All Your Screens are Belong to Us: Attacks Exploiting the HTML5 Screen Sharing API

Yuan Tian*, Ying-Chuan Liu‡, Amar Bhosale†, Lin-Shung Huang*,Patrick Tague†,Collin Jackson*
Carnegie Mellon University
*{yuan.tian, linshung.huang, collin.jackson}@sv.cmu.edu,†{amarb, tague}@cmu.edu,‡{kateycliu}@gmail.com

***Abstract*—HTML5 changes many aspects in the browser world by introducing numerous new concepts; in particular, the new HTML5 screen sharing API impacts the security implications of browsers tremendously. One of the core assumptions on which browser security is built is that there is no cross-origin feedback loop from the client to the server. However, the screen sharing API allows creating a cross-origin feedback loop. Consequently, websites will potentially be able to see all visible content from the user's screen, irrespective of its origin. This cross-origin feedback loop, when combined with human vision limitations, can introduce new vulnerabilities. An attacker can capture sensitive information from victim's screen using the new API without the consensus of the victim. We investigate the security implications of the screen sharing API and discuss how existing defenses against traditional web attacks fail during screen sharing. We show that several attacks are possible with the help of the screen sharing API: cross-site request forgery, history sniffing, and information stealing. We discuss how popular websites such as Amazon and Wells Fargo can be attacked using this API and demonstrate the consequences of the attacks such as economic losses, compromised account and information disclosure. The objective of this paper is to present the attacks using the screen sharing API, analyze the fundamental cause and motivate potential defenses to design a more secure screen sharing API.**

I. Introduction

Web browsers have evolved from applications that render simple web pages to application platforms that handle complex media, which often require installation of plugins, such as the Google voice and video chat plugins [1]. Recently, the Web Real-Time Communications (RTC) Working Group [2] proposed APIs that allow real time communication between browsers using only HTML5 APIs via JavaScript. Audio and video communication, including screen sharing, can be set up using this RTC platform [3]. The Google Chrome browser supports screen sharing as an experimental feature. It is also available in Mozilla Firefox's nightly build. Moreover, this API is interoperable between Google Chrome and Firefox Nightly with a slight modification to the calling site [4].

The screen sharing API could be utilized to build interactive media tools and applications, however, the possibility of sharing the screen with other web browsers and servers raises various security and privacy concerns. Specifically, the screen sharing API creates a cross-origin feedback loop that continuously transmits the screen back to the screen sharing website which belongs to other domains. This loop allows other users and the screen sharing website to see the user's confidential information. The fundamental assumption for the web is that this cross-origin feedback loop does not exist by default; in most cases, what is shown to the user is only accessible to that user and the same-origin site. With the loop, however, an attacker can exploit this information and launch attacks against the user sharing the screen. The integrity and confidentiality of the user's information are severely threatened.

The attacker who has access to the victim's screen is very analogous to a shoulder-surfer [5] but with the additional power of opening and viewing the web pages of the attacker's choice inside the user's browser without the user's consent. Since the browser automatically sends authentication information (e.g. HTTP cookies), the newly opened pages that are only supposed to be seen by an authenticated user can now be seen by the attacker. We show that the attacker can steal sensitive information such as personal messages and bank statements from the attacker-opened pages. Even worse, we show that cross-site request forgery (CSRF) tokens can be stolen from the page source, which can then be used to mount further attacks against the user.

In this paper, we discuss the extensions to the *getUserMedia()* API introduced by Google to enable screen sharing and their potential impact on user security. We present multiple attacks using this API that can compromise a user's confidentiality and integrity. We show how an attacker can steal a user's data from the screen to modify the user's accounts on popular websites. Since the attackers have the ability to see the entire content of the user's screen, the attack can not be restricted by the same-origin policy. Because of the limitations of human sensing, a clever attacker can steal cross-origin page content by forcefully opening cross-origin pages on the victim's browser or embedding cross-origin content inside an iframe. Making the attacks unobservable such as flashing content quickly or using translucent color, the attacker can eavesdrop cross-origin information without drawing attention. We also discuss how the attacker can steal browser resources such as autocomplete and autofill history [6] since these resources are shared across origins for the user's convenience. These attacks can lead to serious consequences such as loss of money from bank accounts, personal information disclosure and identity theft. Simply put, the attacker can not only see what the user wants him to see but also force to user to show other sensitive information.

With the knowledge of the fundamental causes of these attacks, we analyze the possibility of preventing them by a fine-grained screen sharing, use of incognito mode, and blocking third-party cookies. We compare these solutions with a focus on

security and usability. Since the paper is primarily concerned about attacks using the screen sharing API, we leave the implementation and evaluation of these defenses as promising future work.

Summary of Contributions

- We have analyzed the possible impact of the screen sharing API on user privacy and security. Based on our experiments and understanding about the current design of the screen sharing API, we present the security implications and vulnerabilities that this API could possibly introduce.
- Based on the vulnerabilities that we have discovered, we show how malicious screen sharing websites can steal cross-origin content from the user's browser, thus affecting privacy, confidentiality and integrity. We demonstrate how an attacker can perform a CSRF attack even if the target website has employed CSRF defenses, and how the attacker can steal other sensitive user information. We also discuss how these attacks can be made imperceptible to human by exploiting the limitations of human sensing.
- We raised an alarm to the browser vendors about the perils of view-source links. In particular, we highlighted the risk of malicious screen sharing websites gaining unauthorized access to security credentials by viewing the source of cross-origin websites. As a result of our security report, Google [7] patched the Chrome browser, which prevents view-source links to be opened inside iframes.

Organization. The rest of the paper is organized as follows. In Section II, we discuss various screen sharing techniques along with their security implications. We present the threat model in Section III, and demonstrate attacks using the screen sharing API in Section IV. In Section V, we analyze the deficiencies of existing defenses and propose possible defenses to mitigate the attacks using the screen sharing API. We discuss related work in Section VI and conclude in Section VII.

II. Screen Sharing Techniques

Various solutions have been developed to achieve screen sharing functionality in browsers. We classify these solutions into three categories:

- **Techniques that Require Installation of Plugins or Extensions:** Popular products such as Google+ allow their users to share their screens with others after installing the Google voice and video chat plugin [1]. Additionally, the screen can be shared using the Chrome extension API and WebSockets [8]. The API continuously captures the screenshots as images and transmits the images to other sites via WebSocket. The Chrome extension API chrome.desktopCapture is only accessible to whitelisted extensions with whitelisted origins.
- **Techniques that Clone the Document Object Model (DOM) to Simulate Screen Sharing:** Another option to share a user's tab is transmitting an HTML DOM object to other sites. In [9], the author describes two methods to do so: using Mutation Observers to monitor the changes of DOM objects, and mirroring the entire HTML DOM.
- **New HTML5 Screen Sharing API:** The extension of the *getUserMedia()* API built on WebRTC allows a user's screen to be shared via JavaScript as illustrated in Figure 1. The detailed usage of the *getUserMedia()* API is introduced in [10][1]. We use Chrome 26 implementation as an example to explain the screen sharing API here. Figure 2 shows the initiation of a screen sharing session. There are two things to note. One is that the API can work only over SSL connections. The other is that the browser asks for the user's permission before it starts capturing the screen. After the user grants the permission to share the screen, the API will capture the entire visible area of the screen including the area outside the browser. Also, the browser flashes a red notification on the tab icon to alert the user that his screen is currently being shared in this tab, as illustrated in Figure 3.

```
var share = function share () {
  var constraints = {
    video: {
      mandatory: {
        chromeMediaSource: 'screen'
      }
    }
  };
  navigator.getUserMedia =
    navigator.webkitGetUserMedia ||
    navigator.getUserMedia;
  navigator.getUserMedia(constraints,
    onSharingSuccess, onSharingError);
};
```

Fig. 1. The usage of the screen sharing API: Google Chrome exposes the chromeMediaSource constraint for developers to specify the video source. The constraint is used as an argument of getUserMedia(). If the media source is set as 'screen', getUserMedia() would capture the entire screen into the media stream.

Security Implications. Due to the fundamental differences among screen sharing implementations, the new HTML5 API has a very different security implication from the traditional screen sharing techniques. In comparison to the screen sharing techniques that require the user to download a plugin or an extension, the site that uses the screen sharing HTML5 API runs inside the browser and has the ability to control its own page as well as open cross-origin content inside an iframe or a new window. Although the browser's same-origin policy prevents the malicious site from accessing cross-origin content

[1]In Chrome 26, the user needs to enable screen capture support by setting "Enable screen capture support in getUserMedia()" in chrome://flags.

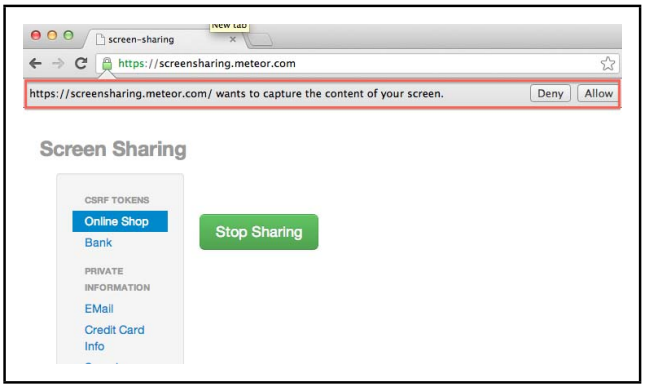


Fig. 2. The browser requests the user's permission for sharing his screen with the host website. The screen sharing does not begin until the user grants the permission.

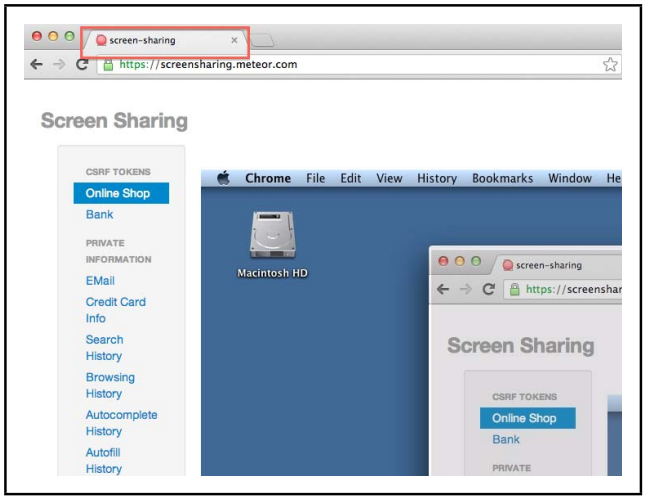


Fig. 3. Once the user provides permission to capture the screen, the browser flashes a red notification on the tab.

directly using JavaScript, using the new screen sharing API, the malicious site can see the content belonging to cross-domains. The malicious site can misuse this ability and force the browser to open sensitive resources such as an email inbox page or a victim's bank account statement page inside iframes or in new windows and capture the cross-origin content rendered inside these new windows. Since the browser automatically sends cookies for the respective sites, if the user is already logged into those sites, the iframes embedded inside attacker's screen sharing website or new windows opened by the attacker will reveal sensitive user information to the screen. Although sensitive user information can be collected by an attacker's server, this information stealing may be imperceptible to victims because of the limitation of human vision. For example, the attacker can hide the malicious behavior by playing tricks such as making content almost transparent or fairly small or flash content fast. As is illustrated in Figure 4, the user's vision is different from the attacker's capture. Note that even if the user notices the attack, they cannot prevent against the attack because the sensitive data has been stolen by the attacker when the user sees the attack.

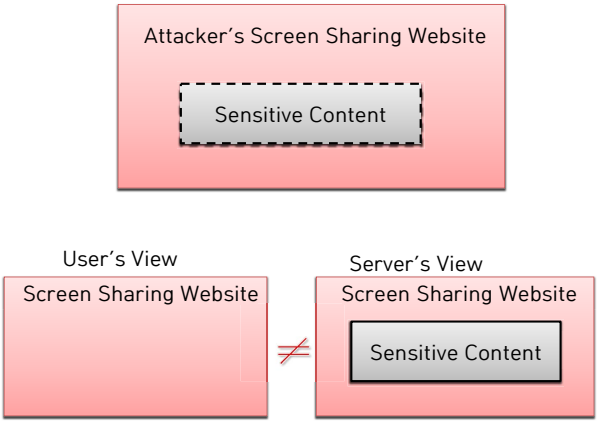


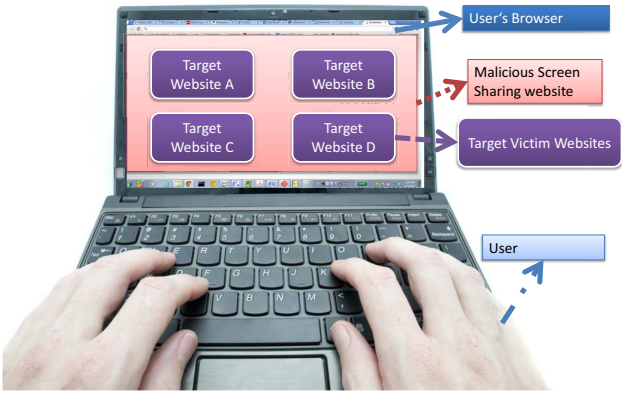
Fig. 4. The limitation of human vision allows the attacker to hide the information stealing process.

III. Threat Model

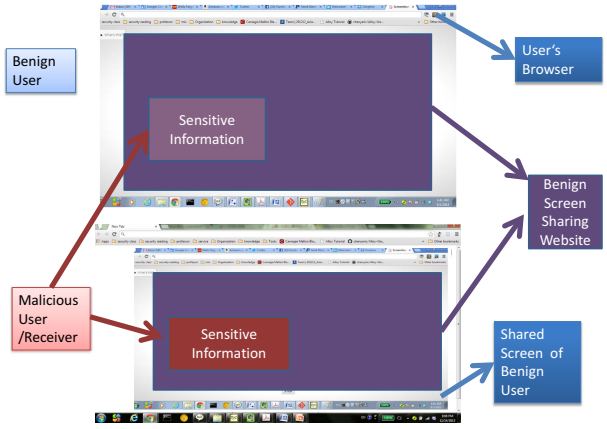
In-Scope Threats. In this paper, we are concerned with the security impact of the new screen sharing API. We discuss attacks which can be launched when a user is using a screen sharing website to share his screen with other users. The attacker could be the malicious screen sharing website or the other user with whom your screen is shared.

As illustrated in Figure 5(a) and Figure 5(b), we consider two threat models. In the first model, a malicious screen sharing website attempts to steal the user's sensitive information from other target sites and the browser and to affect the integrity of the user's account state. Then, the user is using their browser with accounts from target websites logged in, and screen sharing with other users from an attacker's screen sharing website is initiated. The malicious screen sharing website embeds pages of target websites in iframes or pop-ups and makes these pages invisible to the user. In the second threat model, the attacker is a malicious receiver. While they are sharing screens together by a screen sharing service, the malicious receiver sends some malicious links which embeds sensitive information to the victim. As the malicious receiver can use tricks to hide sensitive information and record the shared screen to extract content invisible to human eyes, the victim might not be aware of the attack. Even if the victim user finds something suspicious and quits the screen sharing session, his sensitive information has already been collected by the attacker. To provide a clear outline of the threat models, we identify six roles involved and define their abilities below.

- **Malicious website:** A malicious website is a web attacker who hosts a website with a valid SSL certificate and uses the screen sharing API to view the user's screen. The attacker lures the user to visit the site and convinces the user to share the screen. Once the user starts using screen sharing on the malicious website, the web attacker displays the user's sensitive information on the screen. Since the content on the user's screen is sent to the screen sharing server during the screen



(a) Malicious screen sharing website: The user is logged in on target websites inside his browser and his screen sharing is being shared with other users through the attacker's screen sharing website. The malicious screen sharing website can see the cross origin content rendered inside the users browser.



(b) Malicious screen sharing user/receiver: The malicious user/receiver tricks the benign user to click some malicious links during a screen sharing session. When the benign user clicks on the links, sensitive information is disclosed to the malicious user. Note that in this threat model, the screen sharing website is benign.

Fig. 5. Two threat models of attacks with the screen sharing API.

sharing session, the attacker can collect the user's sensitive information. Furthermore, the web attacker can utilize the security credentials he gets from the screen to launch more sophisticated attacks such as CSRF attacks with the CSRF token, which is collected from the user's screen.

- **Screen sharing website:** Screen sharing website is a website which provide screen sharing service. It captures the screen from one user and forwards it to another user.
- **Benign user:** A user visiting the malicious website authorizes screen sharing with the attacker. The website does not request the user to enter any sensitive data when sharing the screen. Also, the user does not open any browser tabs or windows that will reveal the user's personal data to the attacker. We assume that the user does not log out of target sites such as Gmail, Wells Fargo and Amazon. We argue that the assumption is reasonable because most popular sites do not invalidate the user's session until the user logs out. Users might believe that the attacker's site cannot access their data on other sites since the browser's same-origin policy prevents cross-origin data access. Therefore, the user may not feel it is necessary to log out of services before starting screen sharing. We also assume that the user is an ordinary human with common vision limitations such as not being able to catch content which is flashing fast or see content which is almost transparent. This means that if the attacker exploits these human vision limitations to hide the process of collecting sensitive information, the user cannot notice it. Even if the user indeed notices the attack, it is done almost instantly and their sensitive information is already stolen by the attacker.
- **Malicious User/Receiver:** Malicious user/receiver receives the shared screen from the victim user and tries to collect sensitive information from the victim user. They steal the sensitive information by tricking the user to click some malicious links during the screen sharing session. After the victim user clicks through the malicious links, the malicious user/receiver will be able to look at the sensitive content such as user's account information, browsing history and CSRF token from the shared screen.
- **User's Browser:** We assume that the user uses browsers that are patched for vulnerabilities allowing history sniffing and autocomplete history stealing reported in the past [11] [12] [13]. The web attacker cannot bypass the same-origin policy enforced by the browser, but can execute any JavaScript within the context of the attacker's domain, while still honoring the same-origin policy.
- **Target Sites:** Target sites contain sensitive information that an attacker wants to steal. Our threat model assumes that target sites defend against cross-site scripting (XSS) and CSRF attacks. These sites use secret validation tokens to protect against CSRF attacks and these tokens cannot be stolen by traditional web attackers without screen sharing because they cannot access cross-origin page sources.

Overall, the first threat model contains the following roles: the malicious screen sharing website, the benign user, the user's browser, and the target website. The malicious screen sharing website collects the user's information by using invisible iframe or pop-ups to steal user's sensitive information. In the second threat model, we consider the following parties: screen sharing website, user, malicious user/receiver, user's browser,

targeted websites. During a screen sharing session, and the malicious user/receiver tricks the benign user to click some pages with sensitive information. In this threat model, the attack works even when the screen sharing website is benign. Note that a malicious user/receiver and a malicious screen sharing website can also collude to collect information. The screen sharing attacks we proposed would work in HTML5-compliant browsers (including mobile web browsers) without installing additional software.

Out-of-Scope Threats. We do not consider the network attacker. We assume that network attackers cannot steal any sensitive information by acting as a man-in-the-middle (e.g., if all of the target sites operate over HTTPS).

IV. Attacks using Screen Sharing

A. Overview of Attacks

Screen sharing aims to provide a real-time communication channel that allows users to share the visual contents of their entire computer screen with other users. The visual channel opens up capabilities of the screen sharing website to collect information from different origins. The screen sharing website can utilize some visual effects to make the content seen by the user different from the real information displayed in the screen. For example, the website can flash sensitive information quickly or make an iframe look transparent. By combining screen sharing with other tricks, the malicious screen sharing website can collect sensitive information without users' consent. This idea inherently breaks the same-origin policy that restricts web documents from leaking information to other domains. Unfortunately, the current design of modern browsers does not take screen sharing into account, thereby introducing multiple security holes. In the following sub-sections, we describe how an attacker can exploit these security holes and compromise the user's security. We show that it is possible to attack several popular websites, such as Wells Fargo, Gmail and Bing, using the screen sharing API thereby affecting the integrity and confidentiality of the user's session. A brief summary is provided in Table I.

Steps of Attacks. Attacks listed in the table take two steps as are illustrated in Figure 6. In the first stage, a user starts to share his/her screen with the screen sharing website provided by the attacker. The sensitive information from the user will be collected, such as CSRF tokens and personal information. In the second stage, the attacker server will send attack packets back to their victim's browser based on the information collected in the first step. For example, the attacker server can send forms with stolen CSRF tokens back to the browser to launch the CSRF attack. For attacks on confidentiality, attackers only need to get information from the user, hence they just need to go through the first step. For attacks on integrity, attackers need to implement the second step to send packets from a user's browser to victim websites. Due to the simplicity of this process, the attacks can be easily automated and finished in a short period. Detailed descriptions about attack processes will be provided in the next section.

Security Principles	Vulnerabilities
Integrity	CSRF
Confidentiality	Autocomplete history sniffing User account history sniffing Personal information theft Browsing history sniffing

TABLE I. The screen sharing API disrupts the integrity and confidentiality principles.

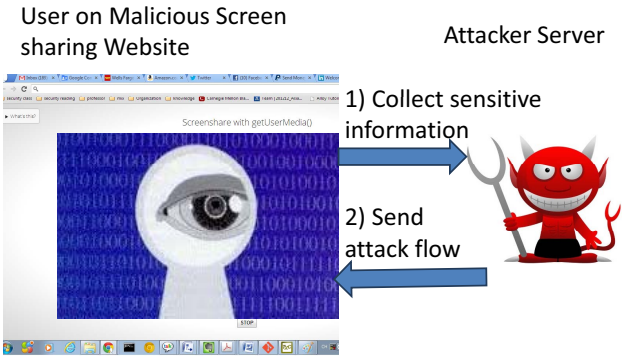


Fig. 6. The steps of launching attacks using Screen Sharing. 1. A victim user starts to share his/her screen with the screen sharing website provided by the attacker, and the attacker collects sensitive information from the user's screen. 2. The attacker sends packets from the user's browser based on the information collected in the first step.

B. Attacks on Integrity

Session integrity is essential for users and the websites they trust to exchange data. If the attackers can compromise session integrity, they can obtain unrestricted access to the target site and perform actions on behalf of the user. One such attack that affects the integrity of the user's session is the CSRF attack. In this attack, the attacker disrupts the integrity of the user's account state by forging a request with the user's credentials. A commonly adopted defense against CSRF attacks relies on the trusted site setting secret validation tokens that are only known by the user's browser and sent back with the request to authenticate the sender. However, the defense is vulnerable in a situation where the content of the target site is likely to be leaked to third parties [14]. Particularly, in the use case of screen sharing, the user's secret validation tokens are accessible to the attacker hosting attacker.com with malicious screen sharing services (see Figure 7). Once the user clicks on a button to authorize screen sharing, the attacker could inject code to embed an iframe with *view-source* links into the DOM. If the target site (say, bank.com) doesn't enable X-Frame-Options [15], the *view-source* link will expose the entire page source including the secret validation tokens to the attacker[2]. Once the attacker obtains the secret token, the attacker can send forged requests from the user's browser. The trusted site accepts this request since it contains the expected

[2]iframes can no longer render *view-source* links in Google Chrome and Mozilla Firefox [7], [16].

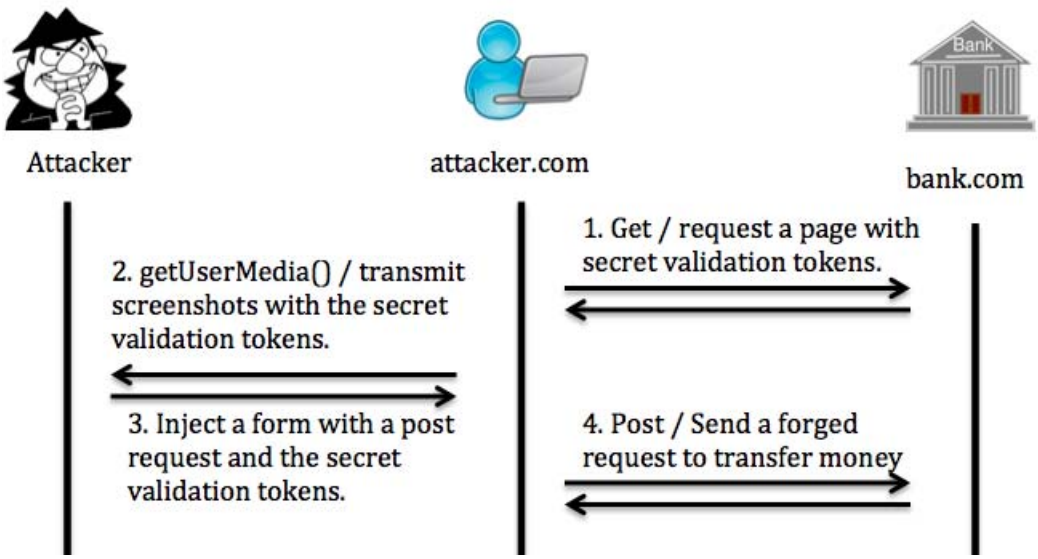


Fig. 7. Steps of a CSRF attack. 1. attacker.com requests a page with secret validation tokens from bank.com. 2. The screenshots with the source code and secret validation tokens are transmitted to the attacker's server via the screen sharing API. 3. The attacker sends a form with a post request and the secret validation tokens to attacker.com to transfer money. 4. The post request is sent from the user's browser to bank.com and accepted by bank.com.

authentication information: the HTTP cookie and the secret validation tokens.

Retrieving secret validation tokens from target sites is a crucial step in a CSRF attack. The following piece of code worked in our CSRF attack experiments. First, we used an iframe with the *view-source* link to expose the source code on the screen. The *view-source* link syntax is different between Google Chrome and Firefox. For Google Chrome, the code is implemented as follows:

```
<iframe viewsource="viewsource"
    src="https://bank.com" ... />
```

For Firefox, the code has slight changes:

```
<iframe src=
    "view-source:https://bank.com" ... />
```

Next, we exploit relative positioning of CSS to locate the lines containing the secret validation tokens inside the iframe. By using a negative value for top along with the position property, we are able to scroll the iframe to a specific position. For example,

```
<div
   style="position:absolute;top:-2000px">
   <iframe
      style="width:800px;height:10000px"
      viewsource="viewsource"
      src="http://bank.com" />
</div>
```

With this technique, we can collect the CSRF token or other security credentials from a size-limited window.

Using the code mentioned above, we test on multiple popular websites that adopt secret validation tokens to defend against CSRF. Below, we describe one case study of Wells Fargo where we are able to break its defense mechanism through screen sharing.

Wells Fargo. Wells Fargo uses session arguments extensively in their websites. One usage is passing a session argument as a URL parameter in the page to send money. The mechanism impedes an attacker's advances toward hacking; however, it will not work while the user is sharing the screen because the attacker can retrieve the URL from the source code. For example, in Figure 8 the source code of the "Transfer" page reveals the URL of "Send & Receive Money" which is supposed to be secured by the session argument. Since the Transfer page doesn't enable X-Frame-Options, the attacker is able to use iframes and *view-source* links mentioned above to extract the URL.

When the attacker successfully lands on the "Send & Receive Money" page, the attacker is able to perform a variety of severe and persistent CSRF attacks because this page contains multiple critical session arguments and URLs that allow the attack to send forged requests. For example, in Figure 9, the source code exposes the session arguments and URLs for requests to update recipients, add recipients, and manage contacts. The attack can expose critical information to manipulate the user's recipient and contact lists. Worse, attackers can transfer money to their own accounts because the page also provides a link to transfer money. In Figure 10, the attacker sends a post request to wellsfargo.com which contains the stolen session arguments and URLs to transfer money to his account. Note that these attacks no longer work on Google Chrome [7] and Mozilla Firefox [16] since these browsers do

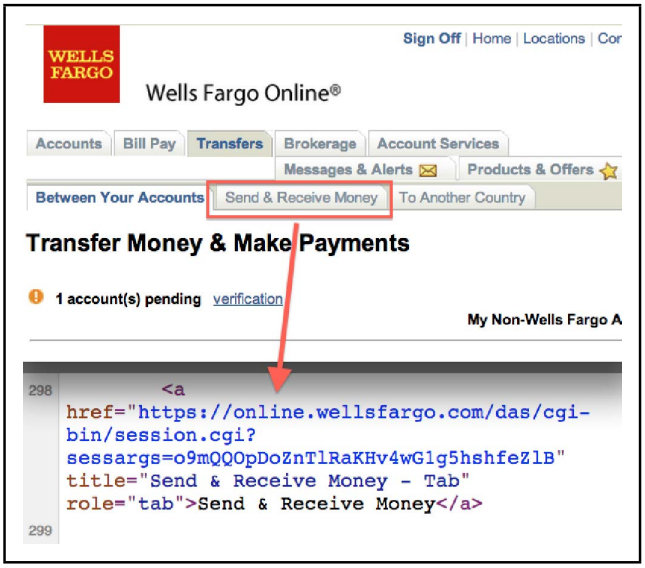


Fig. 8. Secret session arguments can be retrieved from the source code by the attacker during a screen sharing session, and these arguments can be applied to mount further attacks.

not display page source using *view-source* links inside iframes.

C. Attacks on Confidentiality

Confidentiality of web sites guards users' sensitive information, such as email addresses and credit card numbers, from being disclosed to third parties. To guarantee confidentiality, many websites adopt encrypted communication channels such as HTTPS to transmit data between servers and clients. However, with the screen sharing functionality, a malicious screen sharing website can collect cross-domain content from victim websites by displaying target websites on the screen. Therefore, HTTPS defense is entirely broken when the screen sharing API is involved.

In the following discussion, we illustrate four attack methods that can disrupt the confidentiality of trusted websites using the screen sharing API: auto-complete history sniffing, framing target websites, opening target websites in a new window and browsing history sniffing. For auto-complete history sniffing, attackers utilize the auto-complete feature of the browser to collect history with characters invisible to the user. For framing target websites, attackers navigate iframes to websites which contain sensitive information and do not have X-Frame-Options enabled. For websites which have X-Frame-Options turned on, attackers open them in new windows. For browsing history sniffing, attackers embed URLs to test if users have visited certain websites. Through these approaches, attackers collect information in three categories: personal information, account activity, browsing history. Personal information includes the user's account name, password, email, address, and bank account number. Account activities are the records of user behaviors for one account, including purchase history, search history, and transaction history. Browsing history is formed by

```
649 <form id="SendMoneyForm"
    class="fieldErrors sendMoney
    tokenTransfers disableOnSubmit"
    action="https://online.wellsfargo.com/das
    /cgi-bin/session.cgi?
    sessargs=aD9ZpPUL6yDBbuuX3BjXJHIY7l_1N2W0
    " method="post" autocomplete="off">
650
```

Fig. 10. The attacker can create a post request to send money with the stolen arguments during screen sharing.

the sites the user has visited before.
We summarize how the attackers applying the four methods to collect data from the three categories in Figure 11. In the next few sections, we display the results from these attacks.

1) Autocomplete History Sniffing: Autocomplete is a browser feature designed to save the user's effort from typing the same input repeatedly. User-supplied form values for text fields are shared across different websites by the "name" attribute. Take the Gmail login form for example. As shown in Figure 12(a), the username entered into the Gmail login page will automatically populate the autocomplete history for all subsequent input fields bearing the same name as that on the Gmail login page, which is "Email".

The browser shares the same autocomplete database across all websites. The only exception to this rule is that the browser does not populate the input field if the type is set to "password". More specifically, the browser provides the password history only to the page where the user had manually entered his/her password initially. This prevents malicious websites from stealing the saved passwords for other websites. Generally, the user enters the same information for fields such as email, phone numbers and address because this information that identifies the user does not depend upon the website.

Though the browser saves the autocomplete history, the browser prevents websites from directly accessing the auto-complete history. For example, there is no DOM element associated with the autocomplete history, which could be accessed programmatically using JavaScript. However, Jeremiah Grossman has discovered multiple vulnerabilities which can allow a malicious website to steal the autocomplete history by manipulating the autocomplete functionality [11]. Browsers have patched the vulnerabilities that allow stealing personal data [12].

Since the browsers have patched these issues, a malicious website cannot access the autocomplete history programmatically. Firefox resolved this issue by restricting the web pages to programmatically send input to the text fields [13]. Google Chrome 30, however, still allows passing inputs programmatically to the text fields using JavaScript. This vulnerability combined with the screen sharing feature enables the attacker to brute force the autocomplete history for text-fields with common "name" attributes such as name, email, address, and age.

To prove this, we use the proof-of-concept from ha.ckers.org,

```
var TokenTransfersProperties = {

        animationSpeed: "fast",
        updateRecipient: {
                controllerUrl: "https://online.wellsfargo.com/das/cgi-bin/session.cgi?
sessargs=eJVghtw8GDzqDNHi4Qlpfak6vCw4XW94",
                header: "Edit Recipient",
                width: "500",
                acknowledgeButton: "Save",
                rejectButton: "Cancel",
                returnFocus: "true",
                recipientUpdated: "The recipient has been updated.",
                recipientNotUpdated: "Recipient cannot be edited at this time. Please try again la
                sendByEmailLabel: "Send By Email Address",
                sendByMobileLabel: "Send By Mobile Number",
                sendByAccountLabel: "Send By Wells Fargo Account Number"
        }, addRecipient: {
                controllerUrl: "https://online.wellsfargo.com/das/cgi-bin/session.cgi?
sessargs=kl5fBRMEHo7ekNjvWnWxq38DlhE4NeRE",
                header: "Add Recipient",
                width: "520",
                acknowledgeButton: "Save",
```

Fig. 9. Multiple session arguments are exposed in the source code. Therefore, the screen sharing attacker can collect these arguments and use them to generate CSRF attack packets.

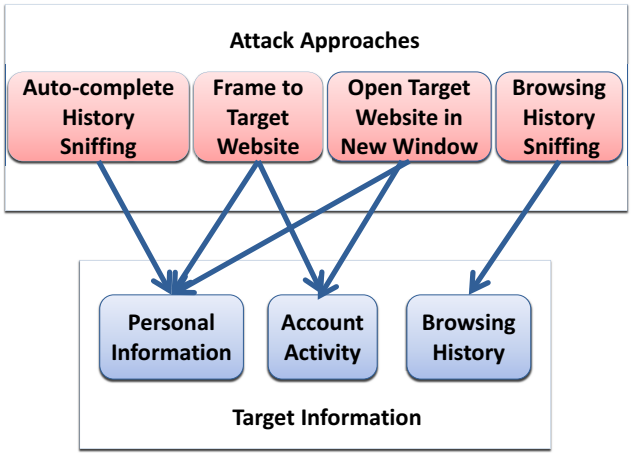


Fig. 11. The attacker can apply the four methods to collect data from the three categories.

which was used to demonstrate the autocomplete history stealing attack on Safari [17]. This JavaScript code programmatically supplies all possible characters as single character inputs to the input text fields for each of the common text field attribute names such as name, company, city, and email. The browser automatically populates the autocomplete history, if it exists, for the combination of the starting letter and the text field attribute name. The sample result is shown in Figure 12(b).

Using this technique, the attacker forces the browser to display the autocomplete history on the victim's own page.

2) Browsing History Sniffing: Browsing history refers to a list of web pages where a user has visited. Although the information is secured by the user's browser, the information can be stolen in the context of screen sharing.

One attacking approach is to embed target links in the attacker's website that provides screen sharing services. The attacker can infer the browsing history from the colors of links rendered on the screen. Although vendors hide the render differences between visited and unvisited links from JavaScript in order to prevent history sniffing, they still expose the differences to users for usability reasons; therefore, the screen sharing attacker can display the links in a way users cannot easily notice and capture the color of the links to infer the browsing history of the user.

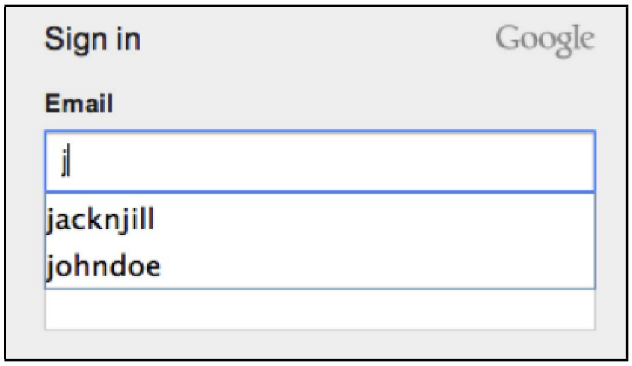
3) Framing Target Website: For websites that do not enable X-Frame-Options, attacks can still succeed if attackers frame pages that expose sensitive information. We demonstrate how the attackersteal personal information in the following sections.

Account Activity Sniffing. Many websites provide users with the option of seeing their account activity. For example, on an e-commerce site, the user can see his transaction history. Similarly, on a search engine site, the user can check his search history. The URL of the page that stores the user's activity information is often static. Thus, the attacker can attempt to open these URLs in a new window on the victim's browser, and if the victim is logged into these services, the pages will expose the user's account history.

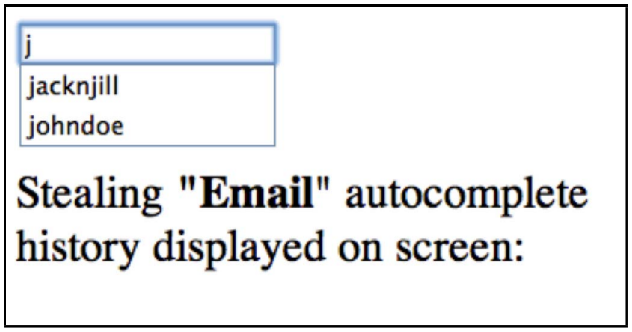
Below, we illustrate two popular services that record the account history at a static URL.

- Amazon:
  1) http://www.amazon.com/%20gp/history?ie=UTF8&ref_=ya_browsing_history
  2) https://www.amazon.com/gp/css/order-history/ref=ya_orders_css.

  These two pages store a user's browsing and purchase history. For example, Figure 13 shows the browsing history of an Amazon account. Since the two pages are



(a) The browser automatically shows the strings from autocomplete history that start with the letter(s) entered in the text field. For example, the figure shows what the browser displays on entering the letter "j".



(b) The attacker's page enters "j" in the input text field to see the autocomplete history related to the element that had the name attribute set to "Email".

Fig. 12. Auto-complete History Sniffing

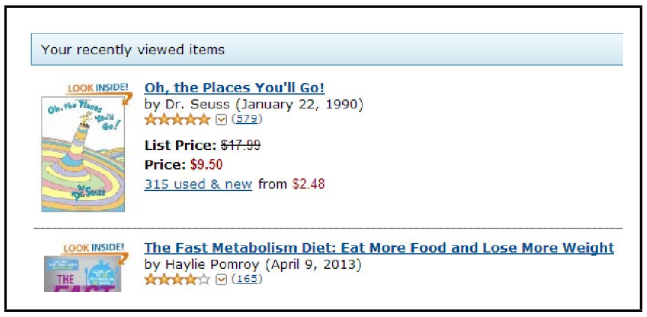


Fig. 13. The attacker can collect the browsing history of Amazon by displaying the history page.

not protected by X-Frame-Options, the attacker can load them inside an iframe on their website.

- Bing:

  http://www.bing.com/profile/history. Bing stores a user's search history locally, irrespective of whether the user is currently logged in or not. As illustrated in Figure 14, the Bing history page keeps the search history, organizing the terms searched by users in blue blocks with respect to time. By opening the URL, an attacker can retrieve the user's search history from the screen.

As for Google Search, users are asked to enter their passwords before they can see their search history. Since our threat model assumes that the user does not have to enter any sensitive information on the screen while his screen is being shared, Google Search history cannot be stolen using this technique.

Personal Information Theft. These attacks are similar to account history sniffing expect for the information at risk. The attacker can get sensitive information from a user if the user is logged in their account. The attacker can steal information stored at static URLs with the help of screen sharing by opening

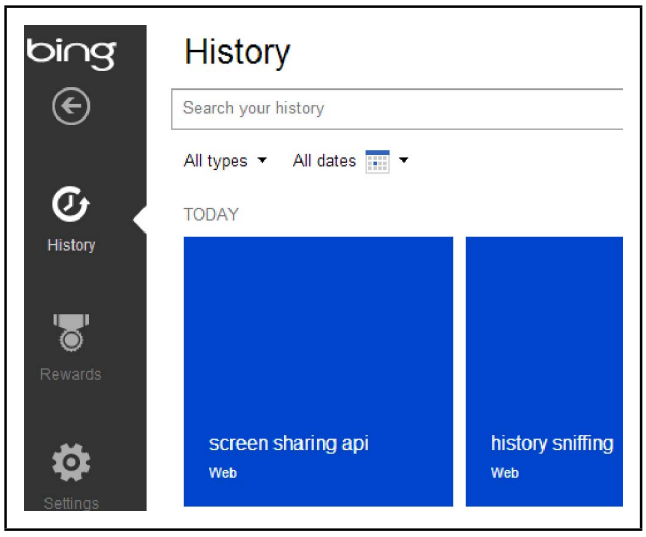


Fig. 14. The attacker can steal the search history of Bing during screen sharing.

such pages inside the user's browser. For example, some of the popular services that store sensitive account information at static URLs are shown below.

- ebay:
  http://my.ebay.com/ws/eBayISAPI.dll?MyeBay&CurrentPage=MyeBayPersonalInfo&gbh=1&ssPageName=STRK:ME:LNLK. This URL reveals sensitive user information such as the user's address email, ID and payment methods.

- Amazon:
  To purchase goods and deliver them to a user's address, the user inputs payment and address information into their Amazon account page. The attacker can embed the page that records payment information in the screen sharing page. Figure 15 is a screenshot for the information that would be collected by the attacker.

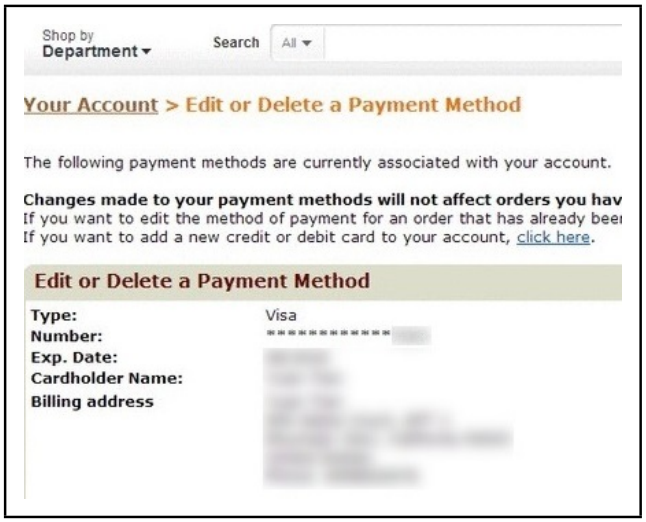


Fig. 15. The attacker can collect payment and address information from Amazon.

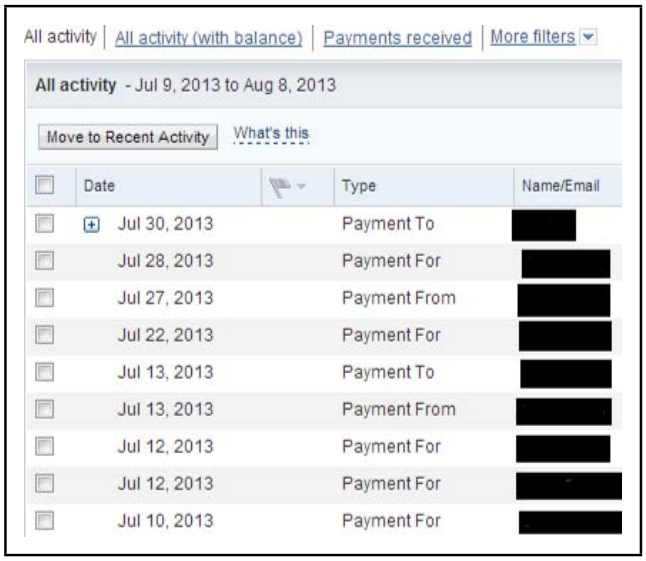


Fig. 16. The attacker displays activity history of Paypal.

4) Open Target Sites in New Browser Windows: Even if the websites are protected from being loaded inside an iframe, an attacker can still steal the secret information because the attacker can see the pages containing sensitive information by opening them in a new window. The attacker can make it imperceptible to the user by opening and closing the window really fast. Similarly, they can also collect personal and account activity information. We use the following examples to illustrate such attacks.

Account Activity Sniffing. Websites hosting sensitive account activity information are vulnerable when X-Frame-Options are not enabled. Here we chose PayPal as an example.

- Paypal:
  https://www.paypal.com/us/cgi-bin/webscr?cmd=_history&nav=0%2e3. Paypal keeps track of transaction history for users, including email address of senders and receivers, as well as other detailed information of the transactions (see Figure 16).

Personal Information Theft

We experimented with popular websites and found that attackers can steal information from many websites by opening the target websites in a new window. This is not an exhaustive list of static URLs that host sensitive content. The aim of this section is only to demonstrate that it is easy for an attacker to launch such attacks.

- Google:
  Google hosts a variety of services, such as Gmail, Google Contacts, and Google Calendar. Since these web pages contain plenty of sensitive information, an attacker can exploit pop-up windows to open these web pages. For example, the attacker can steal the user's Gmail and Google Doc by opening the following links during the screen sharing session:
  1) https://mail.google.com/mail/#inbox
  2) https://drive.google.com/?tab=mo&authuser=0#my-drive
- PayPal:
  PayPal maintains personal information, such as a user's SSN, email, phone number and address, in the profile page (https://www.paypal.com/webapps/customerprofile/summary.view?nav=0%2e6). The attacker can collect the user's information by opening the profile page using a popup window (see Figure 17).
- Facebook:
  For Facebook, the attacker can pop out a window and navigate to https://www.facebook.com/messages/ and https://www.facebook.com/friends?ft_ref=mni to steal Facebook messages and other sensitive information.

V. Analysis and Discussion

With the visual channel created by screen sharing API, the assumption that websites cannot access the cross-origin content directly is broken. Therefore, current defenses which are based on the assumption can be bypassed by the attacks using screen sharing API. In this section, we first present our analysis about the practical relevance of the attack, and then demonstrate how the attacks bypass the current defenses. Finally, we discuss some potential solutions of the problem for browser vendors, websites and users and compare their security and usability.

A. Feasibility of Screen Sharing Attacks

The screen sharing attacks are feasible and imperceptible to users owing to the following reasons. First, the screen sharing attack becomes more workable when it is combined with orthogonal attacks such as phishing attacks and social engineering. The user might be tricked to click on a malicious website for screen sharing. Even when the user is in the process

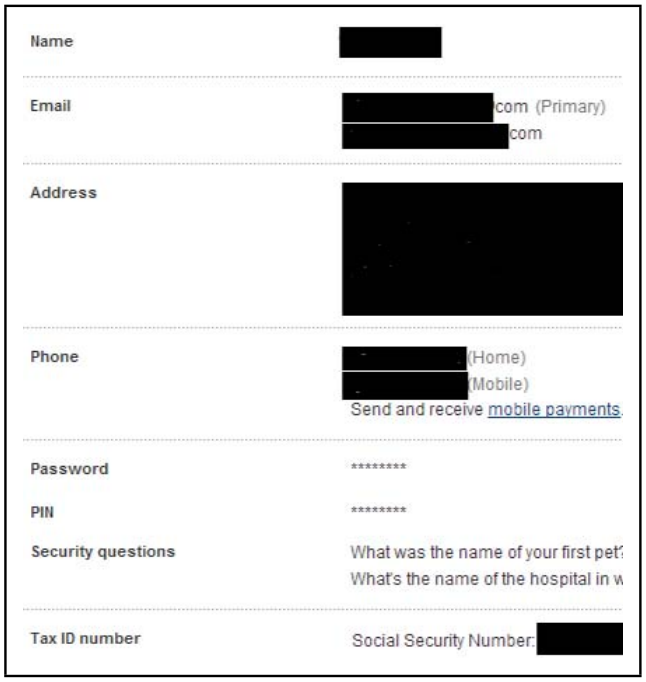


Fig. 17. The attacker can collect sensitive personal information from PayPal by displaying the account information page.

of screen sharing on a benign website, the screen sharing attack could also be fulfilled. For instance, if the user has a friend whose account has been compromised, the friend could invite the user into a screen sharing session and convince the user to click adversarial links. Generally speaking, once the screen sharing session starts, the attack becomes possible.

Second, the adversary is capable of playing some tricks to render the attack invisible to user due to the limitation of human vision. The user can hardly notice the flashing content and almost transparent content. Even if the user observes something suspicious and stops the screen sharing, the attacker has already captured the sensitive information and finished the attack. Finally, to make the attack even less observable, the attacker can combine the timing attacks to determine whether the user is logged into the target websites. The attacker may choose to only launch attacks when the user has active sessions with the target websites.

B. Deficiencies of Current Defenses

In Section IV, we demonstrate a variety of attacks that could compromise the integrity and confidentiality of trusted websites. Modern browsers such as Google Chrome and Firefox may be vulnerable to those attacks because the defenses they currently adopt do not consider the security implications of the screen sharing API. Those defenses make certain assumptions that are no longer valid if screen sharing is being used.

As listed in Table II, the assumptions of the CSRF, history sniffing, and SSL defenses are not valid in the context of screen sharing. The CSRF tokens, which is the most widely implemented CSRF defense, can be stolen by the combination of the *view-source* link and the screen sharing API. The defenses that block JavaScript queries against history sniffing can also be compromised because an attacker can see a user's history directly from the screen. SSL cannot guarantee the integrity and confidentiality of the user's session with a trusted website because the attacker is able to access the raw data that is already decrypted in the client side.

C. Discussion about Potential Defenses

As we discussed in the last section, current defenses deployed by browsers and websites are not designed against the screen sharing API and cannot stop all attacks. This demonstrates that new defenses need to be developed to secure the API. Therefore, we propose several potential defenses against screen sharing atttacks. These defenses can be classified into 3 categories : security enhancement for browser vendors, best practices for websites and best practices for users. We compare the usability and security of the potential defenses (see Table III) and analyze the details of these methodologies in the following sections. A privacy-preserving and more usable solution will require further investigation.

1) Security Enhancement in Browsers: To defend and mitigate against these attacks, browser vendors can enforce constraints over screen sharing. We discuss potential defenses below, and compare the pros and cons of these approaches.

Restrict loading View-source Links The view-source option is vulnerable because attackers can exploit it to get access to source code that contains sensitive information such as CSRF tokens. Browsers should prevent view-source links to be opened through JavaScript. The page that displays the page-source should be opened only when the user makes an explicit request through the browser UI and not programmatically using JavaScript. This lightweight approach can reduce the attack surface introduced by screen sharing. Chrome has removed the access to view-source pages in iframes via Javascript after this issue was reported. This issue is also patched in the latest Firefox version. However, these patches can not defend against viewing the cross-origin page in pop-ups.

Incognito Mode. One of the most intuitive ways to defend against screen sharing attacks is to enforce incognito mode when screen sharing is initiated. However, this approach itself cannot throughly assure privacy during screen sharing. Before incognito mode is launched, attackers can pop up windows containing sensitive information and hide them in the background. Once the screen sharing session starts, attackers can change the focus of the screen to display those hidden windows.

Defenses	False Assumptions	Attacks
CSRF defense using secret validation tokens	Secret validation tokens are secured by the browser	Attacker can use the *view-source* link to expose the tokens
Block JavaScript queries for CSS styles	History information is guarded by the browser	iframe and window objects can directly expose an user's history
SSL	Content transmitted via SSL is not readable to the attacker	Read the data after the browser decrypts it and renders on the screen.

TABLE II. DEFICIENCIES OF CURRENT DEFENSES FOR SCREEN SHARING ATTACK

Category	Approach	Usability	Security	Implementation Effort
For browser vendors	Restrict loading view-source links	High, no user interaction	Medium, can block CSRF attacks, but not the secrecy attacks	Minor, remove a feature
	Incognito mode	Medium, require basic user interaction	Medium, cannot block user from seeing pre-opened pages	Medium, add feaure to run incognito mode when sharing screen
	Fine-grained sharing	Low, need user to configure tediously	High, can block most attacks	Medium, add a feature
	Share one-tab at a time and constrain cross-origin content	Medium, require basic user interaction	High, can block most attacks	Medium, add a feature
For websites	Implement proper CSRF defenses	High, no user interaction needed	Medium, can not block secrecy attack	Medium, add a feature
	Implement X-Frame-Options headers	High, no user interaction needed	Medium, can not block pop-up attacks	Medium, add a feature
For users	Log out other websites	Low, need user to configure tediously	High, can block most attacks	None

TABLE III. COMPARISON OF POTENTIAL DEFENSES

Fine-grained Sharing. Setting up a fine-grained scope for screen sharing and/or asking users to specify the domains they want to share help to confine the capability of attackers. When users share content at the DOM element level, the attacker can only obtain the DOM elements the users choose to share. This presumes that users know which elements contain sensitive information and they are not tricked into sharing elements they do not want to share. However, because it is tedious for users to select the DOM elements every time, they might tend to select more elements than intended, or even all of the elements to share. The same usability issues might also arise when users are asked about what domains they want to share. Users might end up approving the screen sharing website to handle information from more origins than needed.

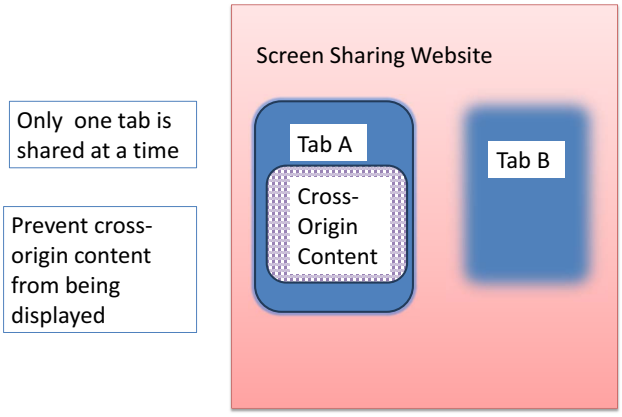


Fig. 18. Defense by sharing one-tab at a time and constraining cross-origin content

Share one-tab at a time and constrain cross-origin content. The idea is to only allow sharing content of one tab each time and set up constraints for cross-origin content, as is illustrated in Figure 18.

First, we can prevent attacks of collecting information from other tabs by enforcing users to only share one tab at a time. Every time the user wants to share another tab, he needs to choose the new tab and reinitiate screen sharing. The new tab

will be reloaded before it is shared to make sure no historic information is leaked to the sharing server. The sharing process will be disabled in the former tab and enabled in the new tab. By enforcing this approach, the attacks which exploit bugs to pop out windows or change the focus of screens will be blocked. Most of the screen sharing use cases involve a single tab and are not affected by this defense.

Secondly, we constrain the content in the iframe during screen sharing to prevent attackers from getting useful information, such as tokens or other credentials, by framing to other websites. There are two options to block the cross-origin information leakage in iframes. One option is to blank out the cross-origin content in the iframe. If the attacker tries to embed iframes to other origins, they will only get empty pages. However, it might bring along some usability issues to the user. Another is blocking third-party cookies. Even if attackers try to iframe to victim websites and the user has logged into their account in the victim website, the attackers can only get the login page. If they want to collect sensitive personal information or account information, they will need the user to log in from the iframe again. This approach will break some applications which depend on third-party cookies, such as advertisements and the Facebook Like button. Though the constraints on iframe content might reduce usability, they improve the security of the API and they are only enabled when users are sharing the screen.

Limiting screen sharing to only one tab at a time and using iframe cross-origin filters will reduce the attack surface and retain most of the screen sharing use cases. It is the most promising idea which strikes a balance between security and usability. Chrome starts to work on tab pickers for screen sharing, while blocking third party content is still under discussion.

2) Best Practices for Websites: Websites can defend against the basic attacks that exploit screen sharing by implementing controls such as X-Frame-Options and verifying the ORIGIN header, however they cannot defend against sophiscated screen sharing attacks such as stealing information by opening new windows. In the following sections, we discuss about these standard defenses which can be used to reduce the attack surface and their deficiencies.

Implement CSRF defenses with ORIGIN header. Websites can prevent CSRF attacks in which the attacker steals the secret token by viewing it in the source by verifying the ORIGIN header. So even if the attacker gets the token via the shared screen, he would not be able to complete the attack if the server verifies the ORIGIN. Note that ORIGIN header should not be used as an alternative to CSRF tokens but should be used as an additional defense, since the ORIGIN header is only supported in WebKit-based browsers, not Firefox or Internet Explorer..

Implement X-Frame-Options headers. Websites can prevent their sensitive pages to be loaded inside malicious websites, thus preventing the attacker to view-source inside an iframe. However, the attacker can still get around this defense by opening the target webpage in a new pop-up window.

Avoid using Double Submit Cookies. Instead of generating a new token, the double submit cookies technique [18] uses a session ID as a random token. The session ID present in the cookie is copied in the page source and is submitted as a CSRF token. It assumes that the attacker cannot extract the session ID, so this ID can act as a random CSRF token. However, with the assistance of screen sharing, the attacker can easily read the page source. In such cases, using the double submit cookies technique will not only break the CSRF defense but also result in a session hijacking attack.

3) Best Practices for Users: There are a few ways for users to mitigate screen sharing attacks, such as only using trusted screen sharing websites with trusted users or logging out of important accounts, although the latter may sacrifice the usability of screen sharing. For example, the users need to log out of their bank accounts before sharing the screen, and log in again after the screen sharing session is terminated. These repeated activities are usually not preferable. Therefore, this approach is not as effective as browser-based or website-based solutions because it heavily depends on user actions.

VI. Related Work

CSRF Attacks and Defenses. Basic CSRF attacks have been known to the community for several years. For these sites that are already vulnerable to basic CSRF attacks, the screen sharing API is not required to compromise the account. These sites are out of scope for our paper. Our CSRF attack focuses on the websites that implement the most popular CSRF defenses. Past studies show various techniques of stealing the CSRF tokens; however, these methods have limitations. E. Vela [19] demonstrates a heavy-load CSS-only attribute reader by using attribute-selectors. However, it is not practical to read CSRF tokens with high entropy within a short period. Heiderich et al. [20] propose another CSS attack by using features such as web-fonts based on SVG and WOFF, CSS-based animations and the CSS content property to extract CSRF tokens. This attack requires around 100 HTTP requests, whereas screen sharing attacks can steal CSRF tokens easily. In addition, their attack focuses on CSRF token protected links, so the attack will not work in the scenario where the CSRF token is not attached to the links. In contrast, the attack using the screen sharing API can extract any CSRF tokens, irrespective of whether the X-Frame-Options are set or not by the target page. A recent work on CSRF is implemented against Facebook [21]. The author proposes to frame the Facebook pages not protected by X-Frame-Options such as plugins, and then generate a captcha from the CSRF token. The attacker has to trick the user to input the captcha to get the CSRF token. The attack needs to interact with the user multiple times such as requesting user permission for the plugin and asking the user to input the CSRF token. Note that the screen sharing attack does not need such information.

There have been multiple proposals for CSRF defenses. SOMA [22] and App Isolation [23] provide CSRF defenses by defining valid entry points for the website. This can protect against the CSRF attacks using the screen sharing API, but

it is infeasible to whitelist every entry point. Moreover, the web relies heavily on interlinks, so these solutions were not widely adopted. Gazelle [24] and Tahoma [25] provide cookie isolation between different apps, which also protect them from CSRF attacks. However, the strict isolation has some usability issues. et al. [14] investigate current CSRF defense methods such as CSRF tokens, Referer header validation, and custom header, and also propose an approach of checking the origin of request. According to their study, the CSRF token, which is the most popular defense, is reliable if well implemented. However, we find that the CSRF token defense does not work during screen sharing because the attacker can read the CSRF token directly. Mao et al. [26] propose a defense by inferring if a request reflects a user's intentions. To judge the intentions, Mao et al. suggest checking Source-set of a request, which includes its referer and all web pages hosted in ancestor frames of the referer. However, referer information can be manipulated by the attacker and sending referer information also raises privacy concerns.

Frame Busting. Rydstedt et al. [15] propose best practices for writing frame busting code in JavaScript, which was used widely before the X-Frame-Options were adopted. A survey about frame busting techniques show that most of these techniques are not reliable and can be bypassed. We find that around 55 percent of the Alexa top 100 websites are using X-Frame-Options, but it is tricky to add the option to all the sensitive web pages within a domain. For screen sharing attacks, even if proper frame busting techniques and X-Frame-Options are used, we can still use pop-up windows to open target sites and steal sensitive information.

XSS. XSS is a common approach to stealing a user's sensitive information [27]. If malicious JavaScript is allowed to be executed within target sites, the attacker can access content from those sites. Many XSS defenses [28] [29] [30] are adopted to defend against XSS. However, the defenses are still vulnerable in the context of screen sharing. The defenses would not hinder the attacker from accessing the user's information because the attacker can directly see the user's information from the screen.

History Sniffing. History sniffing attacks discovered in the past were based on reading the difference between the rendered color of the visited and unvisited links using CSS and JavaScript [20] [31] [32] [33]. However, the vulnerabilities that allowed such attacks were fixed by browser vendors because of the prevalence of the attack vectors. David Baron from Mozilla [34] proposed a defense such that the true status of the link, whether it is visited or not, is never revealed to JavaScript. However, the links are still rendered with different colors on the screen. It was assumed that this differentiation is only visible to the user but the screen sharing API enables the attacker to capture the screen directly to observe the color of the links.

Screen-Capture Attack. To the best of our knowledge, previous work related to attacks that steal the user's screen are limited to only stealing credentials. For example, [35] talks about stealing the password while it is being entered through the virtual on-screen keyboard. However, their assumption for user is stronger than us. We assume that the user will not input any sensitive information while his screen is being shared, which is more reasonable.

Vulnerabilities of other HTML5 APIs Apart from the screen sharing API, there are other HTML5 APIs utilized to perform web attacks. One of such vulnerable APIs is the fullscreen API that allows developers to trigger the web page to be displayed in a full screen. By using the API, malicious web applications are able to launch phishing attacks with fake UI of target sites [36]. Another example is the postMessage API which aims to provide authenticity and confidentiality for cross-origin communication. The API is broken because developers fail to follow its complicated practices [37].

VII. Conclusion

The new screen sharing API enables developers to create rich web applications that can share media in real-time. However, the new API affects the fundamental browser security principle of the same-origin policy by creating a feedback loop from the user to the server. We have analyzed the security concerns raised this new screen sharing API. We discussed how it allows viewing cross-origin content and how attackers can exploit it. As a result, the integrity and confidentiality of user's information is at risk since attackers can manipulate the victim's session state with a trusted website, and view the victim's sensitive information. The browser vendors should analyze the possible impact on user's security before releasing this API in stable versions. Users need to be aware of the various security and privacy concerns raised by this new API so that they can protect themselves from leaking information to malicious screen sharing services. In summary, the capabilities of the new HTML5 screen sharing API have rendered existing security methods vulnerable. To counter those vulnerabilities, this paper provides a new paradigm for understanding screen sharing attacks. We envision that our study will encourage further research to find solutions for browser vendors, web developers, and users to defend against screening sharing attack.

Acknowledgments

We thank the Google Chrome team members, Adam Barth, Justin Schuh, Adrienne Porter Felt, Mustafa Emre Acer, James Weatherall, Victoria Kirst and Sergey Ulanov for their guidance on this topic. We also thank Sid Stamm and Daniel Veditz from Mozilla for their feedback. We also thank our colleagues and friends, Eric Y. Chen, David Liu, Emmanuel Owusu, Brian Ricks and Mike Xie for their suggestions.

References

[1] Google Inc, "Google voice and video chat," 2013, https://www.google.com/chat/video.

[2] WebRTC, "Web real-time communications working group," 2013, http://www.w3.org/2011/04/webrtc/.

[3] WebRTC, "Webrtc general overview," 2013, http://www.webrtc.org/reference/architecture.

[4] WebRTC, "Webrtc interop notes," 2013, http://www.webrtc.org/interop.

[5] V. Roth, K. Richter, and R. Freidinger, "A pin-entry method resilient against shoulder surfing," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 25–29.

[6] Google Chrome, "Chrome autofill forms," 2013, http://support.google.com/chrome/bin/answer.py?hl=en&answer=142893.

[7] Google Inc, "Stable channel update," 2013, http://googlechromereleases.blogspot.com/2013/07/stable-channel-update.html.

[8] E. Bidelman, "Chrome extension api with binary websocket," 2012, http://www.html5rocks.com/en/tutorials/streaming/screenshare/#toc-method3.

[9] E. Bidelman, "Screensharing a browser tab in html5?" 2012, http://www.html5rocks.com/en/tutorials/streaming/screenshare.

[10] E. Bidelman, "Capturing audio & video in html5," 2012, http://www.html5rocks.com/en/tutorials/getusermedia/intro/.

[11] J. Grossman, "Breaking browsers: Hacking auto-complete," 2010, http://jeremiahgrossman.blogspot.com/2010/08/breaking-browsers-hacking-auto-complete.html.

[12] A. M. Lits, "Apple-sa-2010-07-28-1 safari 5.0.1 and safari 4.1.1," 2010, http://lists.apple.com/archives/security-announce/2010//Jul/msg00001.html.

[13] Bugs@Mozilla, "Bug 527935 - (cve-2011-0067) untrusted events should not trigger autocomplete popup," 2009, https://bugzilla.mozilla.org/show_bug.cgi?id=527935.

[14] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[15] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: a study of clickjacking vulnerabilities on popular sites," in *Web 2.0 Security and Privacy*, 2010.

[16] Bugzilla@Mozilla, "Bug 624883- iframe with src="view-source..." should be treated as an unknown scheme," 2014, https://bugzilla.mozilla.org/show_bug.cgi?id=624883.

[17] J. Grossman, "Proof-of-concept: Safari autofill attack," 2010, http://hackers.org/weird/safari_autofill.html.

[18] OWASP, "HTTP csrf cheat sheet," 2013, https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet.

[19] E. Vela, "Css attribute reader proof of concept," 2011, http://eaea.sirdarckcat.net/cssar/v2/.

[20] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless attacks: stealing the pie without touching the sill," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 760–771.

[21] Crazyflx, "Own facebook exploit - silently post to the wall friends walls of visitors to your site," 2013, http://www.blackhatworld.com/blackhat-seo/facebook/597969-own-facebook-exploit-silently-post-wall-friends-walls-visitors-your-site.html.

[22] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji, "SOMA: mutual approval for included content in web pages," in *Proceedings of the 15th ACM conference on Computer and Communications Security*, 2008.

[23] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson, "App isolation: get the security of multiple browsers with just one," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, 2011, pp. 227–238.

[24] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter, "The multi-principal os construction of the gazelle web browser," in *USENIX Security Symposium*, 2009, pp. 417–432.

[25] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy, "A safety-oriented platform for web applications," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 15–pp.

[26] Z. Mao, N. Li, and I. Molloy, "Defeating cross-site request forgery attacks with browser-enforced authenticity protection," in *Financial Cryptography and Data Security*. Springer, 2009, pp. 238–255.

[27] M. Jakobsson and S. Stamm, "Invasive browser sniffing and countermeasures," in *Proceedings of the 15th international conference on World Wide Web*. ACM, 2006, pp. 523–532.

[28] P. Saxena, D. Molnar, and B. Livshits, "Scriptgard: Preventing script injection attacks in legacy web applications with automatic sanitization," Citeseer, Tech. Rep., 2010.

[29] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 921–930.

[30] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A systematic analysis of xss sanitization in web application frameworks," in *Computer Security–ESORICS 2011*. Springer, 2011, pp. 150–171.

[31] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell, "Protecting browser state from web privacy attacks," in *Proceedings of the 15th international conference on World Wide Web*. ACM, 2006, pp. 737–744.

[32] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in javascript web applications," in *Proceedings of the 17th ACM conference on Computer and Communications Security*. ACM, 2010, pp. 270–283.

[33] A. Janc and L. Olejnik, "Web browser history detection as a real-world privacy threat," in *Computer Security–ESORICS 2010*. Springer, 2010, pp. 215–231.

[34] L. D. Baron, "Preventing attacks on a user's history through css :visited selectors," 2010, http://dbaron.org/mozilla/visited-privacy/.

[35] A. Parekh, A. Pawar, P. Munot, and P. Mantri, "Secure authentication using anti-screenshot virtual keyboard," *International Journal of Computer Science Issues(IJCSI)*, vol. 8, no. 5, 2011.

[36] Feross Aboukhadijeh, "Using the html5 fullscreen api for phishing attacks," 2012, http://feross.org/html5-fullscreen-api-attack/.

[37] S. Hanna, R. Shin, D. Akhawe, A. Boehm, P. Saxena, and D. Song, "The emperor's new apis: On the (in) secure usage of new client-side primitives," in *Proceedings of the Web 2.0 Security and Privacy*. IEEE, 2010.