CSE361 Project #2 - TabNabber Attack Detector
Team: 1337 Coders

By: Tejas Prasad & Tyler Maciaszek

1. Intro-

Tabnabbing is a phishing attack where a seemingly normal website changes it look and feel to imitate the login page of a more secure service such as Gmail, Facebook, or your bank. Users usually have many tabs open and switch between them sometimes forgetting about previously open tabs. If a user forgets about a tab for a while and finally clicks back to it, and it looks like their bank's login page sometimes they won't think twice and just immediately login. While the URL would give it away many users aren't trained to constantly look at the URL bar and just judge a page based on its looks. Many tabnabbing pages look like any other blog, forum, or news page initially but with some simple javascript can completely change their look which makes this attack possible.

2. Implementation-

At the core of our extension, we utilize the ability to have content scripts, which are injected into each webpage, and background pages or scripts which run while a browser is open and for the life of the extension. This allows us to simulate a client and server with the content script representing the client and the background script representing the server. The content script also allows us to manipulate the Document Object Model (DOM) and add or remove elements as needed.

Background Page/Script:

The background script is essentially just a javascript file that runs in the background when a browser is open and exists for the life of the extension. We specifically used this file as a way to use many of the Chrome API functions that allowed us to create our extension. The background script handles taking a screenshot of the tab every five seconds through the use of the Google Chrome API which has a function to take a screenshot of the current visible tab. This function returns a data URI of the screenshot which is then stored in a key-value data structure, dictionary, where the URL is the key and the data URI is the value. The background script is also constantly listening for messages that may be sent from the content script so that it can send information to the content script such as the aforementioned dictionary. The reason this needs to be done is that the content script is limited in what it can do in regards to chrome APIs. The background script will also be in charge of setting the color and text of

the extensions icon in the browser's toolbar which is another way we alert users to the attack and how severe, in terms of changed percentage, the attack was.

Content Script:

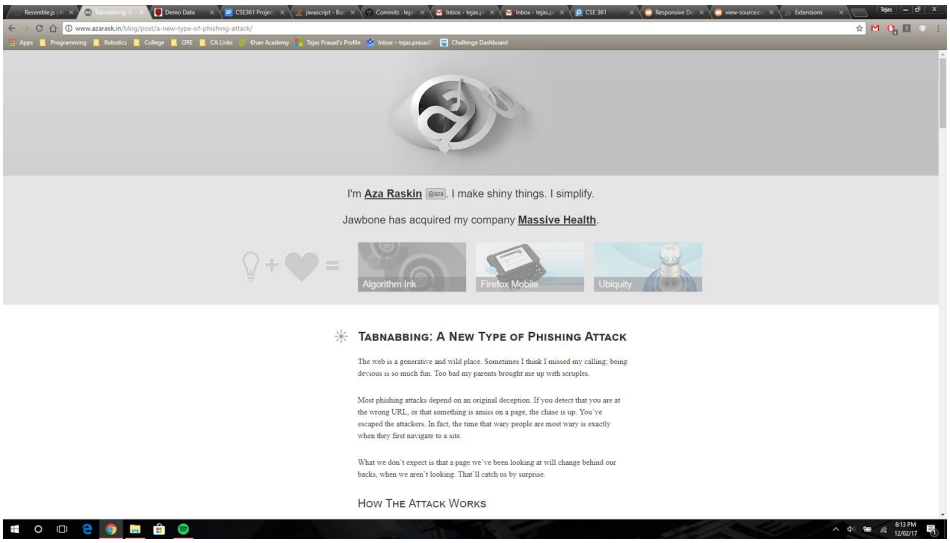
The content script is the javascript file that runs in the context of the webpage and is more limited in functionality so it interacts with the background script for information. The content script handles the actual comparison of the images and displaying the result on the webpage. This is done through a library called Resemble.js. When a tab regains focus it sends a message to the background script for three things. It takes a new fresh screenshot for the comparison, the dictionary of images, and the current URL. After it has those three things it checks to see if there's an image in the dictionary to compare to then calls a Resemble function to do the comparison. When the comparison is done a new canvas element is created and giving properties so that it will overlay the current page. The image returned from resemble is then drawn onto the canvas so that is displayed to the user.

Manifest:

This file is a JSON formatted file that provides important information to the Chrome browser when loading the extension. It provides the browser with key details such as the extensions name and version so that information can be displayed to the user. This file also requests permissions that the extension needs in our case was activeTab, tabs, and <all_urls>. The activeTab allowed us to get the current URL, tabs allowed us to take screenshots, and <all_urls> allowed us to use our extension on every page. We also let the extension know which javascript files we were including in our extension and any libraries we wanted to interact with.

Sequence of Events:

When a user first opens a new tab, the content script sends a message to the background script to take a screenshot so it is guaranteed there is a screenshot saved of the tab in the dictionary data structure. The background script also starts a timer to call the function to take a screenshot of the tab every 5 seconds. This continues while the user is focused on the tab.

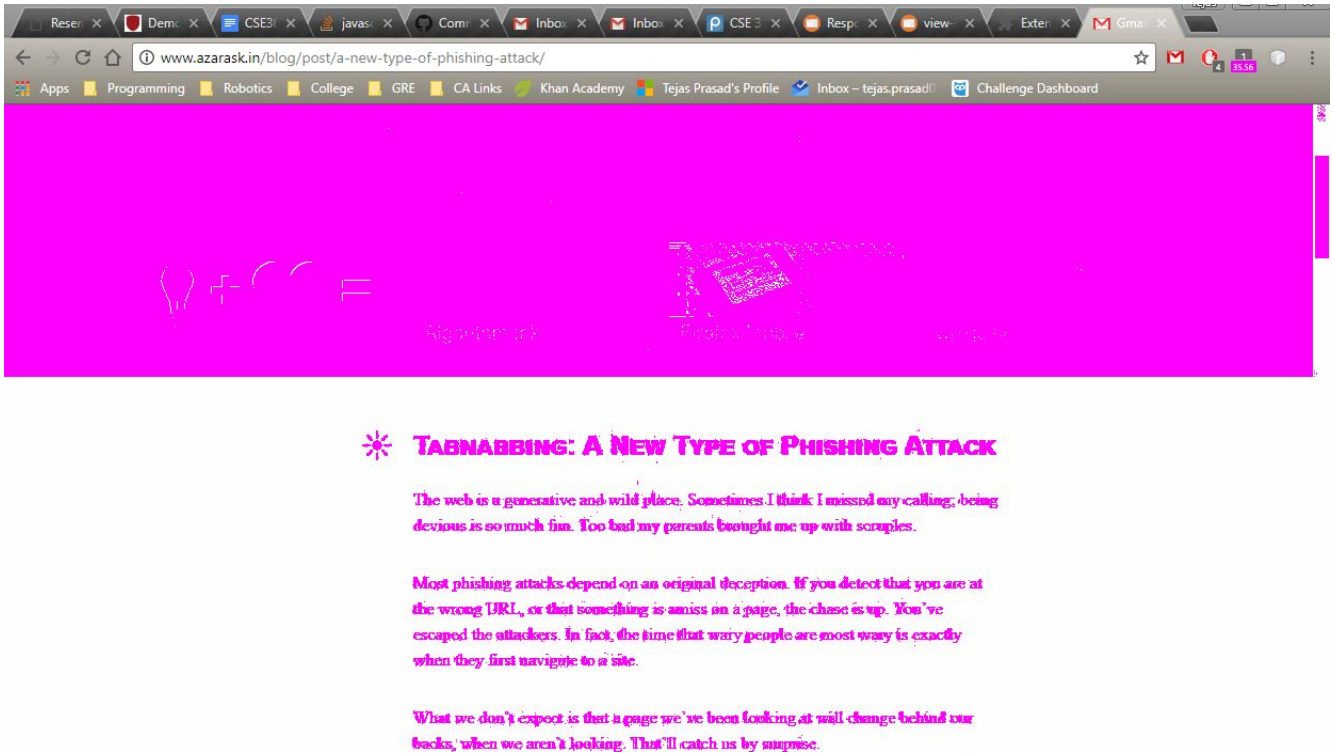


Sample TabNabber Attack Site: Before the change

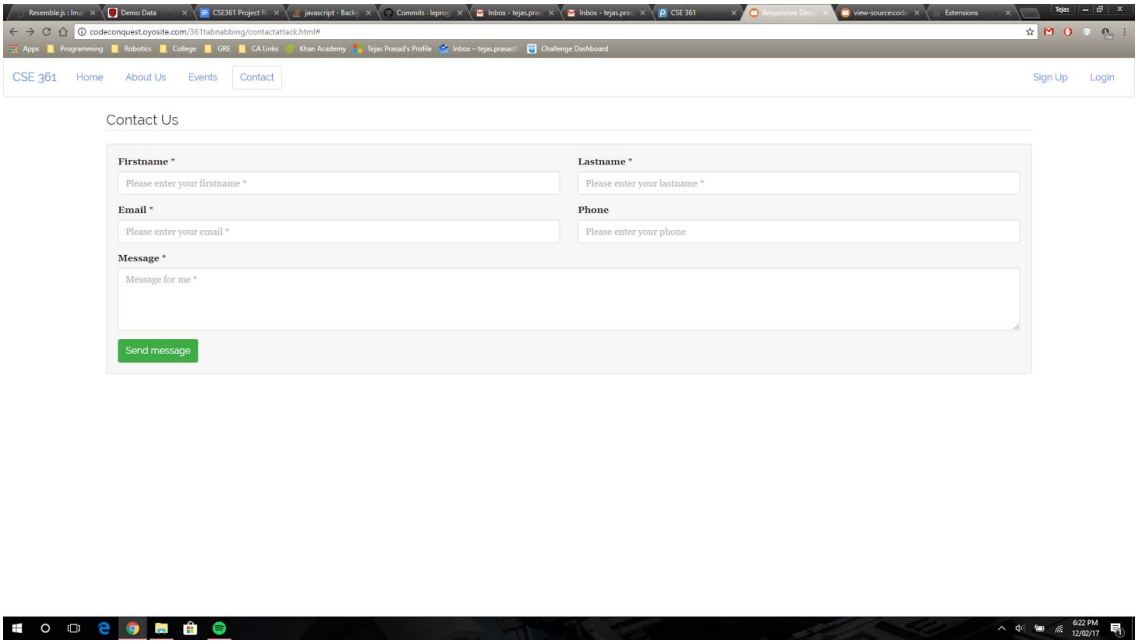
Once the tab is no longer focused, the background page no longer takes screenshots of it. When the tab is then focused again, the content script sends a message to the background script asking for a new screenshot and the previous saved screenshot. The content script uses the Resemble.js library to conduct the comparison and then checks to see if the mismatch percentage of the two images is greater than 0. If it is not, then the canvas element is not created and only the extension badge on the toolbar is updated to show there is no change. If it is larger than 0, a canvas element is created and the image is drawn onto it. Additionally the background script stops the timer if the mismatch percentage is greater than 0 so the extension does not take any screenshots with the canvas visible. This element is then added to the webpage and the extension badge is updated with the mismatch percentage and a color coding based on the percentage. When the canvas element is clicked on, it disappears and the background script receives a message to restart the timer so the extension can continue to take screenshots to detect any other changes.

Site after tabnabbing without extension

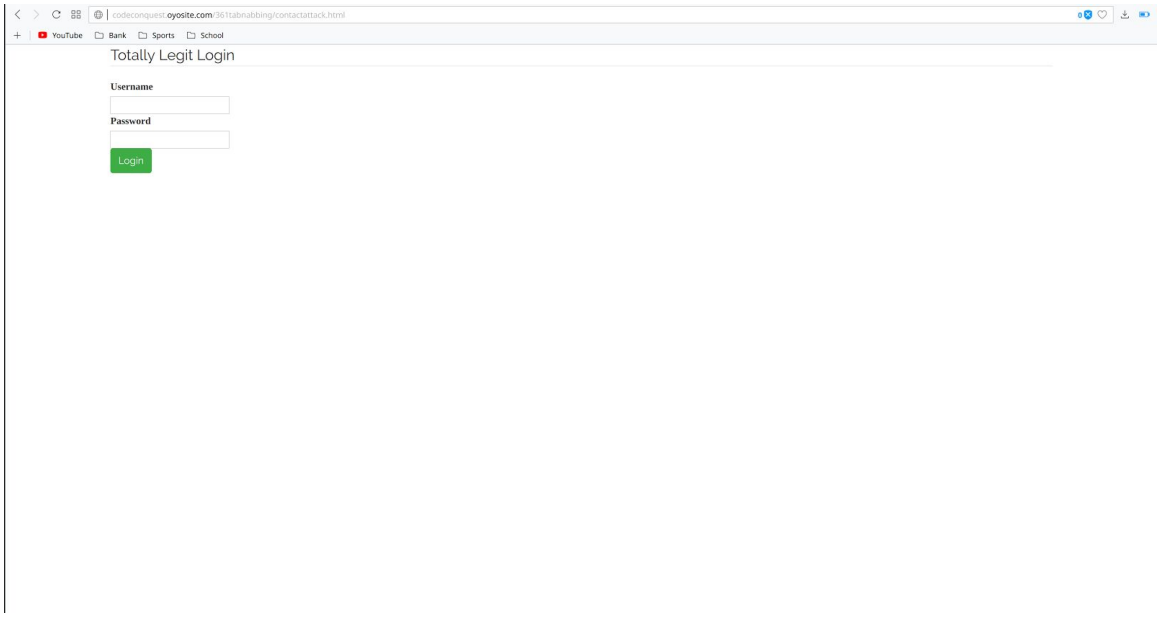
The below image shows what the website will now show with our extension activated. You can see that since after the example attack the website is completely blank the highlights in pink show everything from the first screenshot. You can also notice in the toolbar that the extension's icon shows a pink label with the mismatch percentage.



We made the following website as an example to show how tabnabbing could affect users and put their credentials at risk. The website is originally a contact form for a seemingly harmless page but once the user clicks away then it transforms into a 'totally legit login' page. Note that a real attack would transform into a much more legitimate clone of a real service but this website is just for example purpose. If a user has just clicked back onto this tab and been a way for a while they might not realize what was previously there and just assume this service logged them out so they will log back in. Unknowingly to them this malicious site might send their credentials to an attackers database. That's what the example app does and as you can see here in these example pictures the credentials are stored in a firebase database.

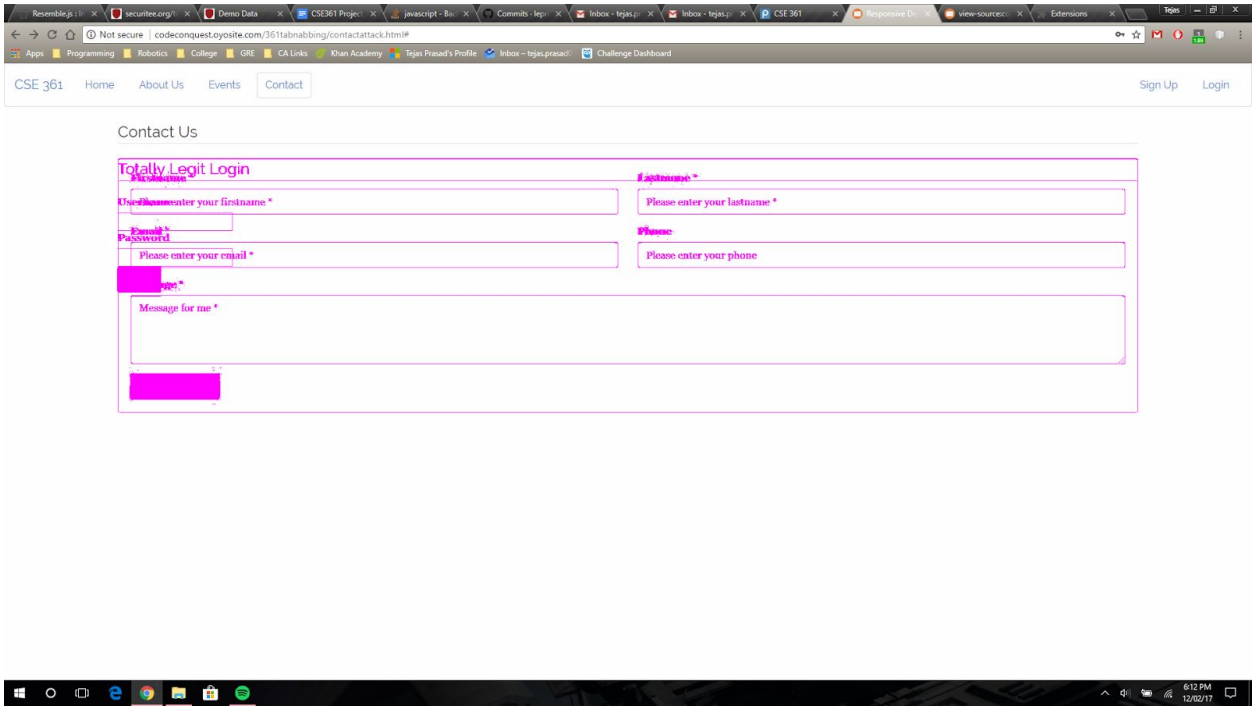


Pictured Above: The Original Site



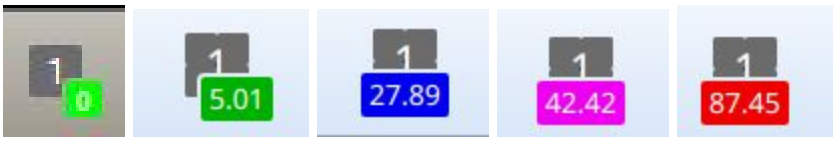
Pictured Above: The Changed Site after going to a different tab

A normal user might login in to this page and then their credentials could be stolen into a database by the attacker but with our extension the user will instead see this.



The user is now informed that a phishing attempt has been made and can safely navigate away from the attack without exposing their credentials.

Below is our extensions toolbar icon and we wanted to find a way to alert the user to how much the page has changed so we put text on the icon alerting them to a percentage of mismatch which is returned by resemble and we also color coded the badge.



3. Design Decisions-

One of the first design decisions we had was how to setup the extension initially. We decided on using the content and background script mechanism as it was similar to a client and server system which we were familiar with. The mechanism of querying the background script by sending a key/value pair as a message allowed us to have only one background script. We only executed specific blocks of code based on the existence or nonexistence of specific key/value pairs in the message.

When comparing the screenshots, we decided to use the Resemble.js library suggested in the project description. The library allowed us to select which differences we wanted to highlight. Additionally when comparing Resemble.js to other JavaScript image comparison libraries, it was the only one which did not require the images to be the same size and included a function to scale the images if they were not the same size. Additionally we decided against cutting the images up into blocks when comparing the two images. This is because the Resemble.js library worked well enough without cutting the images into multiple blocks and the resulting improvement we would get would be insignificant.

Background Script:

In the background script, we decided to store the screenshot with the URL as the key in the dictionary. This is because the URL of the site will never change in a TabNabbing attack. The function which handles taking the screenshot is set to be called every 10 seconds. We decided on 10 seconds as it allows the extension to have data is still relatively fresh but does not need more resources to run the extension. Additionally the badge on the extension icon is set to not be displayed within this function so other tabs do not display the feedback of a tab which already has shown results. For the badge on the extension icon in the toolbar, we decided to display the mismatch percentage of the screenshots to the user so they would have more information then just the image on the canvas. A color coding of green to blue to purple to red was used to display the different ranges in the mismatch percentage. Light green was used to show a mismatch of 0%, dark green for a mismatch between 0% and 10%, blue for between 10% and 30%, purple for between 30% and 50%, and red for a mismatch percentage greater than 50.

Content Script:

In the content script we had to decide how to display the differences given by the Resemble.js library. Initially we decided to create an overlay using the div tag and insert the image into it. We also tested creating the overlay with the canvas element and decided to keep the canvas element over the div element as it allowed us to actually draw the image on the overlay instead of just inserting the image into the overlay.

4. Project Breakdown-

This project was completed by Tejas Prasad and Tyler Maciaszek. Tejas worked on displaying the differences via creating a canvas element and inserting it into the DOM, displaying feedback to the user using the extension badge on the toolbar, and writing the project report. Tyler worked on the using the chrome API to take screenshots on a regular

interval, send and receive messages to pass data back and forth from content to background, debug edge cases for the extension and make the powerpoint presentation.