Timing is Everything — the Importance of History Detection

Gunnar Kreitz

KTH – Royal Institute of Technology
gkreitz@kth.se

Abstract. In this work, we present a *Flow Stealing* attack, where a victim's browser is redirected during a legitimate flow. One scenario is redirecting the victim's browser as it moves from a store to a payment provider. We discuss two attack vectors.
Firstly, browsers have long admitted an attack allowing a malicious web page to detect whether the browser has visited a target web site by using CSS to style visited links and read out the style applied to a link. For a long time, this CSS history detection attack was perceived as having small impact. Lately, highly efficient implementations of the attack have enabled malicious web sites to extract large amounts of information. Following this, browser developers have deployed measures to protect against the attack. Flow stealing demonstrates that the impact of history detection is greater than previously known.
Secondly, an attacker who can mount a man-in-the-middle attack against the victim's network traffic can also perform a flow stealing attack.
Noting that different browsers place different restrictions on cross-frame navigation through JavaScript window handles, we suggest a stricter policy based on pop-up blockers to prevent Flow Stealing attacks.

Keywords. Web Security, Flow Stealing, CSS History Detection

1 Introduction

In this paper we discuss an attack related to when a user's session is transfered between two different sites. One scenario in which such transfers occur is when a user moves from a store, `store.com`, to a payment provider, `pay.com`. We use this as a running example throughout the paper.

A typical integration mechanism is that the store sends information about the purchase to the payment provider (at least the total amount to be paid) and gets a transaction ID. The store then redirects the user to the payment provider with the transaction ID, either by a GET or POST request[1]. In this paper, we outline an attack where an attacker at this point redirects the user's browser to

[1] Several payment providers also provide lightweight integration where the store directly redirects the customer with information about the purchase instead of a transaction ID. This does not materially affect the attack, so we consider this equivalent to sending a transaction ID.

the same payment provider, but with a different transaction ID. The attacker could also choose to redirect the user to a malicious site stealing the user's credit card details, at the risk of such a redirect being more easily detectable by the victim. We refer to this class of hijacking the user's session as it transfers cross-domain as *Flow Stealing*.

The steps in a typical version of the attack are as follows:

1. Victim visits `evil.com`, and follows link to `store.com`
2. Victim interacts with `store.com`, eventually reaching checkout
3. `store.com` creates transaction on `pay.com`, which assigns transaction ID i_u
4. `store.com` redirects victim to `pay.com` with transaction ID i_u.
5. `evil.com` detects that victim hits `pay.com`
6. `evil.com` creates transaction on `pay.com`, which assigns transaction ID i_a
7. `evil.com` redirects the victim's tab to `pay.com` with transaction ID i_a

To the victim, the flow appears normal. She follows a link to `store.com` which opens in a new tab. The site is legitimate, so all interaction and security indicators such as certificates function as they would normally. When she goes to pay, she is transfered to the legitimate `pay.com` site, also with intact security indicators. The only indicator of the attack is in the payment information displayed by `pay.com`. What the difference is, how prominent it is, or if there even is one, depends on the information associated with the transaction that `pay.com` displays. This in turn often depends on the amount of detail about the purchase communicated from `store.com` to `pay.com` when it initializes the transaction.

Two questions arise: firstly, how does the attacker redirect the browser, and secondly, how does the attacker know when to redirect? We address these in Section 2 and Section 3. In one version, our attacker makes use of an old and well-known security hole, CSS history detection [1], in order to time her attack. To be able to redirect the victim's browser, the attacker needs JavaScript running in the browser and a window handle to the window which is to be redirected.

1.1 Attacker and Victim Model

We consider two forms of attackers: an attacker running a web page, and an attacker who can mount man-in-the-middle (MITM) attacks against the victim's network traffic. Our primary focus is on an attacker operating a web site, `evil.com`, visited by the victim. We assume that the attacker can convince victims to click on a link from `evil.com` to `store.com` and buy something. This means that our attacker could make some money (legally) by hosting advertisements or participating in an affiliate program. We remark that our attacker is *weaker* compared to the traditional attacker model in many CSRF and XSS attacks, as the attacker only needs the victim to follow a *legitimate* link to a well-known site.

We also consider a *network attacker* who can intercept and modify the victim's network traffic. There are several ways in which an attacker could get this ability. For an attacker on the same local network as the victim, the attacker can

utilize standard tools such as ARP or DHCP spoofing to get access to the victim's traffic. Alternatively, an attacker could set up a Tor exit node and thereby mount MITM attacks against anonymous victims. Given MITM access to the victim's network communication, all information sent and received over http can trivially be attacked, but our focus is on pages protected by https, a protocol intended to protect against network attacks. We do not assume that the network attacker can trick the victim into visiting her web site, so the network attacker is not strictly stronger than our normal attacker.

We consider a potential victim of our attack who follows the guidelines taught by the security community. She will only provide sensitive information over https, but not before verifying that the certificate is authentic. In addition to a security-conscious victim, we assume that the attacked flow is on domains served only over https.

1.2 Our Contribution

In this paper, we describe a new type of attack which we call flow stealing. Our attack makes new use of a well-known security issue in the CSS specification to time the execution of a redirection attack. By timing the redirection precisely, the attacker can give the victim a false sense of security by having her browse well-known sites before the attack is executed. This new use of an old attack emphasizes the importance of closing also minor security holes where the impact is not fully understood. Most major browsers have now closed the CSS history detection hole in their latest stable versions. However, flow stealing attacks can also be performed as a man-in-the-middle attack. Our flow stealing attack highlights a part of typical web flows which is difficult to protect using current mechanisms, namely legitimate cross-domain redirects.

We identify several scenarios in which the attack can be mounted, and we suggest new protection mechanisms which can be used to prevent flow stealing, as well as similar attacks. In particular, we point out the dangers of allowing JavaScript to navigate and close windows to which it holds a window handle and propose a new policy based on pop-up blocking.

1.3 Related Works

Flow stealing shares some similarities with cross-site request forgery (CSRF) and session fixation attacks. A related form of CSRF is the login CSRF attack, described by Barth *et al.* [2]. In a login CSRF attack, the attacker logs the victim on to a legitimate site using an account controlled by the attacker. The purpose of this is for the attacker to extract or use information stored by the victim's activity on the site. Examples of such abuse includes stealing the search history of the victim, or using stored credit card details to transfer money or make purchases.

As discussed in [2], the login CSRF attack is an example of vulnerabilities in session initialization. Another type of vulnerability in the same class is that of session fixation, where the attacker tricks the victim into logging in on a

legitimate site with a session ID known to the attacker. The attacker can then visit the legitimate site using the same session ID and then be logged in as the victim.

Also similar in spirit to flow stealing is the tabnabbing attack by Raskin [3]. In this attack, a malicious site detects when the victim is not looking at it and then replaces its content with a phishing site looking like a login or error message page at a legitimate site.

2 Redirecting the Victim's Tab

How can the attacker redirect the victim's browser? Firstly, this requires the attacker to get the victim's browser to run malicious JavaScript. This is easily accomplished for an attacker who convinces the victim to visit `evil.com`, as the page can contain the JavaScript required for the attack. A network attacker using a man-in-the-middle attack can insert malicious JavaScript into any page or script content served over unprotected http. For more details, see Section 2.2

Furthermore, the script needs to have a window handle to the tab in which the victim is visiting `store.com`, and later `pay.com`. If the victim opened the tab by clicking a link on `evil.com`, the attacker's JavaScript can store a window handle to the tab. We defer discussion of the man-in-the-middle case to Section 2.2.

Many browsers permit JavaScript to freely navigate any top-level window handles it holds. One notable exception is Opera which does not allow a window w_1 to navigate a window w_2 to which it has a handle if w_2 is currently browsed to a https page at a domain different from w_1. There is a simple way for our attack to get around this restriction in Opera, but it does make the attack easier to detect for the victim. We discuss the circumvention in Section 2.1.

We remark that once an attacker's JavaScript has a window handle to a window, it retains its rights over that window regardless of what happens. In particular, a user manually typing in a different address in the navigation bar does not revoke any of the opener's privileges.

2.1 Working Around Opera's Navigation Restrictions

Opera prevents a window from navigating another window via a window handle in some scenarios. In our flow stealing attack, we need to change the address of the victim's window when it goes to `pay.com`, which we assume is served over https. Thus, we propose a slightly different variation when attacking the Opera browser.

If a window w_1 wants to navigate the window w_2 to some address, it can accomplish a similar effect which may not be noticed if it closes window w_2 and navigates itself to the address it wanted window w_2 to go to. We are not aware of any browser placing restrictions on closing windows via a JavaScript handle. Depending on the victim's configuration and how many tabs she has open, this "navigation" may be more or less noticeable.

If the attacker can close window w_2, why not simply open another window with the right address in its place? The answer is that such an attempt will likely be prevented by a pop-up blocker. All mainstream browsers today prevent sites from arbitrarily opening new tabs, unless the action is initiated by a user action such as a mouse click.

2.2 Page Modification by a Network Attacker

In our attacker model, we consider a network attacker who is not assumed to be able to entice victims to visit her web site. Thus, the network attacker needs some other way to get JavaScript running in the victim's browser, as well as a window handle to a window where the victim then makes a purchase.

Most web browsing is still done over http, instead of https. However, we assume that both `store.com` and `pay.com` have invested in security and are served only over https. Thus, the network attacker cannot perform man-in-the-middle attacks against these domains directly.

Our network attacker can, however, easily modify any other page the victim visits over http. Thus, an attacker can write a proxy inserting malicious JavaScript into all pages the victim visits over http. To make this attack efficient, we assume that the attacker wants to adapt the JavaScript as little as possible to the page the attack is inserted into.

We begin with a discussion on what the JavaScript should do. We assume that the network attacker wants to avoid detection, and thus not modify any user-visible behavior of web sites. This means that she will want to insert JavaScript on the page such that it captures a window reference to any window opened by the page. A page can be opened for one of two reasons, either by the user clicking on a link with the `target` attribute set to "`_blank`", or by JavaScript on the page calling `window.open`.

Thus, the attack flow for our network attacker is as follows:

1. Victim visits `http://example.com`
2. Attacker's proxy inserts JavaScript into returned `example.com` page
3. Victim clicks on link to `example2.com`, opening in new window
4. Attacker's JavaScript captures a reference to the opened window

in which situation the network attacker is almost in the same position as when the victim visits `evil.com` and follows a link from there.

We start with links using the `target` attribute to open a new window. The attacker can insert JavaScript which executes when the page is loaded, and which loops through all anchor tags on the web page. When it reaches an anchor tag with `target` set to `_blank`, it modifies the tag to call a JavaScript function opening the window and storing the window handle when clicked. We remark that as these tags are easily detectable if the attacker parses the page, it would be easy to make this modification statically as part of a man-in-the middle attack as well. We present a simplified JavaScript example in Figure 1.

Handling windows opened by JavaScript on the original web site at first appears more difficult. To detect when windows may be opened could involve

dynamic analysis of JavaScript code. However, there is an easy way to capture references opened by JavaScript on the original page.

```
window.real_open = window.open;
window.open = function(URL, name, specs, replace) {
    var openedWindow = real_open.apply(this, arguments);
    storeReferenceAndStartTiming(openedWindow);
    return openedWindow;
}

function modifyLinks() {
    var links = document.getElementsByTagName("a");
    for (i=0; i<links.length; i++) {
        if(links[i].getAttribute("target") == "_blank") {
            links[i].setAttribute("onClick", "window.open(\"" +
links[i].getAttribute("href") + "\"); return false;");
        }
    }
}
window.onload = modifyLinks;
```

Fig. 1. Simplified JavaScript code to capture window references from non-malicious pages

To do this, we use a technique which has been used by Phung *et al.* [4] to construct a security mechanism for policy enforcement in JavaScript. The technique is based on the observation that even built-in functions can be aliased by user-defined functions in JavaScript. Thus, the malicious JavaScript can replace the `window.open` method with a JavaScript function which calls the original `window.open` method and stores a copy of the returned window handle before returning it to the caller. Slightly simplified JavaScript code illustrating the principle is shown in Figure 1.

3 Timing the attack

We now turn to the question of how the attacker can learn when the victim is redirected to `pay.com`. We present two mechanisms for accomplishing this. The first, and easiest method builds on the well-known CSS history detection attack to periodically poll whether the `pay.com` URL has become visited. The second method is based on traffic analysis by a network attacker.

3.1 CSS History Detection

An early feature in web browsers is the distinction between a visited and an unvisited link. With the advent of Cascading Style Sheets (CSS), the creator

of a web site gained the ability to decide how the two types of links would be rendered. It was soon realized [5] that this feature could be abused by a web site to determine of its visitor had also visited some other site. The CSS 2.1 specification [6, Section 5.11.2] notes the vulnerability and states that browsers may treat all links as unvisited or implement other counter-measures.

We remark that while an attacker can test if a visitor has visited a specific URL, she cannot extract the full browsing history of the visitor. In particular, she does not learn anything about URLs she cannot guess. The rate at which the attacker can test URLs is also an issue as it limits the privacy exposure of the attack. Here, the increasing prevalence of Web 2.0 applications and the accompanying optimization in general JavaScript performance has benefited an attacker. Speeds of 30000 tested URLs/second have been reported by Janc and Olejnik [1] with their optimized version of the attack.

Recall that the integration with a payment provider is typically done by setting up a transaction and then redirecting the user to the payment provider with a unique transaction ID assigned by the payment provider. The attacker is not able to predict the transaction ID, so if it had been a part of the URL, the attacker would not be able to use the CSS history detection attack to learn when the user visited the payment provider. However, common practice is to send the transaction ID to the payment provider as a POST parameter to a static URL, which allows our attack to work.

History detection attacks have been studied in the academic literature, and several demonstration web sites [7,8] have been created to raise awareness of the issue. Wondracek *et al.* [9] showed that stolen history data can also be used for a de-anonymization attack against users of social network sites. Jakobsson and Stamm [10] discussed the potential of using history detection in phishing attacks. Benevolent uses of the history detection attack have also been discussed. One example is to guess at which OpenID provider a user has to ease OpenID-logins [11], and another is to detect if a user has visited malicious sites and may have had malware installed [12].

The threat to user privacy is the most well-known implication of history detection. When coupled with fast testing, a non-trivial part of the user's visiting patterns can be extracted. This allows for testing of URLs containing location information such as zip codes entered on e.g., weather sites. In their real-world experiment Janc and Olejnik [1] noted that they could detect the US zip code for 9.2% of tested users.

3.2 Using History Detection to Learn When the Victim Reaches a Page

In our application of the history detection attack, we are not interested in the victim's browsing history but rather in what the victim is currently doing. In particular, we want to know when the victim's current browsing session reaches a target page (e.g., the landing page of a payment provider). To accomplish this, we can use the history detection attack to frequently poll the status of the target page to detect when it changes from unvisited to visited.

This use of history detection requires that the target page is marked as unvisited in the browser when the attack begins. Thus, the attack is easier to perform the quicker the browser forgets about visited links, in total contrast to privacy attacks which benefit from longer history retention. The CSS specification leaves it up to the implementor to select for how long a link will be treated as visited, and the major browsers have selected different periods. Internet Explorer and Safari stores history for 20 days, and Firefox for 90 days. Opera does not limit the time, but rather limits the number of stored entries to 1000. Chrome does not remove visited status, except when explicitly requested by the user.

Thus, our flow stealing attack is best suited to attacking pages which users trust, but which they visit rarely. We believe that payment providers fall in this category for many users.

3.3 Limitations of CSS History Detection

There are several ways in which the victim can be protected from the way we use CSS history detection in this attack. Firstly, Baron [13] has proposed a mechanism to close the CSS history detection security hole. The most basic mechanism involved is that the data returned by the JavaScript `getComputedStyle` method always return data as if the link had been unvisited. Furthermore, it prevents visited status of link from affecting which pictures are loaded, the layout of the page, and the time it takes to render a page to prevent a number of side-channel attacks. This proposal (or similar defenses) has been implemented in Firefox 4, Internet Explorer 9, as well as in browsers based on the WebKit rendering engine, such as Chrome and Safari. This means that in the latest versions mainstream browsers, with the exception of Opera, have closed the CSS history detection hole. Users may not always be able to upgrade to the latest version, for various reasons. For instance, Internet Explorer 9 is not supported on Windows XP, which will prevent many users from upgrading. Also, even if they could, some users simply refuse to upgrade their browsers. There is also a risk of regressions, or other history detection techniques being discovered. For instance, Weinberg *et al.* [14] reports that beta versions of Firefox 4 were vulnerable to CSS history detection through a debugging feature.

There are some techniques a user can deploy to protect herself, apart from switching or upgrading their browser. A user may choose to configure their browser not to store any browsing history. However, this comes at a usability price. Firefox users may also choose to install the SafeHistory extension [15] which essentially applies the same-origin policy to visited status on links, only treating a link as visited if it has been visited by a link from the current domain.

CSS history detection is not the only history detection attack that has been proposed against web browsers. In [16], Felten and Schneider discuss timing attacks to determine if cacheable elements of pages are present in the victim's cache. However, such attacks are not suitable to our history detection usage where we are not interested if the victim has historically visited a site, but rather in detecting the moment in time when a specific page is visited. Cache timing attacks cause the tested object to be cached, and thus the same object cannot

be tested twice, making the attack unsuitable for repeated polling. Similarly, the history detection attacks building on user interaction of [14] cannot be used in our scenario. We remark that there is a companion extension to SafeHistory called SafeCache [15] to protect against cache timing attacks.

3.4 Network Based Timing

In the case of a network attacker who has access to the victim's network traffic, there are alternative timing mechanism for the cases when the CSS history detection timing mechanism does not work. As we assume that all the victim's browsing of `store.com` and `pay.com` is via https, the attacker is unable to directly observe how the victim interacts with the target domains. However, https does not protect against an attacker learning *that* the victim is visiting a certain domain, or the sizes of requests and responses.

There are several ways for the network attacker to learn when the victim visits `pay.com`. The first is by simply observing the victim's DNS traffic. When the attacker sees the victim's computer performing a DNS lookup for the IP address of `pay.com`, she can assume that the victim's browser is going to request something from that domain. However, if the victim frequently visits `pay.com`, she may already have the IP address cached in her browser, and thus not issue a DNS lookup when visiting the domain again. Another mechanism for the attacker is to look up the IP addresses of servers for `pay.com` and then trigger the attack when she sees the victim's computer connecting to one of those IP addresses on the https port.

Both these mechanisms may trigger the attack too early if other pages include elements from the `pay.com` domain, for instance if `store.com` includes a `pay.com` logo on their payment page. While this type of logo inclusion does occur, we remark that it is common practice for stores to host payment logos on their own servers, or for static content such as logos to be hosted on separate domains.

The attacker can learn if the store features `pay.com` logos served directly by `pay.com` servers by simply visiting the store herself before beginning the attack. If this is the case, she can perform a more thorough flow inspection and instead of just looking for a connection establishment to the right IP and port, analyze the number of bytes sent in each direction and the number of connections made to distinguish between the victim fetching a logo and visiting the landing page at the payment provider.

Communicating Back to Victim's Browser When discussing the alternate timing mechanism available to the network attacker, we stated that the attacker "triggers the attack". However, the attacker is located as a man-in-the-middle to the victim's network traffic, and to trigger the attack, she must activate code running as JavaScript in a tab in the victim's browser. How is the trigger information communicated back to the victim's browser?

We remark that in our network attacker scenario, the malicious JavaScript has been inserted by the attacker on a web page not controlled by the attacker.

Thus, the malicious JavaScript is prevented by the same-origin policy from directly communicating with the attacker-controlled server at `evil.com` via convenient mechanisms such as XMLHttpRequest.

However, as the attacker is mounting a man-in-the-middle attack on the victim's network traffic, this problem can be circumvented by the attacker intercepting and responding to requests to some specific path, regardless of what host the path is supposed to be located at. This allows the JavaScript inserted by the attacker to use XMLHttpRequest to periodically send a request to a path which the attacker will intercept. The attacker will not forward such requests, but instead respond with a boolean value indicating if the flow stealing redirect should be activated. There are several other options available, such as periodically loading images from `evil.com` and using the size of the returned images as a one-way communication channel to the JavaScript running in the victim's browser.

4 Impact and Feasibility of Flow Stealing

We have now described our proposed flow stealing attack, showing how it can be performed both by an attacker operating a web site as well as by a network attacker who can intercept the victim's network traffic. Apart from the conditions imposed by the type of attacker, the feasibility of the attack also depends on the victim's browser.

4.1 Browser Features

Our flow stealing attack combines two different vulnerabilities. Firstly, the attacker must be able to monitor when the victim is directed to `pay.com`. The primary mechanism for accomplishing this is by using a well-known history detection hole. Secondly, the attacker must at that point in time redirect the victim to `pay.com` with a new transaction ID.

While the redirection part is crucial for the flow stealing attack, the CSS history detection vulnerability is not needed for network attackers, as discussed in Section 3.4.

All mainstream browsers allow the redirection part of our attack. However, on the Opera browser, the attacker cannot simply redirect the victim's tab, but must instead close the tab and redirect another tab as discussed Section 2.1. This makes the attack more noticeable, as an alert victim may notice that a tab closed and become suspicious and abort the transaction.

To explore the feasibility of our attack, we have tested recent versions of browsers to see if the classic CSS-based history detection attack works, and what restriction they place on cross-domain window navigation through window handles. We present our results in Table 1. In the table, "CSS History Detection" indicates if the CSS history detection attack works. Redirection indicates if a window handle can always be redirected via JavaScript ("Permissive") or not ("Restricted"). The browsers were tested on Windows 7. We do not believe any of the results depend on the operating system the browser is run on.

Table 1. Summary of browser's susceptibility to flow stealing.

Browser	CSS History Detection	Window Navigation
Firefox 3.6.15	Yes	Permissive
Firefox 4.0.1	No	Permissive
IE 8.0.7600.16385	Yes	Permissive
IE 9.0.8112.16421	No	Permissive
Chrome 10.0.648.151	No	Permissive
Safari 5.0.4	No	Permissive
Opera 11.11	Yes	Restricted

4.2 Experiences with a Proof-of-Concept

In addition to testing the individual pieces of our flow stealing attack, we have also developed a proof-of-concept implementation of the attack as performed by a web-site hosting attacker. We consider the simplest version of attack which can be performed with a static html containing JavaScript for the attack using the CSS history detection timing mechanism. In our proof-of-concept, we replaced `store.com` with the donation page of a charity, to simplify testing (the donation page of the charity contains a link directly to the payment provider).

In our proof-of-concept, the transaction set up by the attacker has the attacker as the recipient instead of the charity. The recipient information is displayed by the payment provider, so an alert victim could notice that their flow had been hijacked by an attacker. To reduce the risk of this, an attacker could register names with the payment provider which are similar, or look identical [17], to the stores or charities that she will attack.

Another option for stealthy attacks is for the attacker to herself set up a purchase on `store.com`. She then records the transaction ID used by `store.com` when referring her to `pay.com`. By using this transaction ID in the attack, the attacker tricks the victim into paying for the her goods. In this scenario, the only indication to the victim that an attack is ongoing is if the information displayed on `pay.com` on what the purchase concerns differs from what she expected.

Guessing the Price To make the attack convincing to the victim, the attacker needs to set up a transaction with the exact same cost that the victim expected. It seems likely that a large fraction of users would notice if the payment provider listed a different price compared to the store. We have not implemented any techniques for creating a transaction with the correct price in our proof-of-concept.

There are several ways for an attacker to guess the price. The easiest way is to attack subscription services or stores which sell a specific item or service for a fixed price, or a small number of different options so that the attacker can simply guess at the most common price. One such example is online streaming services such as Hulu, Napster, Netflix, and Spotify.

For stores with larger inventories, the attacker can use the CSS history detection attack to determine what items the victim has browsed and/or put in her shopping basket, depending on the URL scheme employed by the store. In

the network attack scenario, traffic analysis on the number of requests and size of responses as the victim browses `store.com` may be used instead.

5 Proposed Counter-Measures

In this section, we discuss a simple server-side defense against CSS history detection that can be applied by payment providers for their landing page. We also discuss the information displayed to users of payment sites. We proceed to discuss the problem of frame navigation as it applies to top-level frames and propose a new policy based on pop-up blocking. Finally, we discuss why traditional CSRF defenses do not protect against flow stealing.

We note that our attack uses JavaScript to perform the redirection attack, so users can protect themselves against flow stealing by disabling JavaScript. However, this does remove functionality from a large number of web sites, so most users are unlikely to do so.

5.1 Closing the CSS History Detection Hole

We are happy that almost all of the mainstream browsers now have closed the CSS history detection hole. By closing this hole, attackers are denied the easiest route for performing flow stealing attacks. However, for various reasons, users are not always able to upgrade to the newest version of software in a timely manner. To protect users which are not able to upgrade, we propose that high-profile sites such as payment providers should consider implementing a server-side defense.

While landing pages of payment providers are external URLs in the nomenclature of [10], they could apply a protection technique by recommending sites linking to them to insert a random number in the link, which is simply ignored by the payment provider. As most payment providers want to help stores to very easily integrate payments, standard practice seems to be to provide some static HTML code to be included on the store's web site. Such code could include JavaScript code to generate a random number in the browser which is inserted into the URL of the landing page in a way that is ignored by the payment provider. This would prevent the link from being guessable, and thus detectable via CSS history detection.

5.2 Payment Provider Pages

The key place where the victim could detect that a flow stealing attack was ongoing is in the information shown by legitimate payment providers. It is important to provide as clear feedback as possible to end users of payment sites on who the recipient of the payment is, and what the payment concerns. For instance, the payment provider could indicate if the recipient is a company, a charity, or an individual.

In a typical payment provider integration, the information on the purchase depends on what information is sent from the store to the payment provider

when setting up the transaction. Thus, stores can assist in making flow stealing attacks easier to detect by including more information. For instance, this may include the purchaser's username on the store, or the shipping address.

5.3 Limiting Window Manipulation via Window Handles

There is a difference in policy between browsers on what limits are applied to how a page can change the URL of another window to which it has a JavaScript window handle. Opera restricts such navigation based on the current location of the frame, and protects frames navigated to https sites from being navigated from another window. In Chrome, Firefox, Internet Explorer, and Safari, the opener is allowed to freely navigate an opened window, and in some of them, also other windows apart from the opener.

Frame navigation has previously been showed as being dangerously permissive in the context of embedded frames and iframes by Barth *et al.* [18], which influenced browser developers to implement a more restrictive policy. They note that top-level frames are often exempt from the browser's frame navigation policy, and that top-level frames are less vulnerable as their URL is shown in the location bar.

While it is true that top-level frames are less vulnerable than embedded frames, there is still a danger in permissive policies for navigation of top-level frames. We cannot trust a user to, at every point in time in their browsing session, validate that the location in the location bar is correct. For instance, we cannot expect users to note if their location is changed to a similarly looking URL, or identical looking URL via a homograph attack [17]. Neither can we expect users to notice if opaque identifiers in sessions are replaced.

The fact that different policies have been implemented in different browsers indicates that it is unlikely that a large number of pages rely on the most permissive policies for their functionality. The only policy restricting our flow stealing attack is Opera's. However, as we discussed in Section 2.1, Opera's policy is still sufficiently permissive that it allows flow stealing attacks by closing the window and redirecting the window running the attacking JavaScript. Thus, we argue that a replacement policy should not only restrict navigation, but rather all actions affecting the window, including closing it and resizing it (an attacker could emulate closing by resizing to a very small size).

We are not aware of any important applications where a window w_1 needs to modify another window w_2 where the modification is not prompted by user interaction with window w_1. For what types of user interaction would a user expect w_1 to modify the state of another window w_2? We argue that in any user interaction that would not allow w_1 to open a new window, w_1 should not be allowed to modify the state of another window either. In mainstream browsers today, the situations in which w_1 is allowed to open a window is restricted by a *pop-up blocker*. We believe a user would not expect w_1 to modify any windows unrelated to it, a policy already implemented in the Firefox browser which limits navigation to the *opener* window.

As far as we know, each mainstream browser implements its own algorithm for pop-up blocking, a feature enabled by default. Thus, most web sites have been adapted to page manipulations allowed by the pop-up blocking policies of browsers. We are not aware of any detailed descriptions of pop-up blocking algorithms, but they appear to work satisfactorily in major browsers. According to Chen [19], browser developers are hesitant to specify the exact policies used as that may prevent them from modifying the policy later, if a loophole is discovered.

Thus, we propose the following policy for controlling a window via a window handle:

Policy 1 (Window navigation, Proposed) *A window w_1 can modify (e.g., navigate, close, or resize) another window w_2 only if it is the opener of w_2, and the pop-up blocker policy currently allows w_1 to open a window.*

Furthermore, we believe that rights to a window should be relinquished entirely if the user manually navigates (e.g., by entering a new URL in the address bar) the tab. Currently, this does not appear to affect the rights granted to the JavaScript holding the handle to the window, but we believe it would match the user's expectation more closely that the opener retains no special privileges if the user navigates the window.

5.4 Traditional CSRF Defenses do not Prevent Flow Stealing

Our flow stealing attack has some similarities to traditional CSRF attacks, so one may wonder if traditional CSRF defenses protect against flow stealing as well. Sadly, the answer is no, and new techniques are needed to protect against flow stealing. We briefly mention the two major CSRF defenses from the literature and discuss why they do not protect against flow stealing. At the core of the problem is that in flow stealing the victim's browser is redirected at a point in time where the control is passed between sites operated by two different entities. Thus, the hijacked request is legitimately a cross-site request.

The most common class of CSRF defense consists of a secret validation token that must be sent along with all state-modifying requests, and that is matched to the user's session. There are several different implementations of this technique, and there are some subtleties in implementing the protection correctly, cf. [2]. Such tokens are designed to protect flows internally on web-sites, and are not immediately applicable to cross-site flows.

A second technique is based on inspecting either the `Referer` or the `Origin` HTTP header. Typically, this is described as only allowing requests if the host in the header matches the current host, but the policy could easily be extended to allowing external requests from some specific set of domains. As an example, a payment provider may require that users making payments to `store.com` come to `pay.com` with an Origin header set to `store.com`. However, this does not prevent flow stealing, as the attacker can register as a merchant with the payment provider and redirect via the correct domain for that merchant. The attacker can

also redirect the victim to a fake payment site instead of the legitimate site, thus bypassing any controls that could be implemented by a payment provider.

6 Conclusion and Future Work

In conclusion, we have demonstrated an attack on current web browser implementations. The attack uses the CSS history detection attack, which has been publicly documented for about a decade, to time a redirection attack. By redirecting the tab the victim is using at a point where the victim legitimately expects to perform some security critical action, the victim can be tricked into doing something more sensitive than what can be achieved by e.g. phishing. We hope that our attack further aids in demonstrating the importance of closing the CSS history detection hole, and future holes with similar impact.

As future work, we propose developing a proof-of-concept version of the network attack as well. The purpose of such a proof-of-concept prototype would be to show that while closing the CSS history detection hole is an important step, it is also important to further limit JavaScript cross-site frame navigation, as well as deploying https as a default for a larger fraction of Internet sites. We note that other proof-of-concept attacks such as Firesheep [20] have been able to quickly raise public awareness of security issues and caused deployment improvements at large sites.

Acknowledgments

I would like to thank Emil Hesslow for great discussions, development of a proof-of-concept, and his JavaScript expertise.

References

1. Janc, A., Olejnik, L.: Web browser history detection as a real-world privacy threat. In Gritzalis, D., Preneel, B., Theoharidou, M., eds.: ESORICS. Volume 6345 of Lecture Notes in Computer Science., Springer (2010) 215–231
2. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. In Ning, P., Syverson, P.F., Jha, S., eds.: ACM Conference on Computer and Communications Security, ACM (2008) 75–88
3. Raskin, A.: Tabnabbing: A new type of phishing attack `http://www.azarask.in/blog/post/a-new-type-of-phishing-attack/`.
4. Phung, P.H., Sands, D., Chudnov, A.: Lightweight self-protecting javascript. In Li, W., Susilo, W., Tupakula, U.K., Safavi-Naini, R., Varadharajan, V., eds.: ASIACCS, ACM (2009) 47–60
5. Ruderman, J.: Bug 57351 - css on a:visited can load an image and/or reveal if visitor been to a site `https://bugzilla.mozilla.org/show_bug.cgi?id=57351`.
6. W3C: Cascading style sheets level 2 revision 1 (CSS 2.1) specification `http://www.w3.org/TR/CSS2/`.
7. Anonymous: Did you watch porn (2010) `http://www.didyouwatchporn.com/`.

8. Janc, A., Olejnik, L.: What the internet knows about you (2010) http://www.wtikay.com/.
9. Wondracek, G., Holz, T., Kirda, E., Kruegel, C.: A practical attack to de-anonymize social network users. In: IEEE Symposium on Security and Privacy, IEEE Computer Society (2010) 223–238
10. Jakobsson, M., Stamm, S.: Invasive browser sniffing and countermeasures. In Carr, L., Roure, D.D., Iyengar, A., Goble, C.A., Dahlin, M., eds.: WWW, ACM (2006) 523–532
11. Kennedy, N.: Sniff browser history for improved user experience (2008) http://www.niallkennedy.com/blog/2008/02/browser-history-sniff.html.
12. Jakobsson, M., Juels, A., Ratkiewicz, J.: Remote harm-diagnostics http://www.ravenwhite.com/files/rhd.pdf.
13. Baron, L.D.: Preventing attacks on a user's history through CSS :visited selectors http://dbaron.org/mozilla/visited-privacy.
14. Weinberg, Z., Chen, E.Y., Jayaraman, P.R., Jackson, C.: I still know what you visited last summer. In: IEEE Symposium on Security and Privacy. (2011)
15. Jackson, C., Barth, A., Bortz, A., Shao, W., Boneh, D.: Protecting browsers from DNS rebinding attacks. In Ning, P., di Vimercati, S.D.C., Syverson, P.F., eds.: ACM Conference on Computer and Communications Security, ACM (2007) 421–431
16. Felten, E.W., Schneider, M.A.: Timing attacks on web privacy. In: ACM Conference on Computer and Communications Security. (2000) 25–32
17. Holgers, T., Watson, D.E., Gribble, S.D.: Cutting through the confusion: A measurement study of homograph attacks. In: USENIX Annual Technical Conference, General Track, USENIX (2006) 261–266
18. Barth, A., Jackson, C., Mitchell, J.C.: Securing frame communication in browsers. Commun. ACM **52**(6) (2009) 83–91
19. Chen, R.: The internet explorer pop-up blocker follows guidelines, not rules http://blogs.msdn.com/b/oldnewthing/archive/2007/08/31/4656351.aspx.
20. Butler, E.: Firesheep http://codebutler.com/firesheep.