containing the login form and then we registered our credentials in the PM we wanted to study. There are several ways to do this. It is important to note that the logins are never saved automatically (i.e. without user interaction), regardless of the analysed PM. For built-in PMs, an icon (usually a key) appears in the search bar to indicate that credentials can be saved. By clicking on it, a pop-up appears and we are offered to save the logins entered in the inputs. For PM in extensions, several methods apply but usually the PM icon appears around the inputs to alert the user that they can save their password. As with built-in PMs, it only takes a few clicks to do so. However, Bitwarden opts for a more secure method by asking a click on the extension icon, at the top right of most browsers, and after a series of clicks and validations the password can be saved. We will see later on why this method presents less risk of logins being stolen, especially with regard to clickjacking.

The second step is to go to the web page that is vulnerable to XSS attacks. We do this by clicking on a trap link containing HTML code and a script in the URL parameters in order to inject a form that should trigger the autofill and a way to extract data from the input. It is during this step, once the logins are saved in the PM and the form is injected in the web page that we can analyze if the autofill is done automatically or manually, and thus know if the PM is vulnerable to sweep attacks.

Like Silver et al. [2], we have chosen to analyze the behavior of autofill in relation to the protocol used when saving the password and when it is autofill. We have studied 3 patterns:

1) **Same secured protocol.** Password saved on an HTTPS webpage and attack performed on a malicious HTTPS webpage containing the form
2) **Same unsecured protocol.** Password saved on an HTTP webpage and attack performed on a malicious HTTP webpage containing the form
3) **Different protocol** Password saved on an HTTPS webpage and attack performed on a malicious HTTP webpage containing the form

By applying this workflow on each of the PMs we wanted to study, we came to the following observations:

When the password is saved on an HTTPS webpage and in the same domain you go to another HTTPS webpage (case 1) that contains a form (i.e. in our case, the malicious webpage), most of the PMs ask for an user interaction to fill the input. This is the case for Firefox, Brave, Bitwarden and RoboForm. For Chrome, the autofill behavior is slightly different. It appears that visually, the autofill was successful and the username and password fields were filled in. But in reality, when the script to retrieve the password located in the corresponding field runs correctly, no data is retrieved. The field is considered empty. It is enough to click anywhere on the web page for it to be filled. A simple ”focus” on the page triggers the autofill. We don’t know if this is the desired behavior of the developers, but in any case it is enough to prevent a script from retrieving the logins without any interaction. On the other hand, asking the user to make a single click anywhere to activate the trigger is within the reach of attackers. A simple first ”loading” page that requires the user to click on a button to access the page’s content can do the trick. Or simply, a pop-up in the site asking how cookies should be handled can generate an easy click from the user. Without making it too visible and reducing the stealth of the attack. LastPass, on the other hand, requires no user interaction for autofill. As soon as the web page is loaded, the logins are already filled in the fields. Our script manages to retrieve them without any interaction being requested. This behavior is very vulnerable to sweep attacks and should be avoided.

When the protocol is now only HTTP (case 2), for save and for attack. A user interaction is required by almost all PMs. Chrome on the other hand, uses the same behavior as when the protocol is HTTPS, that is, it requires a click anywhere on the page to fill in the fields. Again, this makes the sweep attack less effective because it can’t be done in the background without any interaction being requested, but it’s still very easy to force the user to click somewhere without them suspecting anything.

```
<form action=”action_page.php” method=”post”>
<input type=”text” name=”uname” required>
<input type=”password” name=”psw” required>
</form>
```

Listing 1. Matrix multiplication pseudo code

As for the change of protocol between password saving and attack (case 3), Chrome, Brave and LastPass simply do not make autofill possible. In fact, this is because they base their autofill method on the url of the initial password saving domain. That is, when saving the password on `https://site.com/login` and visiting `http://site.com/sweepattack`, these PMs do not consider these 2 web pages as part of the same domain because of the difference on the protocole used, making the credentials/website association impossible and therefore not triggering autofill.

We also wanted to explore the behavior of the autofill when the malicious form is located in an iFrame or a Pop-Up window. This being the main channel used in [2] to perform the sweep attack, many PMs have adapted their policies for this kind of situation. All the PMs studied now require user interaction, either by pop-up displayed near the field or by accessing the extension in the top right corner of the web browser. If the protocol of the page including the iFrame is HTTP, LastPass and RoboForm will even display an alert in the web browser to let the user know that the protocol used on the main page is not secure. In Chrome, when the form to fill is in an iFrame and the protocol used in the main page is HTTP, an alert is displayed when the user wants to autofill on a non-secure page. Thus, the sweep attack using iFrames now seems very difficult to implement without arousing the user’s suspicion.

6. Experimentation

In this section, we will detail our attempts to exploit the vulnerabilities highlighted in the previous section. The idea proposed in Oesch et al. [3] led us to try to force the user to interact with the autofill pop-up, while remaining stealthy and unnoticed by the user. We thus present the first, to our knowledge, attempt to implement an attack on the autofill functionality using clickjacking. We then explored the possibilities offered by iFrames and Pop-up windows to try to reproduce an attack as in Silver et al. [2] or Oesch et al. [3]. The data that can be stored and saved in PMs are more and more numerous and varied. In particular, it is possible to save the user’s credit card information and to autofill it when necessary (when purchasing on e-commerce websites for example). We have thus tried to extend the scope of the attack by targeting other data to be stolen using autofill. Finally, we have briefly explored the autofill behavior of mobile PMs (especially built-in PMs).

Password Manager	HTTPS-HTTPS	HTTP-HTTP	HTTPS-HTTP	iFrame/Pop-Up
Chrome	Autofill (with a single click)	Autofill (with a single click)	No Autofill	User Interaction
Firefox	User Interaction	?	?	?
Brave	User Interaction	User Interaction	No Autofill	User Interaction
LastPass	Autofill	User Interaction	No Autofill	UI + Warning on HTTP
Bitwarden	User Interaction	User Interaction	User Interaction	User Interaction
RoboForm	User Interaction	User Interaction	UI + Warning on HTTP	UI + Warning on HTTP

Table 1. Summary of autofill behaviour for the studied password managers. UI = User Interaction. HTTPS-HTTPS means that the logins were recorded on an HTTPS page and that the autofill was tested on a malicious HTTPS page. The same goes for HTTP-HTTP. For HTTPS-HTTP it means that there was a protocol change between the save and the autofill.

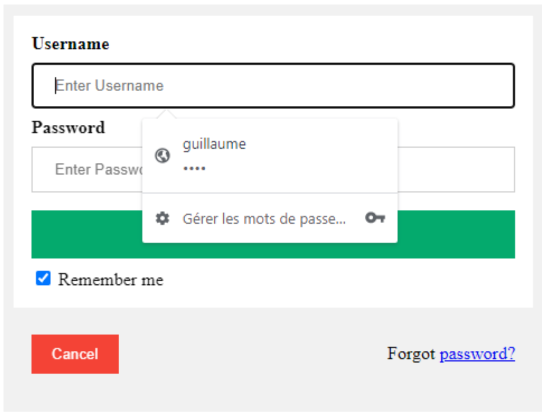


Figure 3. Pop-up asking for UI with Brave PM

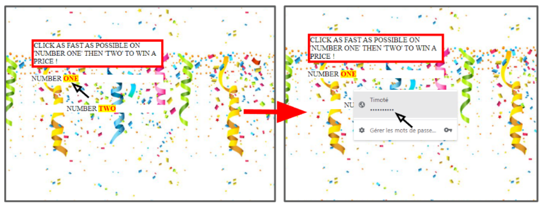

CLICK AS FAST AS POSSIBLE ON 'NUMBER ONE' THEN 'TWO' TO WIN A PRICE !
NUMBER ONE
NUMBER TWO
CLICK AS FAST AS POSSIBLE ON 'NUMBER ONE' THEN 'TWO' TO WIN A PRICE !
NUMBER ONE
NU
Timoté
••••••••••
Gérer les mots de passe...

Figure 4. Double Moving Clickjacking

6.1. ClickJacking

On the contrary as what we supposed at the beginning of our research, Brave's password manager is not blindly autofilling forms. Through our test, on our website, we discovered that, even with the 'autofill' parameter activated in Brave, we needed user's interaction to fill in forms. Therefore, we tried to use clickjacking to bait users, as precedent work [3] suggested. However, how Brave is requesting user interaction is annoying for clickjacking attackers.

As you can see on figure 3, user has to click on the field to activate the Brave's pop-up proposing the autofill. And, additionally, he needs to move his mouse to the pop-up and then click on it for Brave to fill in the form. Hence, an attack using clickjacking is unlikely to work. The most suitable way we find was to use a double clickjacking, asking the user to quickly click on two different place on his screen. We can see the proposition design on the figure 4. However, we do not think every user will genuinely accept to 'play' along, hence, we conclude that Brave's password manager is resilient facing the autofilling issue. Furthermore, as the pop-up is linked to Brave and out of the webpage, the opacity of the iFrame can not make it invisible. We could not find any possible way to hide this pop-up or to dissimulate it, hence, Brave's password manager is reliably protecting its users against this attack.

Explanation of our attack on brave (Figure 4) : We trick the user into a game where he is supposed to click on two numbers one after another as fast as possible. While he thinks he will click on "ONE", he is actually clicking on another website, in an iFrame behind our top website. So, he is actually clicking on the field of "password" or "email" or "username" on a legitimate form behind. Then, he will quickly move his mouse to the number "TWO" and click. But, exactly at the position of number "TWO" will pop out brave's pop-up, asking the user if he wants to fill in his information (password, username, mail...). The user might see the pop-up, but if he was really intending to pass the challenge, the time he realises it, it will be to late as he should have already click on the pop-up, accepting at the same time, to fill in his information. Then, as for every attacks, we will force the submission of the form and send it to ourselves, with the credentials of the user.

One weakness: an user agreeing on 'playing' to this little game seems unlikely. We could offer a prize depending on his speed (i.e. the last iPhone). Whereas, it would still remain suspicious.

6.2. iFrame

Using our server, we implemented a web page as follow: a benign page containing, in an iFrame, another page from our website with a form asking for username and password. All password managers accepted to fill in the iFrame. However, they asked for a user interaction. User interactions needed by the other browser password managers are similar to Brave's one when the form is in an iFrame from another origin.

With an iFrame from the same origin, browsers password managers tend to autofill as if they were in an HTTPS-HTTPS case (if the page is in HTTPS).

It turned out that most of the internet refuse to be in an iFrame. Companies like Facebook or Google refuse to appear in an iFrame, rending the iFrame sweep attack useless on them.

Nowadays, based on what we discovered, a sweeping attack based on iFrames seems unlikely to succeed.

6.3. Window Pop-up

To open the way to sweep attacks on brands like Facebook or Google, we need something else than iFrames. A good other method is using window pop-ups. Unfortunately for attackers, most of the time, browsers are blocking pop-ups. In the previous paper [2], to counteract it, they assumed that they could convince the victim to deactivate the parameter blocking pop-ups in the browser. As they were hosting a hotspot, they could request it to the user in a procedure for him to access the internet.

However, as we did not want to use a hotspot and as we wanted to implement new things, we decided to use the other way to permit pop-up creation : user interaction. Indeed, if the pop-up is generated from an user interaction, then the browser will allow it. Hence, we decided to use clickjacking once again.

By tempting the victim on clicking on a link, we can open a little window, as stealthy as possible on his screen. We tried to use the blur() and focus() function from JavaScript to hide the pop-up. But it seems to be blocked by every browser we tested.

Therefore, we tried to open two pop-ups at once : the first one was the malicious one, stealing credential's victim, and the second one was another benign web page of the size of the screen, supposed to hide the attack.

A new disappointment for attackers, this method can not work because, for browsers, one user interaction is equivalent to one pop-up. Meaning that with just one click from the victim, we can only open the malicious pop-up and not the second one. As asking for multiple click from the victim seems more suspicious than a little pop-up, we decided to not go further. Doing only one pop-up as stealthy as possible is the better vector of attack for sweep attacks using window pop-ups.

6.4. Stealing credit card details

Most password managers also propose to store and autofill various other information such as name, addresses, phone number or credit card no. Some, like RoboForm, offer to create profiles containing information such as income, social security number, passport information, etc. All of this information, if it were to be misappropriated, represents a goldmine of information for targeting users for advertisements for example, but can also be subject to malicious behaviour or surveillance. This type of information is important to protect.

The autofill feature is also available for most of this other type of information, so we analysed the behaviour of password managers by reproducing the same workflow as for passwords. The objective is to get an overview of the current state of autofill of so-called 'personal' data rather than logins. We chose to focus our experimentation on credit card details.

In built-in password managers, credit card number can be saved manually and automatically after validation on the pop-up that appears when you submit a first form with credit card details. When saving information manually, it is not possible to save the Card Verification Code (CVC). The user will have to fill in this field of the form manually. This is a secure behaviour as without this 3-digit code it is not possible to pay online with it. On the other hand, it reduces the usability and the interest of auto-filling the card although it reduces the amount of information that the user has to remember to 3 digits. The 3 other PMs studied, which are web browser extensions, allow to save and autofill this CVC number. This choice allows the user not to have to take out his credit card to enter the CVC or to memorize it. This can be an advantage, but we think that asking for at least the 3 digits is a more than important security measure, since they are originally there to secure the transactions and make sure that it is the cardholder who wants to make a purchase.

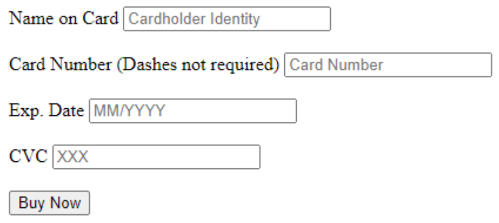


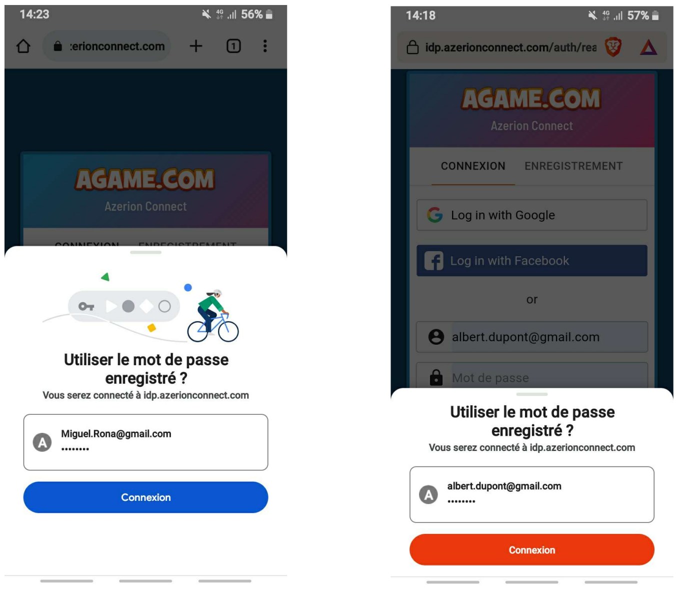
Figure 5. Payment form used to analyze the credit card information autofill

To determine the autofill behavior of these PMs with respect to credit card information, we have created a web page containing a simple form used in e-commerce transactions (Figure 5). This form contains 4 fields: cardholder identity, credit card number, expiration date and CVC. Using a workflow similar to the one used in the previous analysis of autofill logins, we came to the conclusion that with respect to credit card data, PMs are much more restrictive in terms of usability. Indeed, they systematically require user interaction. Whether it is the built-in PMs or the extended PMs, aach of them in its own way ask for user interaction: a pop-up that appears near the inputs or by accessing the interface of the extension in the top right of the web browser.

Whatever the protocol used (HTTP or HTTPS) to autofill this information, user interaction is required.

Again, some of these PMs choose to display an icon or pop-up near the inputs to click on to trigger the autofill. As seen previously with passwords, this mechanism is vulnerable to clickjacking. A skilled attacker could find a way to get the user to click without them realizing it and then activate the sweep attack to retrieve the information.

We would like to point out that all the PMs studied here require user interaction and are therefore less vulnerable to the exploitation of the autofill feature. We therefore wondered why this is not the case for information such as the password, isn't this information just as important to secure? PMs are able to propose sufficient security against sweep attacks but decide arbitrarily according to the type of information to choose one autofill mechanism or another (automatic or manual).



Utiliser le mot de passe enregistré ?
Vous serez connecté à idp.azerionconnect.com
Miguel.Rona@gmail.com
••••••••

Utiliser le mot de passe enregistré ?
Vous serez connecté à idp.azerionconnect.com
albert.dupont@gmail.com
••••••••

Figure 6. Screenshots of the UI pop-ups asked by respectively Chrome and Brave on mobile devices in a HTTPs-HTTPS context

6.5. Mobile Browsers

Precedent work like Oesch et al. [4] showed the lack of security of mobile compared to desktop in the field of PM. They turned out to be really vulnerable in most of the cases.

For example, in Credential-to-Domain Mapping, according to the searchers [4], all PMs on mobile are worst than the worst one on desktop. In fact, they do not change their behavior depending of where you registered and where you log in. If you registered under HTTPS, PM on mobile will not forbid autofill if you try to log in under HTTP.

However, mobile PMs turned out to be more secure when it comes to user interaction asked before autofilling. In the context of HTTPS to HTTPS, on the browser password managers we tested, Brave, Edge and Chrome systematically request for user interaction with a pop-up. On mobile, the pop-up take more than half of the screen (Figure 6), which is more noticeable compared to pop-ups on computers. Hence, for sweep attacks requiring stealthiness or clickjacking, it become much more unlikely to succeed. On the other hand, we found out in our test that Firefox and Opera are autofilling passwords without user interaction, opening the way to sweep attacks.

7. Discussion

7.1. Limitation

In this part we will summarize the different difficulties and limitations we faced during our project.

Server limitations: Our local server might differ from real servers on the web. Consequently, the behavior of PMs we observed during our test might differ from their real behavior. We tested several experiments on real web pages like www.jeu.fr and we relatively observed the same behavior we could have on our server.

As we implemented our own server, we created our own certificate to upgrade it on HTTPS. But, for a reason out of our knowledge, on computer devices, Firefox never accepted our certificates while other browsers did. Therefore, we could not conclude on the real behavior of the Firefox PM on our server.

XSS limitations: It is illegal to use JavaScript injections in South Korea. Thus, we could not try our sweep attacks on real cases. It can results in false assumptions, we may thought something could work because it did on our server while on real servers and web pages it would not.

As the best of our knowledge, XSS injections should not change the behavior of PMs. Whereas, as we could not try it in real cases, we can only do assumptions about it.

7.2. Suggestion

We would like to suggest good security behavior to use with PMs and more precisely with the autofill.

As our analysis highlighted well, when PMs ask for user interaction to autofill the user logins, this makes the data more difficult and much less stealthy to steal (e.g. double clickjacking). By multiplying the number of clicks required to effectively autofill, it makes this feature much less vulnerable to attack without degrading its usability too much. Some users will find it annoying to have to click for each connection, but 2 clicks are enough to avoid any theft and it remains very fast and usable.

8. Related Work

Our paper is not the first one addressing Password Manager's issues. We did a work of research on previous related papers and we will know describe them in this part.

Autofilling : As said before, we principally based our work on the paper of Silver et al. [2]. They were the first to use the autofilling feature to implement the sweep attack. They put the light on a huge PM vulnerability. But other paper like the one from Oesch et al. [4] addressed the autofilling.

Storage : Several papers discuss issues related to password storage. Oesch et al. [3] showed that the passwords are stored, depending on the PM, in the cloud, in servers or in the user device directly. Each of this solutions have their own threat and vulnerabilities. Hence, passwords can be stole, which would be a disaster for customers.

Kumarakalva et al. [7] proposed a method for storage: using the different solutions at the same time. Your passwords could be stored in the cloud and on your device at the same time. Your supposed to need them both because the encrypted data from your device and the cloud need each others to form your password. Therefore, even if your device is compromised by an attacker, he won't be able to decipher your passwords as he will also need the data from the cloud.

Password Generation : How strong passwords are created can also be an issue in case of bruteforce attacks. Oesch et al. [3] found out that, by generating passwords randomly, PM would create weak passwords eventually. As PM should protect every customers with good passwords, they can not afford to create weaker passwords. To ensure the validity and the resilience of every passwords against every kind of bruteforce attacks, the searchers concluded that generating passwords of at least a length of 18 characters was necessary.

XSS Injection : JavaScript injections are well-known and at the same time well-used. Other papers like Stock et al. [1] investigated it specifically in PM context. They give a comprehensive overview on potential XSS based attack patterns on browser-provided password managers and explore potential mitigation strategies.

9. Conclusion

In this paper, we compared the state of security in password manager at the time of the previous paper of Silver et al. [2] with the current one.

We reproduced sweep attacks on our implemented server to figure out the behavior of PM. By using XSS injections, we could steal credentials with some PM while other were more robust and needed new features on the attack, like double moving clickjacking. However, it seems unlikely to succeed in a real attack.

Nowadays, autofill functionality is mostly manual autofill unlike a few years ago. Progress has been made to make autofill as easy to use, non-binding and above all safer. Unlike what we thought at the beginning of our researches, when we saw the autofilling features activated by default on Brave, autofilling does not mean without user interaction. Finally, we think Brave password manager is secure as it is asking for a user interaction every time. Furthermore, a user interaction hard to bypass for attackers.

We briefly investigated other security topics in PM, like storage or password generation. We leave further improvement on this subjects for future works. As all PM do not do password generation, it is still important for customers to know how they should write their passwords (number of characters, usage of different character's type, etc.).

References

[1] Stock, and Johns. *Protecting users against XSS-based password manager abuse.* Proceedings of the 9th ACM symposium on Information, computer and communications security 2014

[2] Silver, David et al. *Password Managers: Attacks and Defenses.* USENIX Security Symposium, 2014.

[3] Oesch, Sean and Scott Ruoti Ruoti. *That Was Then, This Is Now: A Security Evaluation of Password Generation, Storage, and Autofill in Browser-Based Password Managers.* USENIX Security Symposium, 2020.

[4] Oesch, Sean et al. *The Emperor's New Autofill Framework: A Security Analysis of Autofill on iOS and Android.* Annual Computer Security Applications Conference, 2021.

[5] Li, Zhiwei et al. *The Emperor's New Password Manager: Security Analysis of Web-based Password Managers.* USENIX Security Symposium, 2014.

[6] Bonneau, Joseph et al. *The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes.* 2012 IEEE Symposium on Security and Privacy, 2012.

[7] Kumarakalva, Yasaswi and Vidhya Shankar. *Password Managers: A Survey.*, 2016.

[8] Gustav Rydstedt. *OWASP Clickjacking attacks*, Last modified, Feb 2022. https://owasp.org/www-community/attacks/Clickjacking.