Tabnabbing Detection With Chrome Extension

Team CacheMoney

Team Information:

Team Name: CacheMoney

Name	**SBU-ID**	**Github**	**Email**
Tao Lin	talin	TaoLinShowdown	tao.lin@stonybrook.edu
Travis Li	trli	Travis-Li	travis.li@stonybrook.edu
Bryan Lai	brlai	gblaih	bryan.lai@stonybrook.edu
Alexander Chung	alchung	alexander-chung	alexander.chung@stonybrook.edu

Setup and Testing:

Clone/Download our project from https://github.com/TaoLinShowdown/CacheMoney. To use our extension, navigate to *chrome://extensions/* in a Google Chrome browser. In the top right, enable developer mode. Then on the top left, click load unpacked. Navigate to and select the extension folder. The extension is now ready to use.

To test the extension, visit a website that has moving parts. A youtube video works well:

1. Sit on the youtube video as it is playing for a second or so

2. Then move to a different tab

3. After a few seconds return to the youtube video

4. A dark overlay covering the whole screen will appear with a dialog on the bottom right Scrolling is now disabled, so the user is locked to the position they last left the tab at

5. The red highlights will not be on screen yet, but give it a few seconds (maybe a minute) and it should show up

6. On the bottom right of the screen you are given the option to close the overlay and dialog and ignore the warning or to close the overlay and dialog but also report the url

7. Now if you reload the youtube video, an alert will pop up and warning users that the current url was previously reported as malicious

Some notes:

- The red highlighting can take some time if the window is big. To significantly reduce lag time, make the chrome window smaller while testing.
- If you refocus a tab but no changes were detected, nothing will happen

Description of problem:

From: https://www.securitee.org/teaching/cse331/projects/project2.html

Tabnabbing is a variant of phishing that primarily works off the betrayal of users' trust of browser tabs that they already know exist in the browser session. As the user continues to browse the web and collect more tabs, it is easy to get lost in the "sea of tabs". Malicious websites can thus exploit the loss of focus in a tab (when the user clicks off the current tab) and change the contents of the page to a seemingly normal login screen; say, gmail.com or facebook.com. When the user clicks back to that tab, he and she will see that they have to log in again without carefully checking that the url is not that of the login website's. The natural assumption would be that they have been signed out by Google or Facebook and proceed to sign in again on the fake page, thus leaking their credentials to the attackers.

High level details of extension:

1. While a user is browsing a webpage, screenshots are taken on regular intervals (2000ms) using chrome's api *captureVisibleTab* and the setInterval command.

    - Ideally, we would take screenshots only when we detect loss of focus, but chrome does not provide functionality for that.

    - We used setInterval instead of chrome's built in chrome.alarms api, because chrome.alarms can only be set to run code in intervals of minutes, but to keep snapshots relevant and fresh we wanted to take a snapshot 2 seconds.

    Chrome's *captureVisibleTab* returns a dataURL of the image as a result. We store the dataURL and the id of the tab in the form {*tab_id: dataURL}* in a variable *tabDataStore* in snapshotter.js.

2. When a user changes focus to a tab, we check if we already have a screenshot taken of that tab. If we do, we take a fresh screenshot using *captureVisibleTab*. We compare the new dataURL

with the dataURL of the tab stored in tabDataStore using *!==* (dataURLs are strings, so we just used string comparison).

If the dataURLs are not the same, we send a message to *screenshot.js*, the content script with access to the webpage's DOM with both dataURLs as the payload.

3. On receiving a message, *screenshot.js* disables scrolling and adds the overlay and dialog to the DOM. The overlay does not have any red highlights at this point. The overlay is a div that contains many more divs inside it, each 10px by 10px to cover the whole viewport.

   The content script then compares the two images using **resemblejs** and generates a new image that highlights where the two images differ in magenta.

   The resulting difference image is then split into 10 by 10 pixel canvas objects which we call tiles. As we create each tile, we obtain its image data to determine if magenta is present. If magenta is present, there is a mismatch and so we take the coordinates of the tile and set the background of the respective div at that coordinate to a transparent red.

   - We initially planned for the background script, *snapshotter.js*, to run resemblejs on the two dataURLs, then produce an array of coordinates to send to *screenshot.js* to color in the overlay with. However, because resemble took more time than we expected, we decided that it would be best to not stop all background script functions just to wait for resemblejs to finish, and instead offload the work to each tab's content script.

   - As a result, there can be a lot of lag time when refocusing into a tab and having to wait for the red highlighting to show up. But, the background script continues to run and users can switch to a different tab and continue browsing and snapshots will still be taken at a regular interval.

   - We decided to disable scrolling because the overlay had style *display: fixed*. So if the user scrolled and moved away from the position the highlights would be in the wrong areas. We used *display: fixed* for the overlay because then we wouldn't have to find the exact coordinates of the user's viewport and position the overlay around that. We felt that this would keep the overlay simple to work with.

4. In addition to the overlay, we also added a dialog box to the bottom right corner of the page. Users can choose to either report or ignore the changes detected by the extension. With either option, the overlay and the dialog will be removed and scrolling will be reenabled.

When reporting the url, *screenshot.js* will send a message to the background script *snapshotter.js* with the url as the payload. *snapshotter.js* will make an XMLHttpRequest object and make a POST request with the url of the current page as the payload to our server hosted in an Amazon Web Services' Linux VM. The server then takes the url and adds it as a document to our **MongoDB Atlas** database.

- We sent a message to the background script so that it could make the HTTP request, because the content script could not make HTTP requests from an HTTPS environment.

Along with updating the database, the server also sends back an updated list of reported urls, that we use to update our chrome.storage.local with.

5. Whenever a tab is updated, as an additional security measure, the current url is taken and compared with the MongoDB Atlas collection of malicious sites. If the current url has been found on the blacklisted sites collection, the user is alerted that the url they are on has been previously reported by another user.

   On installation, *url.js* will make an XMLHttpRequest object and send a GET request to our server. The server then retrieves data from the URLs collection of our MongoDB Atlas database and sends it back to *url.js* to store into chrome.storage.local.

   An alarm was created to fire every 5 minutes and a listener to react to the alarm. In the listener, *url.js* will do the same thing that was done on installation (sending a GET request and putting the data in chrome.storage). We decided to store the data in chrome.storage.local instead of querying the database every time a tab was updated because it was more efficient.

   A tab listener was added to detect when the url changes. When it activates, it takes the page url, retrieves the data from chrome.storage, and checks if the url is found in the data. If it is found, then it sends an alert to the user telling that the site is malicious. If the url cannot be found, then nothing happens.

6. To facilitate a connection between our extension and Mongodb Atlas, we built a rest api server using **express** and **body-parser**. The functionality of the server is described in the previous points.

7. We hosted the server on one of **Amazon Web Services' EC2 cloud Linux servers** and used the **forever** library to keep the api running as a daemon script. Even without someone actively running "node app.js" on an ssh connection, we can make GET/POST requests to the api.

Distribution of workload:

Tao Lin:

- Composed other member's parts into a consolidated app + server project
- Implemented taking screenshots of tabs on regular interval
- Created overlay with recolored changed tiles in webpage
- Set up and deployed Mongodb Atlas cluster

Bryan Lai:

- Worked with resemblejs
- Created function that inputs two images and compares them, returning an array of coordinates of the mismatched tiles
- Resemblejs testing

Alexander Chung:

- Dialog popup box, Yes to push url, no to cancel
- POST requests of REST api
- Set up Amazon Web Service EC2 cloud linux server to host RESTful api from a remote source (foreverjs for server)

Travis Li:

- Mongodb Atlas and Expressjs RESTful api
- GET requests to REST api
- Implemented comparing current url to database collection and alert user if found

Third Party content/references:

How to disable scrolling temporarily - https://stackoverflow.com/questions/4770025/how-to-disable-scrolling-temporarily

RESTful API with Nodejs and Mongodb Atlas guide - https://www.thepolyglotdeveloper.com/2018/09/developing-restful-api-nodejs-mongodb-atlas/

resemblejs - https://www.npmjs.com/package/resemblejs

forever - https://www.npmjs.com/package/forever

express - https://www.npmjs.com/package/express

body-parser - https://www.npmjs.com/package/body-parser

AWS EC2 Linux VM - https://aws.amazon.com/mp/linux/