CSE 509 – System Security
Project Report

SECURITY ENHANCING BROWSER EXTENSION

Team – The_Nights_Watch

Anandh Varadarajan
Veerendra Ramesh Kakarla
Nishant Kulkarni

REPORT SECTIONS

I. Overview

Passwords are means by which a genuine user proves that he/she is authorized to access a device/system/object in general. Using the same passwords across accounts is one of the ways to become vulnerable to an attack. If an attacker cracks the password of a single account, he could potentially gain to access to all other accounts using the same password. Consequence of such attacks leads to access to your personal information, and thus you could become the victim of identity theft. Therefore, passwords are critical to ensure privacy and security of your systems/accounts.

Also, security researchers have argued that most users would be protected if software would stop them from visiting unpopular websites. We have addressed these concerns in our project.

In this project, we have developed a Chrome browser-based browser extension that will encourage users avoid password reuse, prevent them from phishing attacks and generally help them browse through the web content more securely.

Our extension covers the following use cases:

1) **Detecting Password Reuse**

Given, the user has installed Chrome and the plugin for the first time and has no previous account setup yet,

When, the user is trying to create an account / sign up in a site

Then, as soon as the user repeats a password he has already used, we warn him and ensure that he enters a unique password. If it was unique in the first place, user proceeds to setting up the account, without any warnings.

2) **Detect the entering of Passwords on the wrong website**

Given, the extension has convinced users to chose unique passwords across their accounts,

When, the user tries to login to a web site/account

Then, if the user enters the password of an account that is different from the account he is currently in, then we immediately alert the user and protect him from falling victim to phishing attacks.

3) **Secure link clicking behavior**

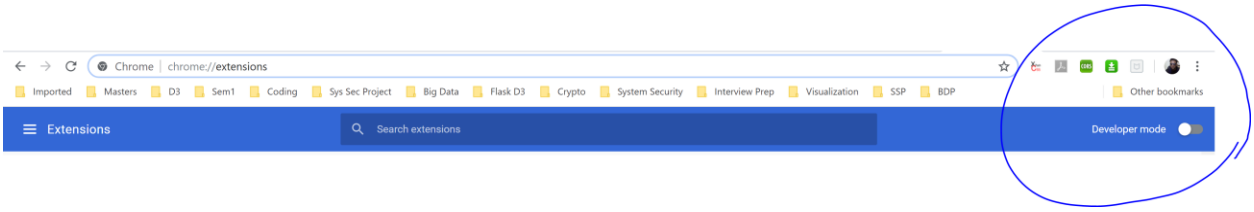
Given, the user is currently in a web page with many links to other pages/sites

When, the user clicks on a link

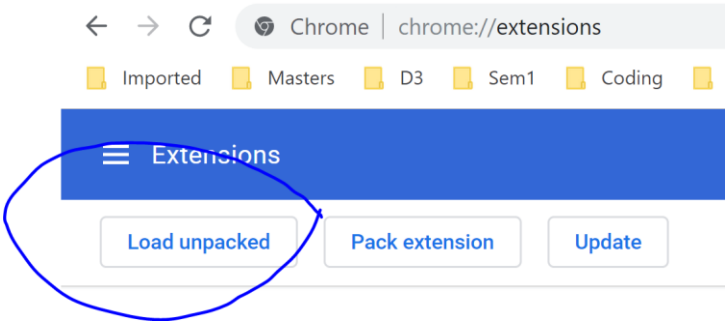
Then, If the domain of the web page/site is different from the recommended list of websites, we warn the user against visiting the site and leave the choice of proceeding to the site, to the user. We have also provided the option to dismiss the warning once or forever.

II. Setting up the Project

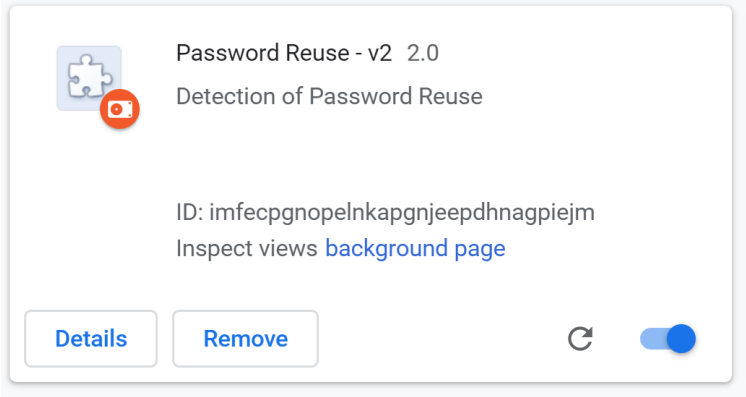
1. Pull the entire source code from – https://github.com/avaradarajan/SecurityEnhancingBrowserExtension
2. Unzip the folder and place it anywhere in your system.
3. Open your chrome browser profile and enter **chrome://extensions** in the address bar.



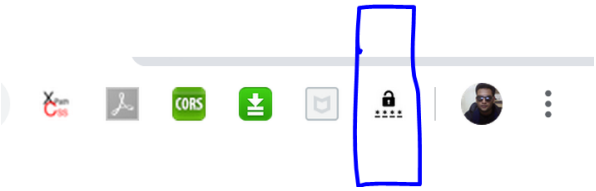
4. Enable the Developer mode option on the top right corner.
5. Once you enable the developer mode, there will be options to load the extension. Choose “Load Unpacked” button as below.



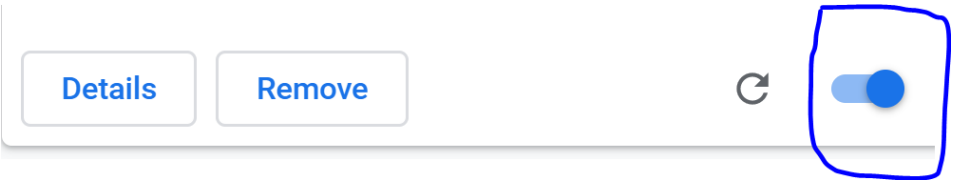
6. In the file explorer, choose the parent folder of the repository you just downloaded. i.e. the immediate parent folder where your manifest.json and other extension files are packed.
7. In our project, it is the Final folder that you have to choose and proceed.
8. Once you load the folder, you should get the following plugin block in the page.



9. Click on Details (shown above), and enable "**Allow in Incognito**" if you want the plugin to even work In the Incognito mode.
10. Setup is complete. Have a secure browsing experience.
    You should be getting an icon similar to the below added to the plugins section.



11. You can also choose to disable the plugin by clicking on the toggle highlighted below.



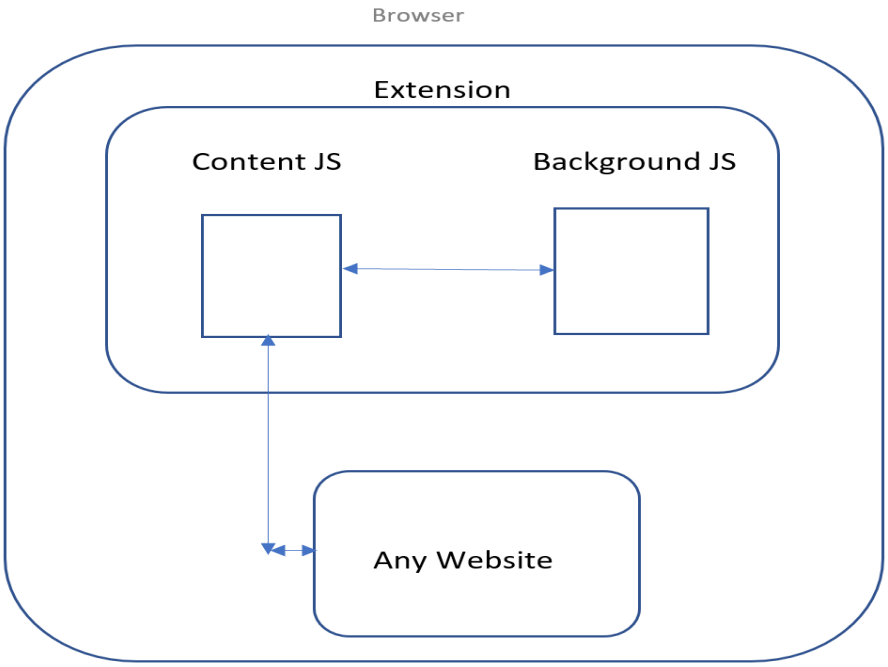
III. High-level Architecture/Implementation Details

Terminologies/Concepts

Content JS - Content scripts are files that run in the context of web pages. By using the standard **Document Object Model** (DOM), they are able to read details of the web pages the browser visits, make changes to them and pass information to their parent extension. It interacts with background scripts using message passing. For more information regarding content script, please check

https://developer.chrome.com/extensions/content_scripts

Background JS – A background page consists of the core logic of the extension. This part of the extension cannot interact with a websites DOM directly, but has access to Chrome APIs that can be used to perform any logic and indirectly control the web page DOM using the content script. This page is loaded when needed and unloaded when not used. For more information regarding the background page, please refer to - https://developer.chrome.com/extensions/background_pages.



Content Script logic for use cases 1 and 2:

1. As soon as the DOM loads, we query all the forms in the current page which has an input element with type password, as a collection. This generally means that the page has one or more forms that is displaying either a SIGN UP / SIGN IN form, apart from possible other forms like forgot password.

2. We then iterate through all forms collected and add an event listener to the first password field of each form (if multiple password fields present).
   - The design decision behind adding listener to only one password field is with the general assumption that the password field of any form is a required field and hence the user cannot proceed without filling it at least once.
   - Also, the rationale behind adding events to password fields is to later identify the form, user is currently interacting with, dynamically.

3. The event listener added is 'blur' event which will get triggered as soon as any password field, goes out of focus. The event is associated with a callback function (getFormDetails here) that will get the password entered and do further processing.

4. Assuming that the callback of blur event is triggered, we start with dynamically identifying the form the user is currently interacting with using the closest() function.
   - For example, **activeForm = event.srcElement.closest('form')**

5. Once we identify the active form, we then use the querySelector() function to get <input of type submit> or any <button>. The idea to get these buttons is later detect if the active form deals with the signup or signin user action.
   - The design decision behind getting both <input type='submit' …> and <button> is to support a wide set of sites. As we have seen sites, that do not use 'input type submit' and use button with normal click action to achieve form submission.

6. Based on the type of element present (input/button), we then get the label using innerHTML.

7. Then we check for possible texts that could be uniquely associated with identifying the SIGNUP / SIGNIN page.
   - For example, based on the analysis of popular public websites, SIGNUP forms have the following labels in their submit element – “SIGN UP” / “SIGNUP” / “REGISTER” / “CREATE ACCOUNT”.
   - SIGNIN forms have the following labels in the submit element – “LOGIN” / “LOG IN” / “SIGN IN” / “SIGNIN”

8. This is our primary level of form detection. If the forms do not fall in these categories, we then fallback to a more generic form detection scheme:
   - If number of password fields In the form Is 2 and the number of all visible input fields in the form section is more than 3, then we identify the page as ‘LOGIN’.
   - If number of password fields In the form Is 1 and the number of all visible input fields in the form section is less than equal to 3, then we identify the page as ‘SIGNUP.

9. The idea behind using the above scheme is the following:
   - Generally, sign up forms have two password fields, one for entering the password we want to use and another for reconfirming the password we entered. Login forms generally have just one password field to login.
   - Also, the number of input fields in a login form can be at most 3, i.e. Username/Email field, Password Field and the Submit button.
   - The number of input fields in a signup form will be more than three i.e. apart from the Username/Email and password fields, it will have other fields that would like to capture more initial information about the user like, Date-Of-Birth, FirstName, LastName, Mobile Number etc..

10. We also detect the username field as the field before the first password field of the form. This decision is also from analyzing more than 20 popular websites.

11. Once we identify these aspects, we add event listeners to the submit('submit' event)/button('click' event) elements, so that we can store the credentials users entered while creating accounts.

12. Based on the page type detected, we use the chrome sendMessage API to call our background JavaScript file where the main logic to detect password reuse for use case 1 and 2 resides.

13. We capture the response from the background JavaScript file and warn the user if reuse is detected. Based on the response, we also decide when to store the user credentials

    Note – When user re-enters an already existing password in **sign up form**, we show the alert irrespective of the domain. But, If user re-enters an already existing password in the **login form**, we show the alert only when current domain does not match with stored passwords' domain.

Content Script logic for use case 3:

We have tried to handle the primary ways in which the user tries to open a link in the current web page.

I. Click on link directly, loads the new content/page in same page
II. Ctrl + Left click, loads the page/content in new tab.
III. Right click -> open link in new tab

For Type I and II.

1. We add the click event listener for all <a> elements in the current page.
2. We then prevent the default click operation of opening the link using event.PreventDefault().
3. We then extract the complete domain name of the site he is intending to visit, check if the domain matches with any of the top 10K Alexa Websites.

4. If it matches, we load the link normally, else we raise an alert to the user and wait for his response to our warning.
5. User can choose one of the following buttons in the alert box:
   - He can click on 'PROCEED' to visit the site once. Next time if he tries to visit the page with blacklisted domain, he will be warned again.
   - He can click on 'ALWAYS ALLOW' to notify plugin that it should not throw an alert from next time onwards for that specific site. We then add his preference to the browser's local storage.
   - He can click on 'CANCEL' to stop visiting the site.

Note – We have handled the case where href attribute has relative paths rather than complete URLs.

For Type III.

1. We add the contextmenu event for all <a> elements in the current page.
2. Steps 4,5,6 from the previous section repeat

Background Script Logic for Use case 1 and 2:

We have used the Chrome onMessage listener to listen to events from content scripts.

We are also using the Chrome local storage to store the credentials and the Alexa sites with user preferences.

Assuming the user entered the password in sign up form:

1. We first create a salt for the user using the cryptoJS library and store the salt in the local storage. Salt is created only the first time.
2. We then create a password list for the user in the JSON format, similar to the one shown below. List is initiated only the first time.

   "passwordList":[
   {"domainName":{username:<name>,password:<pwd>}},

{"domainName2":{username:<name2>,password:<pwd2>}}
..
..
]

3. First, we use the PBKDF2 scheme to generate the hashed password of the password value currently entered. We use SHA-512 along with salt and do hash stretching for 10 iterations.
4. Then we iterate through the local storage to check if the hashed password matches with any of the stored hash password.
   - If there is a match, we send the response back to content script indicating that the user has to be alerted.
   - If there is no match, we send the response back to content script indicating that the current set of credentials is good to be stored when user clicks on submit.
5. When user finally clicks on submit, we again iterate through our local storage and add/update the password list.

Assuming the user entered the password in the login form and we convinced users to enter unique password for any account:

1. We use the PBKDF2 scheme to generate the hashed password as above.
2. We get the current domain where user is in, and verify the password entered with the stored password.
   - If they match, the user continues without any warning.
   - If they do not match, we again iterate through the other passwords and check if there is a match. If yes, we send the response to raise an alert.

Background Script Logic for Use case 3:

1. We first load the whitelisted sites from the Alexa recommendations into the local storage. We do this load operation only once i.e. when the extension is first installed or the extension version changes. We use chrome.OnInstalled API for this.

2. Once the background script receives a request to check if a link clicked by user is a part of the whitelisted set, it checks the local storage with O(1) access using domain name as key.
   - If there is a match, we send the response back to ensure user is not blocked.
   - If there is no match, we send the response back to content script asking to raise an alert. Based on the user action, the content script can again interact with background script to store user preference on "ALLOW ALWAYS" case.

IV. Technologies/Libraries Used

- JavaScript / JQuery – For use cases
- Papa Parser library – For parsing Alexa file
- SweetAlerts.js package for alert popups
- Libraries in CryptoJS for Hashing passwords
- Chrome APIs
- Chrome localStorage

V. Additional cases handled:

1. We have captured all methodologies of user click action for use case 3, i.e.
   - Normal mouse click
   - Control Key + Left mouse click
   - Right click and Open from Context menu
2. We have handled cases where the website is trying to redirect to one of its own pages i.e. href is '#' or any relative path. Interesting case is If user enters a blacklisted site in address bar, currently we do not block it. But if user then proceeds to click any relative link within that block listed site's page, we throw the alert.
3. We have used secure hashing scheme along with Salting and Hash Stretching techniques learnt in class.

4. We have added two layers of page detection technique. One with the keywords and other with general form page heuristics.

VI. Project Contribution:

Basic use case implementation -

- Use case 1 – Anandh Varadarajan
- Use case 2 – Veerendra Ramesh Kakarla
- Use case 3 – Nishant Kulkarni

Additional scenarios implemented -

- **Anandh Varadarajan**
  - Right click + Context menu click event handling of use case 3
  - Hashing + Salting + Hash Stretching for saving passwords in Use cases 1 and 2.
  - Page detection scheme using general form page heuristics.
  - Relative link path handing

- **Veerendra Ramesh Kakarla**
  - Right click + Context menu click event handling of use case 3
  - Relative link path handling
  - Page detection scheme using general form page heuristics.

- **Nishant Kulkarni**
  - Control + Left mouse click event handling of use case 3
  - Page detection scheme using general form page heuristics.

VII. Enhanced Security

For enhanced security, we need to use JS obfuscators to obfuscate the content and JavaScript files to avoid exposing any code.

VIII. Testing (Flow)

1. We first create an account in Quora.
2. Then try to sign up in Facebook by reusing password entered in Quora.
3. Then we try to again sign up in Facebook with a unique password.
4. Then go to Facebook login form and try to use password of Quora.
5. Then we give password of Facebook.
6. Go to some random stack overflow site / securitee.org site and click on a link.

IX. References

1. https://developer.chrome.com/extensions
2. https://code.google.com/archive/p/crypto-js/
3. https://www.papaparse.com/
4. https://sweetalert.js.org/guides/
5. https://www.w3schools.com/tags/ref_eventattributes.asp