CSE 361 Project Report

Project 2: TabNabbing

Group Name: Shellphish

Varun Chembukkavu 109937670

Jayesh Ranjan 109962199

INTRODUCTION

Tabnabbing is a variation of a phishing attack that was first proposed by Aza Raskin in 2010. Phishing is the process of making a vulnerable victim click on a link that takes the user to page that look like one they are familiar with (like a Bank of America login or a Gmail Login page). The victim's credentials are stolen when he enters them suspecting that the page is actually the real website. Far often than not, looking at the URL before entering credentials mitigates this attack. On the other hand Tabnabbing is more subtle and works by changing a tab that is in the background. This is harder to detect since the user has already visited this page and likely to trust it. The user may suspect he may have been logged out and is likely to submit their credentials without looking at the URL. The user may have originally visited a harmless page that ended up changing in the background when the user was working on another tab. This attack dubbed 'Tabnabbing' works in the following manner.

1) The first involves changing the title of the page.
2) Second, the favicon is changed.

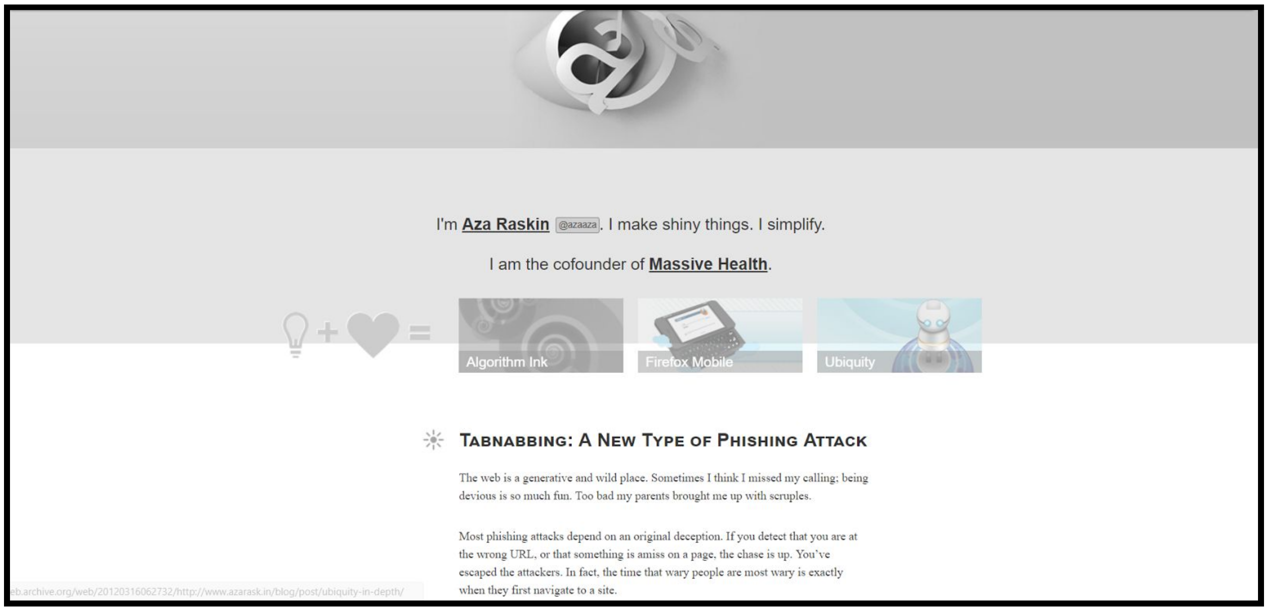
3) Finally the look and feel of the website is changed.

The website that intends to change in the background may use different strategist to make the user move to another tab and forget about the current one. An attacker might make a very long but interesting blog post that user may choose to not complete and move to another tab. Similarly, a user may be led to open a link in a new tab. This allows the attacker to change the title, favicon and the look and feel of the page. For example the page may be changed to a Gmail login. An unsuspecting user may switch to this tab and login to Gmail again.

Aza Raskin's blog post about this attack also serves as a great demo. At first glance Raskin's blog looks interesting but harmless. However upon moving to another tab, Raskin switches the title, favicon and page to the Gmail website, Raskin talks about a CSS History miner that can be used to find out which websites are most visited by a particular user. Based on the data generated from this CSS History miner, an attacker can carefully craft a webpage and change favicon on the fly. This is especially harder to defend against when it is the login of a Bank website. Bank websites frequently log users out after a period of inactivity. Crafting a page that looks like a bank's session timeout page, can lead an unsuspecting user to submit their credentials for a second time.. The attacker may or may not choose to log them in their actual bank account while spoofing the bank credentials.

The image below is a screenshot of Aza Raskin's blog post at

http://www.azarask.in/blog/post/a-new-type-of-phishing-attack/


I'm **Aza Raskin** @azaaza. I make shiny things. I simplify.

I am the cofounder of **Massive Health**.

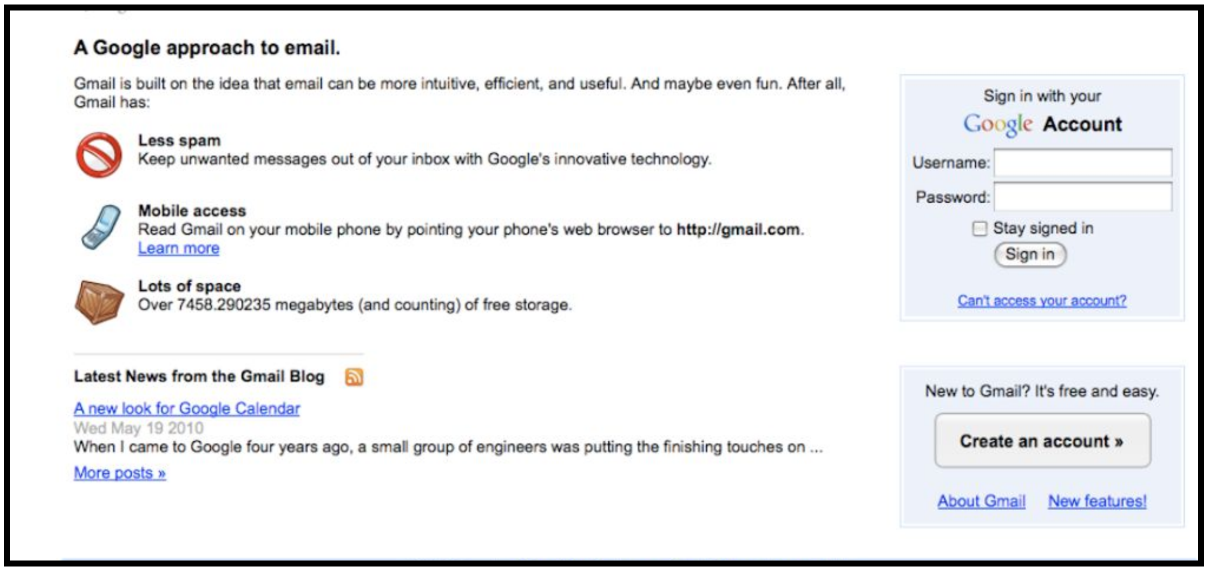
Tabnabbing: A New Type of Phishing Attack

The web is a generative and wild place. Sometimes I think I missed my calling; being devious is so much fun. Too bad my parents brought me up with scruples.

Most phishing attacks depend on an original deception. If you detect that you are at the wrong URL, or that something is amiss on a page, the chase is up. You've escaped the attackers. In fact, the time that wary people are most wary is exactly when they first navigate to a site.

eb.archive.org/web/20120316062732/http://www.azarask.in/blog/post/ubiquity-in-depth/

A user may move on to another tab leaving this tab open. Silently Raskin changes the page to a GMail login through JavaScript.


A Google approach to email.

Gmail is built on the idea that email can be more intuitive, efficient, and useful. And maybe even fun. After all, Gmail has:

Less spam
Keep unwanted messages out of your inbox with Google's innovative technology.

Mobile access
Read Gmail on your mobile phone by pointing your phone's web browser to **http://gmail.com**.
Learn more

Lots of space
Over 7458.290235 megabytes (and counting) of free storage.

Latest News from the Gmail Blog

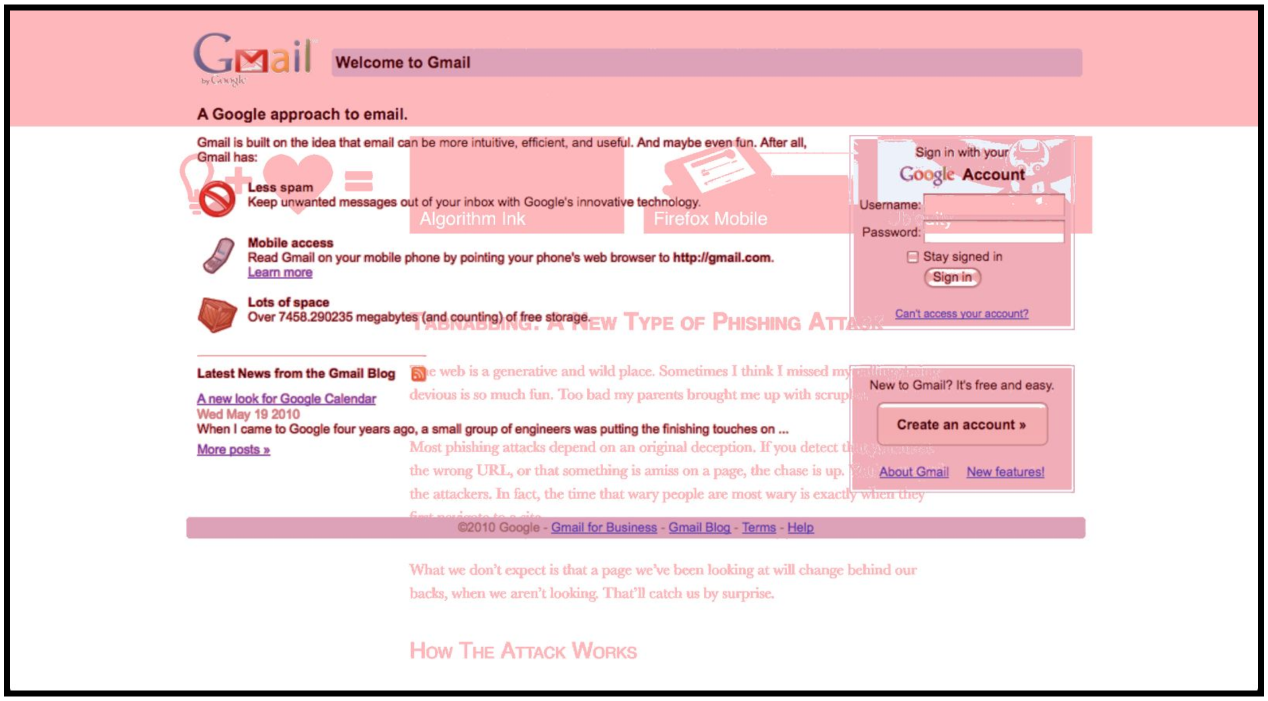
A new look for Google Calendar
Wed May 19 2010
When I came to Google four years ago, a small group of engineers was putting the finishing touches on ...
More posts »

This looks exactly how an actual Gmail login page looked in 2010 and a user might actually give away their credentials. This attack could be reproduced with any website and once phished an attacker may test the same credentials on other websites for password reuse.

COUNTERMEASURE

As a countermeasure against this tabnabbing attack, we propose a defence strategy involving a Google Chrome extension, **TabMatch** that detects this change and alerts the user by highlighting the changes in red. We came up with idea after debating multiple defence strategies for this attack. A Javascript based detection by analyzing the DOM was one method to detect the attack. But Javascript is very volatile and it is incredibly difficult to keep track of these changes. In the end, we went ahead with a simpler approach that involved taking screenshots at regular intervals. When the user revisits an earlier tab, we use an image comparison library Rembrandt (https://github.com/imgly/rembrandt) to analyze these changes and display an overlay with the highlighted changes in red. To make things as unobtrusive as possible, the user can just click on the overlay to make it disappear. We thought about adding a close button but decided against that to make the experience better for the user. The snapshots are taken every 5 seconds. The extension works on the assumption that the user has not scrolled through the website since the last snapshot was taken for the given tab. This assumptions necessary to avoid false positives as the image will be different if the snapshot was taken after the user scrolled. The implementation consists of adding an overlaying a **translucent composition image** of the changes from the original website and the phishing website that is displayed on top of the webpage.

The resulting image looks similar to this:



Welcome to Gmail

A Google approach to email.

Gmail is built on the idea that email can be more intuitive, efficient, and useful. And maybe even fun. After all, Gmail has:

Less spam
Keep unwanted messages out of your inbox with Google's innovative technology.

Mobile access
Read Gmail on your mobile phone by pointing your phone's web browser to http://gmail.com.
Learn more

Lots of space
Over 7458.290235 megabytes (and counting) of free storage.

Latest News from the Gmail Blog
A new look for Google Calendar
Wed May 19 2010
When I came to Google four years ago, a small group of engineers was putting the finishing touches on ...
More posts »

Sign in with your Google Account
Username:
Password:
Stay signed in
Sign in
Can't access your account?

New to Gmail? It's free and easy.
Create an account »
About Gmail New features!

©2010 Google - Gmail for Business - Gmail Blog - Terms - Help

Tabnabbing: A New Type of Phishing Attack

The web is a generative and wild place. Sometimes I think I missed my calling; being devious is so much fun. Too bad my parents brought me up with scruples.

Most phishing attacks depend on an original deception. If you detect that you are at the wrong URL, or that something is amiss on a page, the chase is up. You've escaped the attackers. In fact, the time that wary people are most wary is exactly when they first navigate to a site.

What we don't expect is that a page we've been looking at will change behind our backs, when we aren't looking. That'll catch us by surprise.

How The Attack Works

This is a screenshot taken from **Ruskin's tabnabbing website** following changes.

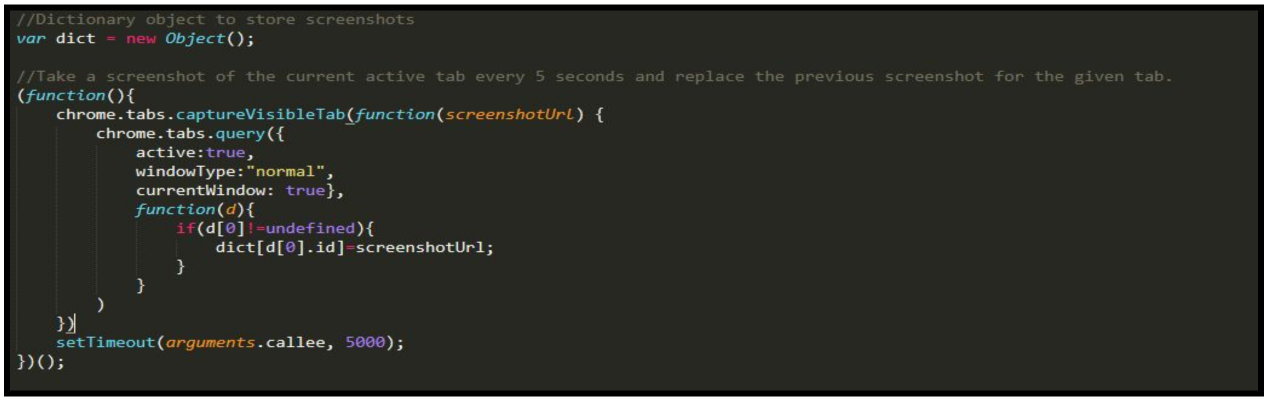
IMPLEMENTATION

As stated earlier **RembrandtJS** was used for image comparison and to create a **composition image**. The composition image generated by the library was trimmed to just include the difference between the two images and not the original images. The above image shows the the old website's layout overlayed on top of the current website's layout.

We used chrome's Tabs API and the **captureVisibleTab** function to take screenshot of the page every 5 second. If a user opens up a new tab or visits a tab whose ID is not listed in the **dict Object**, then the ID is added. The dict Object serves as dictionary of the tabId to the screenshot of that particular tab. When a tab becomes active and we have a previous screenshot of the same tab, we check the screenshot saved with the ID of the tab in our **dict Object**. If the screenshots differ, then we show the difference by **overlaying** the **composite image** on top of the changed website. Since this is an overlay and sometimes may not be visible to user, if the change is minimal.

The source code has been shown below

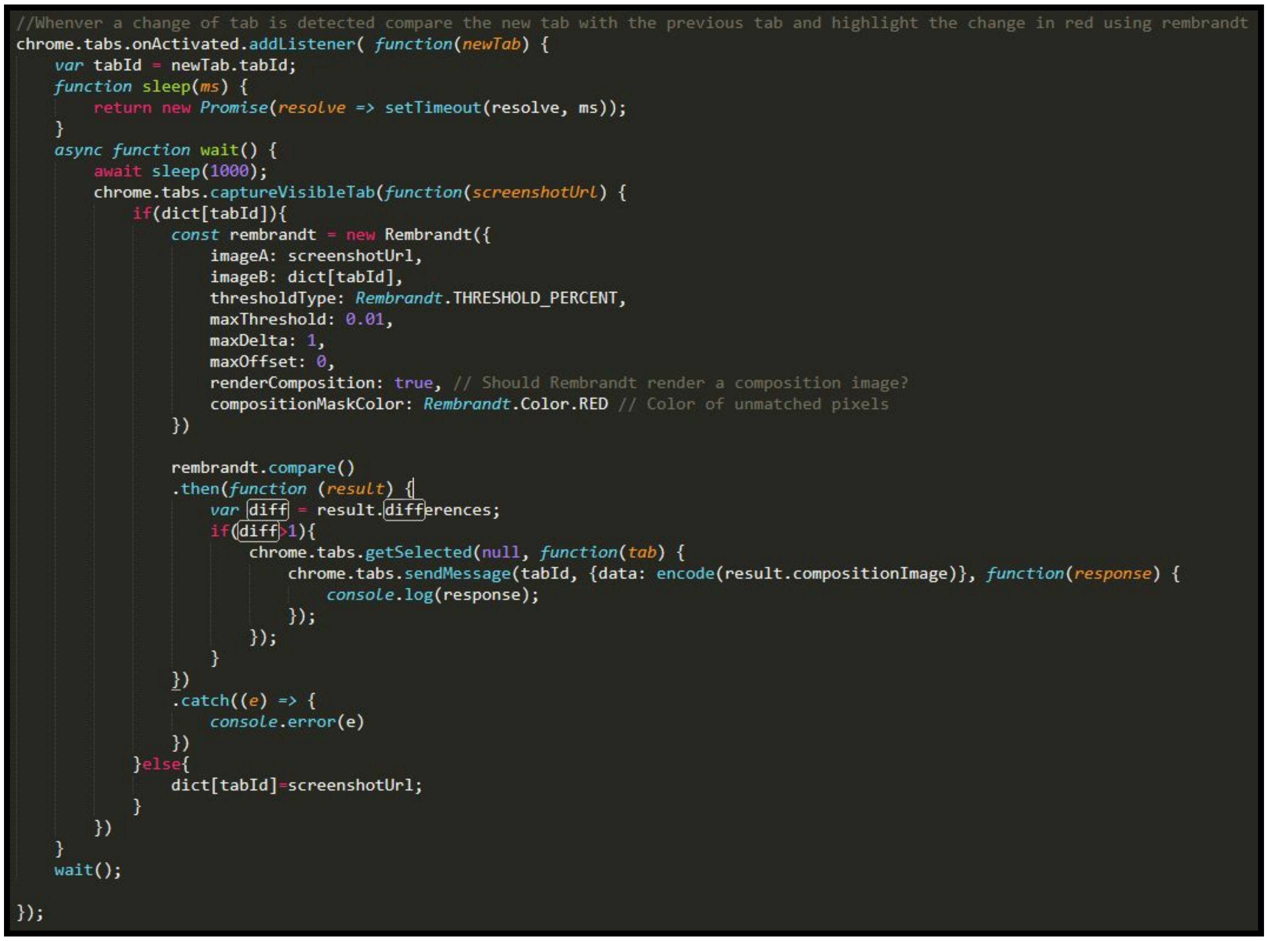
This is the code we used for our **background script**, and it's part of **background.js**, This part of the extension was written by Varun Chembukkavu



```
//Dictionary object to store screenshots
var dict = new Object();

//Take a screenshot of the current active tab every 5 seconds and replace the previous screenshot for the given tab.
(function(){
    chrome.tabs.captureVisibleTab(function(screenshotUrl) {
        chrome.tabs.query({
            active:true,
            windowType:"normal",
            currentWindow: true},
            function(d){
                if(d[0]!=undefined){
                    dict[d[0].id]=screenshotUrl;
                }
            }
        )
    })
    setTimeout(arguments.callee, 5000);
})();
```

This is where we capture the screenshot every 5 seconds and save it with the respective **tabID**.



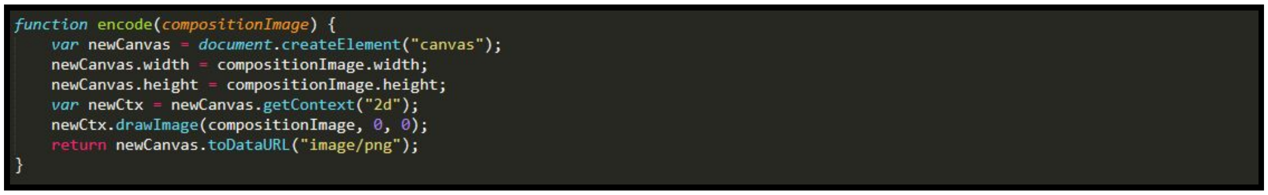
```
//Whenver a change of tab is detected compare the new tab with the previous tab and highlight the change in red using rembrandt
chrome.tabs.onActivated.addListener( function(newTab) {
    var tabId = newTab.tabId;
    function sleep(ms) {
        return new Promise(resolve => setTimeout(resolve, ms));
    }
    async function wait() {
        await sleep(1000);
        chrome.tabs.captureVisibleTab(function(screenshotUrl) {
            if(dict[tabId]){
                const rembrandt = new Rembrandt({
                    imageA: screenshotUrl,
                    imageB: dict[tabId],
                    thresholdType: Rembrandt.THRESHOLD_PERCENT,
                    maxThreshold: 0.01,
                    maxDelta: 1,
                    maxOffset: 0,
                    renderComposition: true, // Should Rembrandt render a composition image?
                    compositionMaskColor: Rembrandt.Color.RED // Color of unmatched pixels
                })

                rembrandt.compare()
                .then(function (result) {
                    var diff = result.differences;
                    if(diff>1){
                        chrome.tabs.getSelected(null, function(tab) {
                            chrome.tabs.sendMessage(tabId, {data: encode(result.compositionImage)}, function(response) {
                                console.log(response);
                            });
                        });
                    }
                })
                .catch((e) => {
                    console.error(e)
                })
            }else{
                dict[tabId]=screenshotUrl;
            }
        })
    }
    wait();

});
```

The moment the user returns to the page, a screenshot is captured. Using **RembrandtJS** library, we put the new screenshot and the screenshot we saved with the tab's ID (if captured before) as **imageA** and **imageB**, respectively and create a new Object. **RembrandtJS** has a **compare** function which we use to compare the results. The differences variable gives us the number of pixels that are different. We made it extra sensitive, so if there is a difference of more than 100 pixels, we send the composition image to the content script which is running in the foreground.

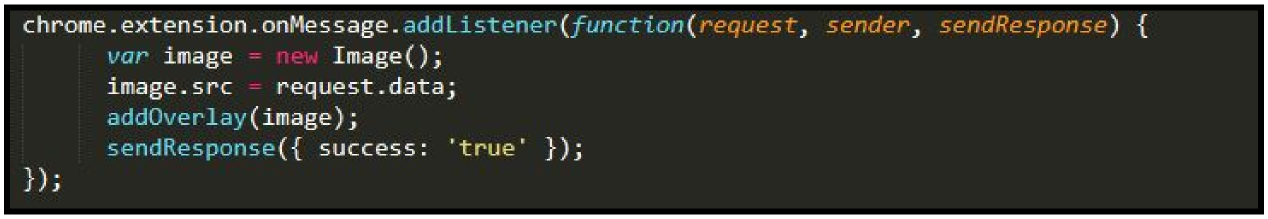
In order to send the image as a message to the content.js file, the image was converted to a **DataURI**. We used **encode()** as an auxiliary function for this purpose.



```
function encode(compositionImage) {
    var newCanvas = document.createElement("canvas");
    newCanvas.width = compositionImage.width;
    newCanvas.height = compositionImage.height;
    var newCtx = newCanvas.getContext("2d");
    newCtx.drawImage(compositionImage, 0, 0);
    return newCanvas.toDataURL("image/png");
}
```

We used HTML's canvas element to convert the data to URL. Which we pass as **MessegeSender** parameter to Chrome's **tabs.sendMessage function**, in which we added the id as **data** and **url** as the **url** returned by **encode()** function.

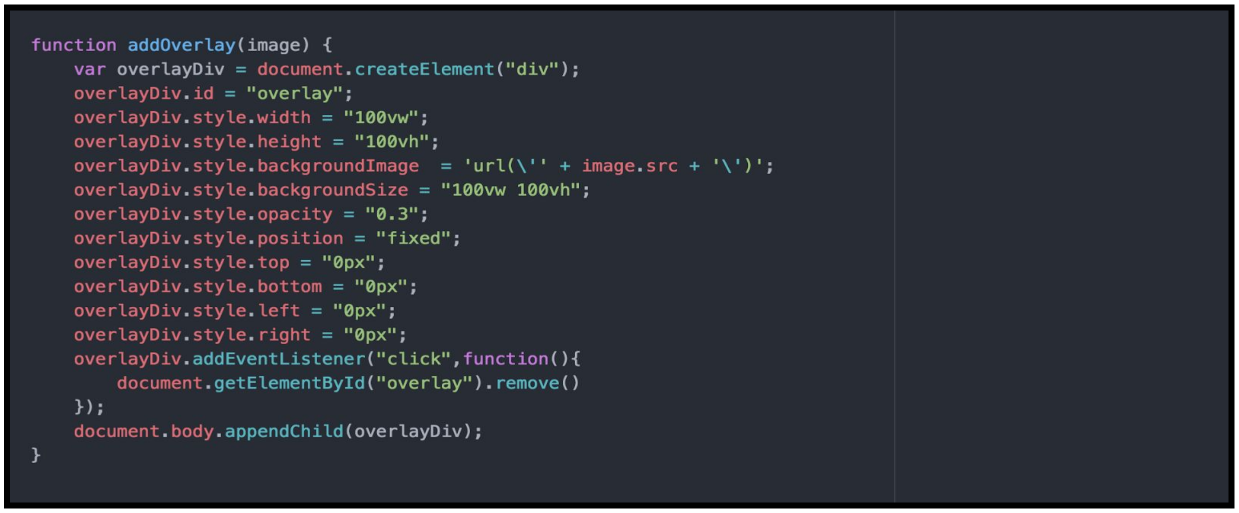
This is the code we used for our content script. Jayesh Ranjan worked on the message parsing and displaying the overlay on the page.



```
chrome.extension.onMessage.addListener(function(request, sender, sendResponse) {
        var image = new Image();
        image.src = request.data;
        addOverlay(image);
        sendResponse({ success: 'true' });
});
```

addListener() function is from chrome's API of content scripts, which is typically used for message passing from **background** to **content.** The functions listens for messages passed from **background script's tabs.sendMessage()** function. Any message sent from **background** is sent to **request,** from which we extract the **value** from the **key** we originally used in **sendMessage()'s MessegeSender** parameter. In this case we have **data**. An **image object** is created from the **URL** that was passed as **data** and the **image object** is passed to **addOverlay() function** which we use to add **overlay** on the entire page.

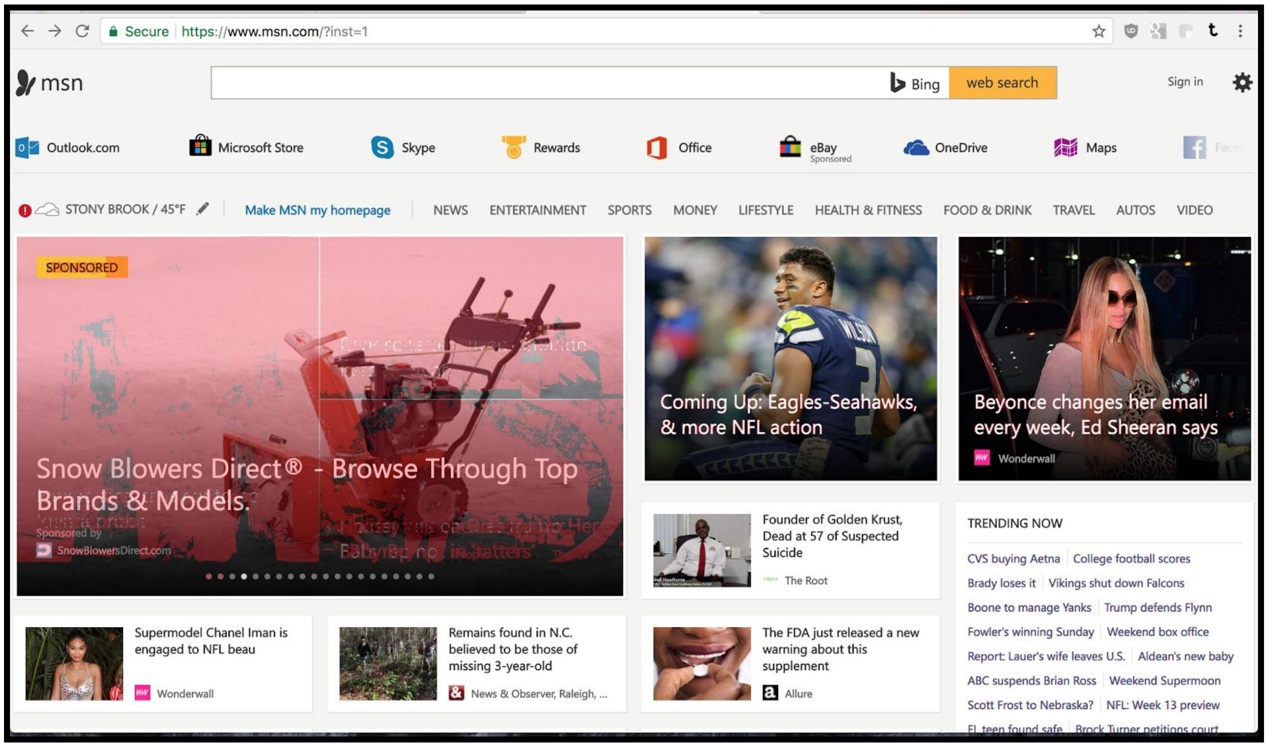
Below we have implementation of **addOverlay() function** which takes an **image object** as parameter.



```
function addOverlay(image) {
    var overlayDiv = document.createElement("div");
    overlayDiv.id = "overlay";
    overlayDiv.style.width = "100vw";
    overlayDiv.style.height = "100vh";
    overlayDiv.style.backgroundImage  = 'url(\'' + image.src + '\')';
    overlayDiv.style.backgroundSize = "100vw 100vh";
    overlayDiv.style.opacity = "0.3";
    overlayDiv.style.position = "fixed";
    overlayDiv.style.top = "0px";
    overlayDiv.style.bottom = "0px";
    overlayDiv.style.left = "0px";
    overlayDiv.style.right = "0px";
    overlayDiv.addEventListener("click",function(){
        document.getElementById("overlay").remove()
    });
    document.body.appendChild(overlayDiv);
}
```

We first check if any element with the **id of overlay** is present in the page. If not, we add our own overlay. This is to prevent double or triple **overlaying** by our own function when the page changes, while we are displaying the **translucent overlay**.. The **opacity of the overlay div** is set to **0.3** in order to make the **overlay image** translucent. To ensure that the user can return back to the page he/she was browsing on (and not permanently block the page from being viewed again). We added functionality to remove the **overlay div**. The user can dismiss the overlay by just clicking on it. Although the user may accidently dismiss the div, the chances of this happening are minimal. The main purpose of the div is to just notify the user.

<u>Another Example on msn.com</u>



Snow Blowers Direct® - Browse Through Top Brands & Models.
Sponsored by SnowBlowersDirect.com

Coming Up: Eagles-Seahawks, & more NFL action

Beyonce changes her email every week, Ed Sheeran says
Wonderwall

Founder of Golden Krust, Dead at 57 of Suspected Suicide
The Root

Supermodel Chanel Iman is engaged to NFL beau
Wonderwall

Remains found in N.C. believed to be those of missing 3-year-old
News & Observer, Raleigh, ...

The FDA just released a new warning about this supplement
Allure

CVS buying Aetna | College football scores
Brady loses it | Vikings shut down Falcons
Boone to manage Yanks | Trump defends Flynn
Fowler's winning Sunday | Weekend box office
Report: Lauer's wife leaves U.S. | Aldean's new baby
ABC suspends Brian Ross | Weekend Supermoon
Scott Frost to Nebraska? | NFL: Week 13 preview

The above image is another example of **TabMatch** is action. **TabMatch** highlights dynamic content on the page in red when the user returns to the page.

SUMMARY

A quick summary of the implementation

1) A User opens a new tab. (Tab1)
2) A Screenshot is taken and saved with associated ID for the tab
3) User chooses to browse and leaves the current active tab (Tab1) and goes to another tab (Tab2)
4) Meanwhile Tab1 changes without without the user knowing.(since the user can't see it)
5) The User goes back to Tab1 and another screen shot is taken.

6) RembrandtJS compares the old screenshot and the new screenshot.
7) If they are different the composition image is passed to content script.
8) The image is overlaid on top of Tab1.
9) The User can choose to close the overlay by clicking on the div.

CONCLUSION AND FUTURE IMPROVEMENTS

TabMatch is currently a very basic extension. There is a lot of room for improvement. Since the Chrome API doesn't provide a way to take a screenshot of the old tab when a user switches tabs, we have to resort to periodic screenshots. The **OnActivated** API only lets you know the id of the current tab. One way would be to revisit the old tab, take the screenshot and return. But this would degrade the user experience by causing a flicker. Taking a screenshot when the user leaves is far more efficient.. Our current implementation is based on comparing images of when the user left the tab, and after the user came back to the tab. However, this approach is flawed since this can only effectively work if the pages are static or where the user last left off was static. For example; if screenshot was taken just before a user scrolled down and left the page, a false positive can cause the whole page to be highlighted and ruin the user experience. One idea to combat this would be to take full page screenshots but the Chrome API again does not provide a way to do this currently. For now, the frequency of the screenshots is set to 5 seconds. This means that the browser is forced to take screenshots and reduce the performance of the browser. If the frequency is increased the scrolling issue will be more apparent. On the other hand increasing the frequency means data and space tradeoffs. The user might not want to cache so many images in the browser. These tradeoffs come at the cost of increasing browser security. The best possible scenario without impacting performance would be for this feature to be built in to the browser without the need for additional extensions.