UniqPass

A Security Enhancing Browser Extension

Every day, millions of users and websites are joining the internet. Users avail common services like online shopping, social connectivity, banking transactions, booking cabs, flights, hotels, appointments, etc on these websites. Often, users actions on these websites involve establishing an online identity, expressing their personal opinions, sharing sensitive documents and transferring money. This flow of sensitive information and money on the web is what makes security so important for the web. Modern browsers employ a wide range of security mechanisms to protect users information against malicious websites. Unfortunately, these protections are not sufficient and therefore users are often exposed to malicious and unwanted content. To help users secure their information on the internet, we present UniqPass -
a security enhancing browser extension.

UniqPass provides the following functionalities:

1. **Detect password reuse**

Users, unfortunately, tend to reuse passwords across websites. Whenever one of these websites is compromised, attackers can take advantage of these passwords and use them against different services. The ideal solution to this problem is to stop users from reusing passwords in the first place. UniqPass detects when a user is creating a new account on a website and compare the password that the user has selected against all other previously stored passwords. If the password is the same, the extension warns the user and encourage him to choose a different password.

2. **Detect the entering of passwords on the wrong website**

UniqPass detects whether the user is trying to login to a website with the password of a different website. This should allow us to protect users from falling victim to phishing attacks (e.g. detect that the user is entering her paypal.com password to the attacker.com website).

3. **Modify link-clicking behavior**

Some security researchers have argued that most users would be protected if the software would stop them from visiting unpopular websites. Your extension should inspect all links in all web pages visited and for those links that are leading the user outside of the Alexa top 10K websites, the user should be warned if they click on that link. The user should have the option to dismiss the block once (i.e. be allowed to visit that website) or forever (i.e. ask the extension not to bother her next time she visits that particular website).

In the rest of the document, we explain the different design decisions made during the implementation of these functionalities. We explain who did what and provide a list of references that we used for implementing the project.

Design Overview

Design Component	UniqPass Decision
Browser Support	Chrome
Javascript Specification	ECMAScript 2017
Database	Chrome Storage
Password Digest API	window.crypto.subtle
Password Digest Algorithm	SHA-256
Chrome API's	chrome.storage, chrome.runtime, chrome.webNavigation
Alert UI	sweetalert.js

Overall Algorithm For Detecting Password Reuse on Sign Up and Log In

```
1. Add an 'input' event listener on the DOM
2. If 'input' event target is a 'password' field
      a. Add an 'oninput' event listener on password field
          i.   Get form type - LOGIN/SIGNUP
         ii.   Compare current password and previously stored
               VERIFIED passwords
        iii.   Show appropriate warnings
      b. Add an 'onclick' event listener on submit button field
          i.   Get hash of the password
         ii.   Store the url, password, timestamp and status in
               chrome storage with status = UNVERIFIED
      c. Add an 'onmouseover' event listener on submit button field
          i.   If already used password, show appropriate warning
         ii.   clear the password
3. Observe page load using chrome.webNavigation.onCommitted listener
      a. If page loads due to 'form_submit' action and if page url
         has an entry in DB with timestamp in last 10 seconds
          i.   Update latest password status = VERIFIED
         ii.   Delete all UNVERIFIED passwords for that URL
```

Detect Password Reuse

Decision 1 - When to add listener to a password field?

Approach 1 - On Page Load

Initially, we tried to add 'input' listeners to password fields on page load. We tried both the 'load' event listener and jquery's $(document).read() function. We realized that there were many websites where the SIGNUP/LOGIN form opens quite some time after the load.
For example, the sign up form on instagram.

In general, whenever forms load as a modal or dialog box, we were unable to attach 'input' listeners to password field on page 'load' event.

Approach 2 - Using MutationObserver API

We used javascript's 'new MutationObserver(callback)' API to observe the DOM elements. This callback gets triggered as soon as a new html element is added to the DOM. In the callback, we added a condition to look for an HTML element with input[type='password'] and then add a click handler to such event. Even if a input element was added to DOM long time after page was loaded (for example, in modal or dialog) , still the observer was able to catch this. Thus the approach worked for us. However, observing addition and removal of every DOM element is time taking and unnecessary. It made the page a bit slow.

Approach 3 - On Any Input Event On DOM

We listen to all 'input' event on the DOM and if the event target is a 'password' type element, we attach an oninput event to the password element. Thus the event is added only when the user starts changing the text in the password field.

Advantages

1. This approach assures that the form has been loaded and our logic runs only when the user is able to see and make changes to the form.
2. We observed that many websites have hidden password fields in the DOM. While using approach 1 or approach 2, we had to exclusively handle that testcase. Approach 3 handles that case by design.
3. Even if a page has multiple forms, this approach lets us know which form is being edited currently. Using our form detection algorithm, we can then easily differentiate between the type of the form.

Password Matching

Once the event listener is added to the input field with type password, we are comparing the password entered with all the stored passwords. If the user entered already saved password

we are alerting the user depending on if he is trying to use the password in login form or signup form. In our logic, we are also ensuring that the user doesn't subsume the already stored passwords.

Decision 2 - How do we detect Password and Submit field?

```
let passwordElem = event.target; //event is the input event on dom
let closestFormElem = passwordElem.closest('form');
let buttonSelector = 'input[type=\'submit\'], button[type=\'submit\'],
input[type=\'button\']';
let buttonElem = closestFormElem.querySelector(buttonSelector);
```

Looking for the closest form element makes it sure that we are getting the correct form,out of multiple forms which might be present in a single page.

Decision 3 - Are passwords stored in plaintext?

No.

We hash the password using SHA-256 algorithm before storing it in chrome.storage.local. While comparing the passwords, we compare the hash of current password with the hash of previously stored passwords. This protects our extension from an offline attack if some bug is found in chrome.storage.local api. We use browsers window.crypto.subtle.digest() api to calculate the password digest.

Decision 4 -Where to store the user passwords?

We explored both the HTML local storage API's and Chrome Storage APIs in detail.

Approach 1 - window.localStorage

1. Local Storage is accessible to every extension.
   It can be read, modified and cleared by anyone, including the user.

   Threat Model - If another malicious extension tries to read UniqPass data, they can easily do so using the local storage API.

2. Local Storage is not available in background scripts. We need to rely on message passing to pass data between content script and background script.
3. Local Storage is synchronous and blocking.

Approach 2 - chrome.storage.local

We prefer Chrome Storage API because of the following reasons

1. Using the chrome storage, each extension gets its own storage. Thus, an extension cannot modify other extension's data.
2. chrome.storage.local is available at both the content script and background script easily.
4. Chrome Storage is asynchronous and thus faster than Local Storage if done properly.
5. Chrome Storage also has an API chrome.storage.sync which can sync extension data across multiple devices.
6. Chrome Storage automagically serializes JSON-compatible data, can store non-strings with no additional boilerplate.
7. It also does not have the 5MB limit which local storage has.

Decision 5 - Detect the form type - Log In vs Sign Up?

When a user enters her password on a form, UniqPass detects the form type. Depending on the form type, it displays appropriate messages to the user.

Possible Form Types - LOGIN, SIGNUP, UNKNOWN

Form Type Detection Algorithm

```
1. Get 'form' element closest to the input[type='password'] element
2. Apply different heuristics in order of priority
3. Heuristic 1 - Detect form type using submit button text
      a. Get the submit button element on the form
      b. if the submit button text is ['login', 'sign in'], return
         LOGIN
      c. if the submit button text is ['sign up', 'create'], return
         SIGNUP
4. Heuristic 2 - Detect form type using a number of password fields
      a. Get the number of password fields in the form
      b. If the number of password fields > 1, return SIGNUP
5. Heuristic 3 - Detect form type using a number of input text/email
   fields
      a. Get the number of text/email input fields in the form
      b. If number of text/email input fields > 1, return SIGNUP
6. Heuristic 4 - If the submit button was not part of the form:
      a. Search for the span in the document with text as ['sign up',
         'create', 'next'], return SIGNUP
      b. This was helpful in Gmail signup and twitter signup forms.
7. return UNKNOWN
```

Algorithm Analysis

- The above algorithm can easily incorporate any new heuristics to differentiate between Log In and Sign Up form
- We don't check for the 2nd heuristic if the 1st heuristic is sufficient. We believe that checking for button text is a much stronger heuristic than checking number of password fields
- Heuristic 1 and Heuristic 4 can also be extended to more number of websites by adding more keywords in the log in or sign up list

Decision 6 - When to store the user passwords? - On Click vs URL Redirect vs Status 200

Approach 1 - On Status 200

The best time to store password is when we are sure that the password being stored is correct. Thus our first approach to saving passwords was to look for a 200 status code of the submit network call. We tried multiple approaches:

1. Using chrome.webRequest.onRequestCompleted

chrome.webRequest API can be used to observe and analyze traffic and to intercept, block, or modify requests in-flight. We intercepted all the sub-requests which were made by the page. We came up with a regular expression which could identify the submit request related to the submit call. For the regular expression, we used heuristics like

- Request URL containing 'login' keyword
- The request should be a POST request
- The request should contain 'password'/'pass' etc as an attribute in POST form

This approach worked for a few websites. However, our regular expression did not generalize for multiple websites.

2. Using chrome.webRequest.onBeforeRequest

We were analyzing all the calls before they were sent to the server side. The reason behind this approach was inspired by Google password manager, where we were trying to save the passwords no matter if it was a successful login or not. The main motivation for this approach was to ensure that we were not bypassing the form validations. To do this, we wrote some rules to match the request of the form submit. The main heuristic was to catch the POST calls, but after analysing these calls for different websites and different types of form submit, we got to know that no set of fixed rules could cover all the possibilities. Also,

an action on a website triggered many other actions which were also of type POST. This introduced fuzziness in the logic of saving the password.

Approach 2 - On Click Of The Submit Button

Next, we saved the user passwords on the click of submit button. The obvious caveat with this approach is that we will be storing wrong passwords in our database. For example, if the frontend form validation failed on click of the submit button, we still would save the password. After trying this approach on multiple websites, we realized that our extension is storing a large number of wrong passwords. It also blocked users from entering passwords which is not an already existing passwords (as we stored wrong passwords).

An advantage with this approach is that we would never miss a correct password. However, for failed login/signup attempts on a website, this approach would store wrong passwords.

Approach 3 - On URL redirect after form submit

Approach 1 was the best but we couldn't generalize our approach on multiple websites. Approach 2 generalized well but it stored wrong passwords. Thus, we tried to come with a middle ground approach.

We did an extensive study of existing Chrome Password Manager. We realized that the official chrome password manager also stores wrong passwords. It detects a form submit and gives an option to users to store passwords only when the url changes after a form submit action.

We played with chrome.webNavigation API to detect a new page load on form submit action. Our final algorithm looked like this:

```
1. On each submit click, store the following json in chrome storage
   {
      url: 'facebook.com',
      password: 'cse509',
      timestamp: Date.now(),
      status: 'UNVERIFIED'
   }
2. In background script, keep observing the page load event using
   chrome.webNavigation.onCommitted.addListener. On each page load,
   if page load was due to 'form_submit' transition type
      a. Get the entries of this url from the storage
      b. If there is an entry with timestamp within last 5 seconds
          i.   Update the entry status as VERIFIED
         ii.   Delete all the UNVERIFIED entries
```

The approach, thus, marks the password as VERIFIED only after a successful URL redirect.
Note:

1. While comparing passwords, we only compare VERIFIED passwords with the user input passwords.
2. UniqPass stores passwords on both log in and sign up forms. Before installing our extension, user would have already signed up on different websites. Once the user installs our extension and tries to login, UniqPass catches user passwords for those websites if a URL redirect occurred. Thus, UniqPass also secures user on websites where user had created accounts even before installing the extension.

Limitations of this approach
On a few websites, URL redirects occur even when there is no successful login. In such cases, we will wrongly mark the password as VERIFIED. Example of such cases are login on facebook.com.
However such cases are very less compared to the save of wrong passwords on click.

Decision 7 - Enforce Unique passwords

Approach 1 - On click of submit

As soon as user clicked on submit, we are doing a password matching again to see the user changed the password after the warning. If he didn't we are alerting the user again that he is still using the password which has been used before and intercepting the submit action. The interception was tried by these two ways:

a. **Monitor the WebRequest:** In this approach, we tried intercepting all the calls. Using the chrome.webrequest.onBeforeRequest we wrote a couple of rules to match the submit call. The rules couldn't be generic enough to intercept all the submit calls for different websites and thus we didn't proceed with this approach.
b. **Prevent default:** On click of submit, if the user was reusing the existing passwords we intercepted the action by using preventdefault method of an event and making onsubmit false. This worked for few sites but this was breaking the default behavior of the site by loading the home page.

Approach 2 - On mouseover of submit

If the user ignored the warning of password reuse and is still trying to submit, as soon as he put the mouse over the submit button, we are validating if there is password reuse. We are alerting the user of the same and clearing the password. This ensures that the default behavior of the web app is not affected.

For enforcing the user for unique passwords we had to make some of our calls

Decision 8 - Alert Box Designing

Approach 1 - Bootstrap

In our efforts to show user friendly messages, we styled the alert box. Our obvious choice was to use Bootstrap modal as Bootstrap is widely popular and quite stable.

To our surprise, we soon hit a blocker as any css/js file used by extension in contentscript.js interferes with the CSS of the website. For example, as soon as we added the bootstrap css to facebook.com, the layout of facebook.com changed with div's misaligned. In order to keep the website as it is, we need to include the same bootstrap.css version which facebook.com is using currently. As our extension will be used on multiple websites, it's not possible to choose one specific version of bootstrap which satisfies all the websites.

A common solution to these problems is to load our custom css file in an iframe. We did try that. However, we started facing issues in positioning the iframe at the center of each page. Thus we decided to go with creating our own css.

Approach 2 - Own Implemented Alert Box

In this approach, we tried implementing our own custom css for the alert box.
We used the modal css from https://www.w3schools.com/howto/howto_css_modals.asp.
Though we could get the modal working, the modal alert box did not wait for the user input. The modal showed and vanished without user action. In order to wait for user input (Ok, Cancel), we would have to use promises on our custom modal. We also had to position this modal at the page center. This felt like reinventing the wheel as there are numerous already implemented modals.

Approach 3 - Sweet Alert

We explored a common alert library - sweetalert.js. It provides different UI designs along with full functionality support for an alert box. Thus, we used it for all our alert messages. We use promises to wait for the user input.

Detect the entering of passwords on the wrong website

Threat Model: Sometimes the users are tricked into entering credentials on the masquerading sites which aim to steal user information, especially the user credentials that provide an attacker the access to genuine sites. In these cases, our extension will alert the user the current URL on which the user is entering the credentials and the URL for which he actually used the password. Our extension acts as credential phishing prevention.

The task can be defined as detecting the password reuse and detecting for which site was that password reused.

The first subtask of this task is the same as mentioned above which includes detecting password reuse by various heuristics. Once the password reuse is detected, we fetch the URL saved against that password in the chrome storage.

The user is then alerted that he has used the password of the fetched URL on the current URL. If the user was not aware of the same, this saves the user from phishing attacks. On the other hand, if the user was trying to reuse the password, the user is then enforced to input a unique password. This enforcement is fail-proof enough that the user NEVER enters the same password on different sites.

Modify link-clicking behavior

Decision 9 - How to fetch and store top 10k websites?

Approach 1 - make a request to Alexa Api dynamically

The Alexa Top Sites is a web service that provides lists of web sites, ordered by Alexa Traffic Rank. Using this web service, developers can page through lists of top sites and incorporate traffic data into their applications. There is a documentation describing how to use Alexa Top Sites.

A site's ranking is based on a combined measure of reach and page views computed over a trailing 3 month period. Reach is determined by the number of unique Alexa users who visit a site on a given day. The site with the highest combination of users and page views is ranked #1. The list is updated everyday. However the api is "*private"* and an AWS user is charged $0.0025 per URL returned (e.g. $.25 for 100 URLs). But there is a way around this as the api to access the top 1 million is available and public.

Another problem with above approach is that if for some reason the alexa api is not up then the extension will fail to classify websites altogether.

Approach 2 - save the top 10k websites locally and parse the json each time.

As we have to access to top 1 million websites we can parse the csv file and create a json of top 10k websites. The generated json can then be added as a resource to extension itself in the manifest file. Converting to json has added benefit of file being easily readable by the javascript. Now the only problem is that for each time the extension is used the json has to be loaded which introduces few vulnerabilities to the extension. As other extensions could edit the json file once they know the location of the file.

Approach 3 - load top 10k json only on install to save to chrome.storage.local

As mentioned earlier using chrome local storage provides the best approach to saving the extension content in a persistent storage. In this approach we have produced the top 10k websites json and added it to our extension. On first install of the extension the json is loaded in background and stored in chrome local storage. After that chrome local storage is used to access the list of websites.

Threat Model: If somehow the attacker could add a malicious website to this json then the website could be whitelisted. This could happen at various level.

1. If the system the extension is running on is compromised then the attacker could easily add any website to the json and surpass the website checker. But in approach 3 makes sure that json is read only on install so even if the attacker changes the json it won't be of much use.

Another way for an attacker could be to compromise chrome local storage.Although once the system is compromised this would be last priority for an attacker given he/she has full access to the system

2. Other extensions or even local programs could try and add their websites to the json. The Web browser(Chrome in our case) provide protection against it by creating a *.crx* file which is protected access for only given extension. Still other programs running in the system can change the file as .crx file can be edited locally. Again in approach is secures against the attack by loading json on load.

Decision 10 - Handling user clicks on each website

Whenever each page loads in the browser, all the links in the webpage are parsed using anchor tag and for each link a new mouse click event listener is created. A whitelist is created by loading the domains from local storage. The whitelist is saved as a shared variable which is maintained for throughout the lifecycle of a webpage.

When user clicks on a hyperlink or button or any other anchor tag and click event is intercepted by the custom listener event. Then *"prevent default*" is called which essentially stops the event. The extension matches the link from the click event with the list of saved websites.

Matching hostnames with domains
The hostnames and domain names naming convention follow [RFC 1178] and [RFC 819] . The list of hostnames provided by Alexa top 10k websites can be contained in 2 different websites. The domain needs to searched instead of the hostname consider a case of *'google.com'* and *'mail.google.com'.* Another interesting example is, *Texas instrument* website 'www.ti.com/' and Professor Amir Rehmati's personal website 'https://amir.rahmati.com/' and there is another case. Thus it makes sense to just compare the domain name of the 2 urls

After the domain name is matched. If it is matches anyone domain name from local storage then the click behavior is resumed and no prompt is shown to the user. But if there is no match then user is shown a prompt to choose whether to cancel the click event, continue to website once or to mark the website as safe permanently. If user asks to mark website safe an asynchronous call is made to save the new website in the local storage.

Hyperlink Cases:

1. Hyperlinks with page reload:
   This is the default case with most of the websites where a click event to the hyperlink triggers a call to the website on results in a page reload. In this case if the website is once whitelisted then it won't result in a prompt again.
2. Hyperlinks without page reload:
   There are growing number of websites that use hyperlinks to ease page navigation. These events doesn't necessarily result in page reload. Thus if the website is saved once after first click subsequent clicks shouldn't result in a prompt.

Best Practices

1. We have used many ECMAScript 2017 functions like let, arrow operator and async/await
2. We have tried to keep the code modularized by creating multiple functions and abstracted different functionalities in different files
3. We have used ENUMS to make the code more readable
4. We have tried to avoid multiple calls for the same thing. For example, we just load the Alexa-10K websites just once on install.
5. We have removed all the console errors and all the errors on the extension page by adding error checking wherever possible
6. Before starting the development, we created basic chrome extension examples (like Color Change Extension) to explore different API's given by chrome.

How to set up UniqPass?

1. Download the master-dev branch code from our repo - https://github.com/Vishakha93/PasswordSecurityExtension

   In case you are using git, use the following command -
   ```
   git clone --single-branch --branch master-dev
   git@github.com:Vishakha93/PasswordSecurityExtension.git
   ```

2. To view the code,
   cd PasswordSecurityExtension/Security-Enhancing-Browser-Extension/

3. Go to chrome, switch the developer mode on at chrome://extensions and load the code folder

4. Once the extension is loaded, you will see a lock icon and can start testing different websites

How to test UniqPass?

What flows to try?

1. Sign Up on a website followed by Sign Up using the same password on a different website
2. Sign Up on a website followed by login using the same password on a different website

3. Log In on a website followed by Sign Up using the same password on a different website

What to test in each flow?

1. UNVERIFIED password save on click
1. VERIFIED password save on URL refresh in 60 sec
2. Appropriate message on repeating password on sign up
3. Appropriate message on repeating password on login
4. Detect forms correctly when there are multiple forms in a page
5. Password clear on mouseover submit
6. Non-Alexa Click - Warning
    a. Pass this once - Go to website & warning shown next time
    b. Whitelist website - No warning shown next time -
    c. Cancel - stop click event & warning show next time
7. Mail.google.com should be allowed if .google.com
8. Same page links - 7 should work

Some example websites

We tried the above flows on different websites. We present our findings in the table below.

Status represents if UniqPass successfully works on that website for that particular action.

Website	Action	Status	Reason
https://www.instagram.com/	Sign Up	Yes	
https://www.instagram.com/	Log In	Yes	
https://auth.geeksforgeeks.org/	Sign Up	No	No form submit or link
https://auth.geeksforgeeks.org/	Log In	Yes	Error is shown Password updated on the login
https://leetcode.com/accounts/login/	Sign Up	No	Button outside form
https://leetcode.com/accounts/login/	Log In	No	Button outside form
https://piazza.com/stonybrook	Log In	No	Log In
https://twitter.com/login	Sign Up	No	No form
https://twitter.com/login	Log In	Yes	Use this specific link
https://blackboard.stonybrook.edu/webapps/login/	Log In	Yes	
https://psns.cc.stonybrook.edu/psp/csprods/EMPLOYEE/CAMP/?cmd=login	Log In	Yes	

https://www.facebook.com/	Log In	Yes	
https://www.facebook.com/	Sign Up	Yes	
Gmail	Sign Up	Partially	Shows error message if you use the password of a different website. Password is cleared. Submit button not inside a form so we are not able to save password.
Gmail	Log In	Partially	Same as above
https://www.securitee.org/	Mouse click	Yes	
https://profile.oracle.com/myprofile/account/create-account.jspx	Sign Up	No	It does not have a form
https://sweetalert.js.org/guides/	Mouse click	Yes	
https://amir.rahmati.com/	Mouse click	Yes	
https://climate.com/	Mouse click	No	Sweetalert uses css modules which interferes with native js modules of webpage.

UniqPass Limitations

1. UniqPass works properly only when the password and submit button field is wrapped in a form element
2. It becomes difficult to identify the submit button if it's not a button or input type. Many a time, submit buttons are span/div based and have text as "Next/Continue". This makes it difficult to identify both the submit button and the form type.
3. UniqPass sometimes stores wrong passwords if there is a URL redirect even on unsuccessful login.
4. As heuristic 2, 3 and 4 of the form detection algorithm are on Sign Up forms, the algorithm detects sign up with fairly good accuracy. However, the Form Detection Algorithm may fail on websites which have log in forms with button text different than 'log in' and 'sign in'.
5. UniqPass works only for a single user. We do not store usernames in our database as we are just supporting login and sign up of a single user.
6. The clicks to new tab and right click are not covered in our test cases. As this creates a new page and the context to previous call is lost.

Who did what?

All decision decisions were jointly taken by the team. The team met almost every day to discuss the next steps. As functionality 1 and functionality 2 had a lot of overlapping content, Mitesh and Vishakha worked jointly on many of the common tasks like when to add the listener to password diels, when to store passwords, etc. Functionality 3 was solely owned by Divyam. We jointly tested the whole extension end to end and thought about various threat models.

Team Member	Contribution
Mitesh	• Mostly worked on 'detecting password reuse', • Figured out chrome.storage, chrome.webNavigation and window.crypto.subtle APIs • Improvised adding event listener on user input instead of page load • Came up with the form detection algorithm • Wrote the logic to store passwords on URL redirect • Tried bootstrap modal and custom implementation of alert box
Vishakha	• Project setup • Explored basic chrome extension development • Mostly worked on 'detecting entering of password on the wrong website', • Wrote the logic for enforcing unique passwords • Explored edge cases for detecting submit button on span text • Explored custom popup for the extension • Wrote password matching algorithm. • Handled different asynchronous events using async and await • Handled intercepting submit event using chrome.webRequest
Divyam	• Mostly worked on 'modify link click behavior' • Came up with algorithm to manage whitelist across lifecycle of webpage and domain matching algorithm. • Inspected security aspect of the extension and various threat models. • Sweetalert framework to show custom dialogs. • Handled api request, Ajax and handling csv. • Asynchronous calls to local storage to get and put urls. • Explored edge cases for detecting clicks to different websites and different types of click.

References

1. https://developer.chrome.com/extensions/getstarted
2. https://developers.chrome.com/extensions/api_index
3. https://www.w3schools.com/howto/howto_css_modals.asp
4. https://developers.chrome.com/extensions/storage
5. https://sweetalert.js.org/guides/
6. https://getbootstrap.com/docs/4.0/components/modal/
7. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
8. https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/digest
9. https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver
10. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip
11. https://stackoverflow.com/questions/53939205/how-to-avoid-extension-context-invalidated-errors-when-messaging-after-an-exte
12. https://stackoverflow.com/questions/8837454/sort-array-of-objects-by-single-key-with-date-value
13. https://www.sohamkamani.com/blog/2017/08/21/enums-in-javascript/
14. https://chromium.googlesource.com/chromium/src/+/master/chrome/common/extensions/docs/examples/api
15. https://developer.chrome.com/extensions/webRequest#implementation
16. https://stackoverflow.com/questions/8498592/extract-hostname-name-from-string/16934798
17. https://dev.to/aussieguy/reading-files-in-a-chrome-extension--2c03
18. https://gomakethings.com/why-event-delegation-is-a-better-way-to-listen-for-events-in-vanilla-js/