What Private Browsing Leaves Behind

John Filleau, Milda Zizyte

Electrical and Computer Engineering, Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
jfilleau@cmu.edu
milda@cmu.edu

Abstract— Private browsing mode in modern browsers is expected to behave as a standard browsing session would, but store no data. Add-ons and extensions complicate this by taking the use of an add-on in private browsing mode to be user permission to store some data. We discover what local data modern browsers store, focusing on add-on behavior, by replicating and building upon previous work in the area. Using various forensic tools, we find that Chrome and Firefox add-ons sometimes store private browsing data if add-on preferences are changed. Based on the behaviors we observed, we give some suggestions to add-on developers to achieve stronger privacy and inform the users of add-on behavior in private browsing mode.

I. INTRODUCTION AND MOTIVATION

Private browsing mode is a feature on web browsers that allows a user to hide their browsing activities from other users of the same machine. It also may be referred to as “inPrivate browsing” (Microsoft Internet Explorer) or “incognito” (Chrome web browser). Users can expect private browsing mode to behave just as a standard browsing session would, with no data (such as log-in credentials or browsing history) saved upon exit. While browser developers attempt to keep users’ browsing habits private, there can sometimes be a mismatch between what the user expects and what the browser provides.

Namely, if a user’s browser advertises a private experience, they may fill in the blanks with their own subjective ideas of privacy [1]. Chrome, in particular, is fairly good at warning about the risks - it displays a splash screen on each new incognito tab that lays out the functions and limits of this mode.

In this work, we duplicate prior work in understanding the privacy of private browsing mode. We lay out some failed duplication of static analysis to identify privacy violations in browser extensions. Then, we duplicate prior work in a forensic (after the fact) analysis of persistent memory. Finally, we focus on some popular third party browser add-ons for both Chrome and Firefox, to see what they get right about privacy and where they may improve.

We find that some of these add-ons do a very good job of removing obsolete records of history, and all require at least some level of user interaction before any information is saved. From this analysis, we identify best practices that add-on developers can use to maintain user privacy.

II. THREAT MODEL

For our work, we define our threat model as an attacker who has physical access to a user’s computer and log-in information immediately after the user completes a private browsing session [2].

We define a privacy violation as the persistence of any information in physical memory that the user would not want revealed about their private browsing session.

If the user explicitly authorizes the browser or add-on to to save any information about their private browsing session - e.g., saving a bookmark or downloading a file - then we do not consider this information private. We will discuss later how this consent process is fuzzy, and higher intensities of consent may be required for higher sensitivity data.

III. RELATED WORK

We explored two main works in the length of this project.

A. Strobe: Javascript Static Analysis to Identify Privacy Violations

Lerner, et al. [3] extended the power of static type-checking to identify potential privacy violations in browser add-ons, by detecting writes to persistent memory as type errors. Add-on code is
written in HTML, JavaScript, and CSS, and the add-on is forced to interact through the browser’s API. They were able to take advantage of this and identify a handful of API calls that were able to write to persistent memory. Then, using static analysis, they were able to identify if the add-on was ever able to make these calls while in private browsing mode.

Any potential violations could then be manually inspected. In practice, they were able to find some actual violations in then-available add-ons using this method.

B. Forensic Analysis of Private Browsing

Satvat, et al. [2] used forensic methods to identify artefacts of private browsing after the private browsing session was closed. They surveyed the private browsing literature to find attacks on private browsing mode that leaked browsing or user data, and attempted to replicate and build on the results. They used freely available tools (discussed in Section V.) to inspect the paged memory, browser history, and browser SQL databases. They found persistence of private browsing information in the memory and instances of the browser logging private browsing URLs. Even if those URLs were not explicitly displayed through the normal user interface, the data remained and could be easily accessed using a database viewer.

We mostly extend our work from this work, using the same forensic methods, but focusing primarily on how add-ons violate this privacy.

IV. STATIC ANALYSIS WITH STROBE

Our extension of Strobe\(^1\) was meant to bring up to date several-year-old research code and analyze current add-ons with modern APIs. The code was written primarily in OCaml, with JavaScript input components.

Our first step was to compile the program and run it on the basic JavaScript examples provided with the tool. Through several e-mails with the authors, we were able to get the tool mostly working on example code. The code uses .idl and .js files as inputs to provide the lexical rules for code parsing.

While we were able to modify the original files to work with the code as is, we were unable to get all examples working in time for deliverables.

Using static analysis for such privacy verification is a very novel idea, and we haven’t been able to find any recent work on the subject (for browser add-ons, at least). By eliminating large swathes of code that are irrelevant to private mode, we would be able to reduce the lines that require manual review, which helps decrease the number of unfound violations.

One helpful github.com user who had submitted an issue ticket with the Strobe project responded to our emails with suggestions for other typechecking tools. Readers who are interested in pursuing this subject further are directed to TAJS\(^2\), TeJaS\(^3\), Infernu\(^4\), RefScript\(^5\), or Flow\(^6\).

V. FORENSIC ANALYSIS WITH FREELY AVAILABLE SOFTWARE

Our extension of Satvat’s work with forensic analysis yielded much more fruitful results than our static analysis. Our attack model is a subset of the one presented in their paper (we omit the network and downstream server attacks); we only consider an adversary with physical machine access after a user has closed their private browsing session [2].

A. Tools

We used the same tools listed by Satvat, et al. These are an open source SQL database reader, DB Browser\(^7\); a memory viewer/editor, WinHex\(^8\); and since our experiments were run on Windows, Windows Process Monitor\(^9\), which proved very useful to find the explicit locations where add-ons were writing files.

Some of the searching was carried out using grep in CygWin, and viewing of text files and database logs was done in Notepad++.

\(^{1}\) [https://github.com/brownplt/strobe](https://github.com/brownplt/strobe)

\(^{2}\) [https://github.com/cs-au-dk/TAJS](https://github.com/cs-au-dk/TAJS)

\(^{3}\) [https://github.com/brownplt/tejas](https://github.com/brownplt/tejas)

\(^{4}\) [https://github.com/sinlaw/infernu](https://github.com/sinlaw/infernu)

\(^{5}\) [https://github.com/UCSD-PL/refscript](https://github.com/UCSD-PL/refscript)

\(^{6}\) [https://github.com/facebook/flow](https://github.com/facebook/flow)

\(^{7}\) [http://sqlitebrowser.org/](http://sqlitebrowser.org/)

\(^{8}\) [https://www.x-ways.net/winhex/](https://www.x-ways.net/winhex/)

B. Duplication of Prior Work

All experiments are conducted on a virtual machine running Windows 7 with all forensic tools installed. Each experiment starts with a fresh snapshot of the VM. We open a new private browsing window, log into GMail, log into Twitter and search for “lady gaga”, and log into Amazon and search for “sporks”. We then close the private session and employ our battery of forensic tools. If testing an add-on, we enable it in private mode, then do the above as “passive” mode, i.e., we do not interact with the add-on at all. Then we repeat these actions, but instead interact with the add-on in its basic functionality. We then close the private session and reemploy our forensic techniques.

We first duplicated the methods presented in the prior work. Satvat, et al. showed that after closing a private window, data from that session persists in paged memory. Using WinHex, we duplicated this attack, and found that Chrome and Firefox still persist the data, which included the URL visited and html code cached from the website. A screenshot of this is shown in Figure 1.

We opened a private browsing window in Chrome, and searched on Twitter for the phrase “lady gaga”. We then closed the browser and inspect the paged memory. This attack was duplicated on Firefox.

![Fig 1. Screenshot of WinHex inspecting Chrome’s primary memory after searching for “Lady Gaga” in a Twitter private session.](image)

This attack, however, is not dependent on any add-ons being installed, and is simply a consequence of how computers page and cache memory. A more relaxed attack model, such as the attacker only having access to a machine after it has been power cycled, could resist this attack just fine.

Second, we attempted to duplicate the timing attack described in the paper. This attack notes the timestamps of the profile files for the browsers.

![Fig 2. Hidden history record of bookmarking Twitter in private browsing mode. Record persists even after removing bookmark.](image)

Previous work showed that one of the files did not change in size, but had an updated timestamp if private browsing mode was used, which would tell an attacked that the user had used private browsing mode at the time. The authors stated that this vulnerability had been fixed in Firefox, so we tested Chrome, only. We checked the timestamp of every file in the Chrome profile directory before and after opening and using a private browsing window, and observed no change. This indicates that, sometime since 2014, the vulnerability was fixed in Chrome as well.

Next, we duplicated the persistence of private URLs in browser history databases. In Firefox, we found that visiting a page in private mode logs a record of that page in the local database. This record is flagged with a “hidden” bit, indicating that it shall not be shown when viewing history, but we are able to view this information using DB Browser.

In contrast, Chrome saves no record of privately visited websites. The exception is when a user bookmarks a site in private mode. The bookmark is saved in a separate “bookmark” database, but a record is also created in the “history” database. Like Firefox, this record is marked as hidden, so it’s not shown in the standard browser history view. However, it’s still accessible using these most basic forensic methods. Most surprisingly, after we removed the bookmark through the Chrome UI, the hidden history record remained in the database. This is shown in Figure 2, after visiting twitter.com.

C. New Work for Add-ons

We extend this work by inspecting the forensic traces left from private browsing mode after installing and using different add-ons. The add-ons we target are described collectively as providing security and privacy. We want to see if they achieve their privacy goals by sacrificing others.

We selected Ghostery, AdBlock Plus, uMatrix, and Anonymox as our target add-ons. They were in
the “most downloaded” add-on lists for the privacy categories of both Chrome and Firefox add-ons, and exhibit a wide array of behaviors. For each trial, we loaded a fresh snapshot of our virtual machine, installed the add-on from the Chrome or Firefox marketplace, enabled in private mode (for Chrome), and executed our method described in section V-B. For Chrome, the all of the privacy violations committed by add-ons were found in the local storage log.10

Ghostery- Ghostery monitors web trackers and allows users to toggle settings to block, allow, or engage in default behavior across different trackers and websites. Ghostery did not violate privacy in passive mode. In active mode, where we modify settings, Ghostery saved a record of the visited domain, in plaintext, in the log. When the site-specific settings were then set back to default, the record of the website persisted in the local storage.

AdBlock Plus- AdBlock Plus disables all types of ads on websites. Like Ghostery, it has site-specific settings. In passive mode, AdBlock did not violate any privacy. In active mode, when settings were changed, a record of that website was present in the local storage. Unlike Ghostery, however, when a website’s settings were set back to default, the record was permanently deleted from storage.

uMatrix- uMatrix is similar to Ghostery in function, in that it allows site-specific and tracker-specific blocking. Also like Ghostery, it had no passive violations and the same active violations as described above. Further, when resetting a website’s settings to default, the record is not removed.

Anonymox- Anonymox is a privacy add-on that operates on a user’s connection. When enabled, it routes all web traffic through one of Anonymox’ proxy servers to anonymize a user’s identity and IP location from the server end. The potential for privacy violations server side is certainly questionable, but as far as client side privacy is concerned, Anonymox stores no private information (in fact - no information at all) related to a user. Not even user settings, so aside from being installed, no record exists of when a user was using the add-on.

VI. DISCUSSION

A. Ambiguity in configuration behavior

We observed different ways that add-ons in private browsing handled configuration data. Some did not save any configuration data at all, some saved the data and cleared it when private browsing mode was exited, and some saved the configuration data and used the same configuration between private and standard browsing modes. All of these are potentially valid behaviors for an add-on and are up to the add-on developer to decide. However, the user rarely knows which model any given add-on is using. For example, if a user expects everything done in private browsing to be temporary, they may be surprised to see that some of their add-ons save their configuration from private browsing and use it in standard browsing. Hence, we recommend that add-on developers state which behavior they implemented in the installation description.

B. Best Practices for Add-on Developers

Based on our observations, we suggest several best privacy practices that add-on developers should keep in mind when developing:

1) Private and non-private session data should be separate, or at least the data stored in private mode should not be accessible in non-private mode. For example, an add-on that saves bookmarks will show only those saved in non-private mode in non-private mode. When in private mode, the add-on may display bookmarks saved in both private and non-private modes.

2) Data that users may want to keep private should not be stored in plain text. At the very least, reversible obfuscation can be used, to detract a less enthusiastic attacker. If possible, encryption should be used. This may be infeasible for add-ons where the user must constantly enter a password, where the annoyance of doing so outweighs a reasonable fear of privacy disclosure.

3) Before saving any information from a private browsing session, the add-on should prompt the

10 ~\AppData\Local\Google\Chrome\User Data\Default\Local Extension Settings\<addon>\000003.log
user that doing so may violate their privacy, and give them the option to opt out. This is the property of explicit consent. We feel that equating a user’s zeal to enable an add-on in private mode with informed consent is not a best practice, and should be avoided.

C. Custom Add-on

We developed a custom Chrome extension to highlight these best practices called “Best Privacies” (a portmanteau of privacy and practices). The extension can be installed by downloading the zip file[1] > extract the folder > chrome://extensions developer mode > load unpacked extension.

The extension has very little actual functionality; it allows you to save the hostnames of sites you’ve visited. The hostnames are saved in local storage, and any hostnames you save in private browsing sessions will be observable on your hard drive. Best Privacies has three best practice options that allow you to switch between storage mode (saves the hostname in either plaintext or an obfuscated base64), requires a user confirmation when trying to save hostnames in private mode, and whether or not to fully delete records of hostnames after they are removed from the hostname list.

The local storage can be viewed by opening the log file as described in footnote 10.

VII. Conclusion

Users rely on private browsing mode to perform tasks that they do not want saved on their local machine. However, users also demand a wide array of functionality from their browsers, and actions such as downloading a file locally should not be forbidden in private browsing mode. Hence, there exists a tradeoff between privacy and utility in browsers, and this is highlighted further in browser add-ons, which may save local data even in private browsing mode.

We explored several ways add-ons might save this data. We first described a static checking tool that would allow developers to audit their add-ons for unexpected writes. We also described an array of forensic tools and techniques to detect writes to memory and disk from private browsing mode. We replicated these techniques and showed how data can persist after a private browsing session ends.

We then expanded this work to browser add-ons, and described the various ways these add-ons function in private browsing mode. We saw that some add-ons store configuration data that leaks browsing history, even when the configuration is reset. We thus recommend that developers are more mindful of what data they store when writing add-ons, use obfuscation techniques, and, most importantly, are transparent with the user about what they are storing. Users can opt-in to have data locally stored on their machine, but, in private browsing mode, need to know that it is happening.

VIII. Distribution of Work

John contacted the developers and users of strobe to get and understand the code base. He also worked through OCAML examples in order to understand how strobe worked and helped compile and run examples in strobe.

In the forensic portion of the project, John replicated attacks in Chrome and performed experiments on the Chrome extensions. He also wrote a best practices extension to demonstrate the conclusions we reached.

Milda got the strobe tool running on some basic examples in the given directory. She updated one of the most important strobe configuration files (with many of the javascript definitions) to match the API in the rest of the code.

She also led the lit review for the forensic portion of the project, including initial work in replicating the attacks in the Satvat paper in both Chrome and Firefox. She replicated attacks in Firefox and performed experiments on the Firefox extensions.

References


---

[1] https://users.ece.cmu.edu/~jfilleau/bestprivacies