Browser History Stealing with Captive Wi-Fi Portals

Adrian Dabrowski*, Georg Merzdovnik*, Nikolaus Kommenda† and Edgar Weippl*
*SBA Research, Vienna, Austria
Email: {adabrowski|gmerzdovnik|eweippl}@sba-research.org
†Technische Universität Wien, Vienna, Austria
Email: nikolaus.kommenda@alumni.tuwien.ac.at

Abstract—**In this paper we show that HSTS headers and long-term cookies (like those used for user tracking) are so prevailing that they allow a malicious Wi-Fi operator to gain significant knowledge about the past browsing history of users. We demonstrate how to combine both into a history stealing attack by including specially crafted references into a captive portal or by injecting them into legitimate HTTP traffic.**

Captive portals are used on many Wi-Fi Internet hotspots to display the user a message, like a login page or an acceptable use policy before they are connected to the Internet. They are typically found in public places such as airports, train stations, or restaurants. Such systems have been known to be troublesome for many reasons. In this paper we show how a malicious operator can not only gain knowledge about the current Internet session, but also about the user's past. By invisibly placing vast amounts of specially crafted references into these portal pages, we can lure the browser into revealing a user's browsing history by either reading stored persistent (long-term) cookies or evaluating responses for previously set HSTS headers. An occurrence of a persistent cookie, as well as a direct call to the pages' HTTPS site is a reliable sign of the user having visited this site earlier. Thus, this technique allows for a site-based history stealing, similar to the famous link-color history attacks [1]. For the Alexa Top 1,000 sites, between 82% and 92% of sites are affected as they use persistent cookies over HTTP. For the Alexa Top 200,000 we determined the number of vulnerable sites between 59% and 86%.

We extended our implementation of this attack by other privacy-invading attacks that enrich the collected data with additional personal information.

I. Introduction

Browser history stealing discloses information about user's past browsing behavior without their knowledge, e.g., by visiting a website that mounts such an attack. The history is directly or indirectly extracted from the browser itself. One of the first and widely known methods facilitated the `:visited` attribute of links. Visited link targets can be rendered differently by the browser to ease navigation. Variants of this attack examine the displayed color [1] or load external images [2], [3] to determine if the user visited a specific URL earlier. Jang et al. [4] showed that several sites use these techniques to spy on their users.

Browser manufacturers reacted in two ways: First, they fixed a number of vulnerabilities and included mitigations, and secondly they introduced a *privacy mode* that exempts specific browsing sessions and associated data from appearing in the browsing history.

Captive portals are a technique to redirect the user on her first request to a portal website. These are heavily used in public Wi-Fi hotspot systems such as cafés, restaurants, airports, and hotels. The user is informed about the sponsor of the access, possible restrictions as well as potential payment methods and has to accept terms and conditions. After completion, the user is given access to the Internet. Especially in transit areas the majority of users use hand-held devices such as tablets and smart phones. Additionally, these devices automatically connect to known Wi-Fi access points. We specifically elaborate on these cases and their implications.

Public hotspots have been known to be troublesome in many regards: They are often unencrypted, fake access points can easily attract legitimate users, and they offer a single point through which all traffic has to pass. This gives the Wi-Fi or captive portal operator great power and insight on the users' current sessions.

In this paper, we describe how such an operator can also gain knowledge about a user's past. The attack can be carried out by the legitimate hotspot operator, but also by an evil-twin network pretending to be the real network [5], or generally by any man-in-the-middle attacker. They can trick the browser to disclose information that allows conclusions about the user's past browsing history. This also applies to VPN users, as they have to go through the login process before starting the VPN session and the browser keeps only one history regardless of the connection type.

The paper is structured as follows: In Section II and III we expand on the technical background of history stealing and why it is such a lucrative attack. Section IV describes the attack in detail whereas Section V expands on the differences with regards to mobile operating systems. We estimate the impact and applicability of the attack in Section VI by crawling Alexa's Top 200,000 sites. After describing our proof-of-concept implementation (Section VII) and its extensions and limitations (Section VIII), we present our conclusion in Section IX.

II. Motivation

A browsing history is a comprehensive picture not only about a user's past activities, but also about their interests, political opinion, sexual preference, geographical or ethnic heritage, spoken languages, social contacts and so forth. For example, a user who visits localized websites is most likely from that region or speaks its distinct language (e.g. *amazon.fr*, *amazon.jp*). Visitors of *grindr.com* or *transblog.de* most likely have a very specific sexual orientation, whereas *okcupid.com* or *parship.com* visitors are probably currently single. Likewise, history entries of websites for certain medical conditions (e.g. pregnancy, AIDS, depression), political campaign websites, or those of religious communities do not require much imagination to draw conclusions about a particular user. A reference to *intranet.somecompany.com* tells much about the employer.

In the past, URL-based history attacks have also been used to uncover social network contacts and de-anonymize users [6]. This information is of great interest for targeted advertisement [4], but also for targeted attacks.

Multiple history stealing methods have been presented in the literature. They range from analyzing the link color (visited links can be displayed differently) via Javascript [1] or conditionally loading external resources based on CSS rules [2] to GPU timing attacks [7].

III. Background

In this section we briefly describe the technical foundations of this work.

Captive Portals are a technique to display a certain content when a new user connects to a (wireless) network. There are multiple ways to achieve that goal. The most common is to intercept the first HTTP request from the user's browser to any site and redirect it (spoofing *HTTP 302 Moved Temporarily* response) to the web page of the operator's choice. This web site (the portal) will explain the user the terms and conditions for the usage of the network and its Internet connectivity. Sometimes this includes payment, other times just advertisement, or acceptance of a legal disclaimer. After completion, the user's physical MAC address or local IP address is put on a whitelist and its traffic is passed unaltered to and from the Internet.

HTTP cookies [8] are a technique (to be precise: an optional header) within HTTP requests to add a common state between the browser and the server to the otherwise stateless HTTP protocol. On a technical level, cookies are small pieces of data (directly or indirectly) set by the server that the client's browser attaches every time it sends a request to a specific site. Cookies are often used to either directly store user configuration (e.g., language choice), indirectly store a state (e.g., a session identifier with the actual data stored on the server), authenticate a user (e.g., a login cookie), or uniquely identify a client (e.g., tracking cookie).

There are multiple ways how cookies can be set in a browser. In the traditional way, the server sends a `Set-Cookie` HTTP response header to the client, giving the cookie a name and a value. Additionally, cookies can be set using Javascript on the client side. Many tracking libraries (e.g., Google Analytics) use this method.

By default, a cookie is only valid for the lifetime of a browser session. By setting an additional expiry date, the cookie becomes a *persistent* or *permanent cookie* which is able to survive multiple browser sessions. An *httpOnly cookie* can be set and read by the server, but not through Javascript. This is an option introduced against cookie stealing attacks via cross

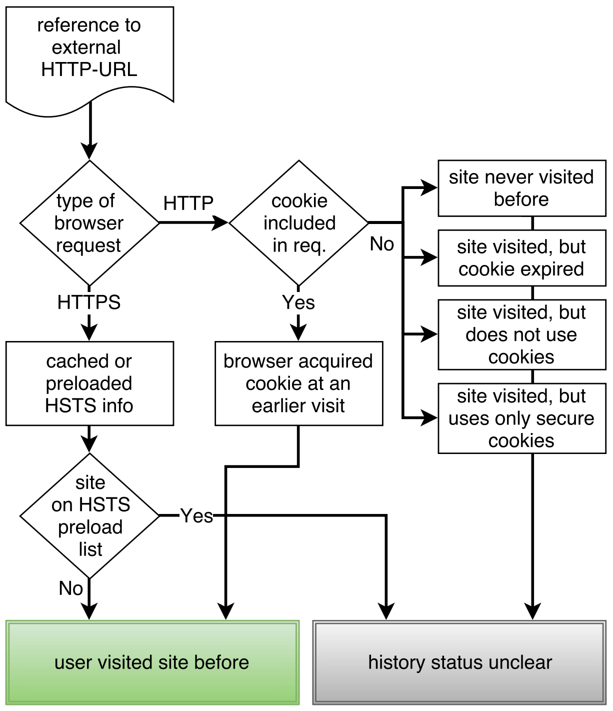
reference to external HTTP-URL
type of browser request
HTTP
HTTPS
cookie included in req.
No
Yes
site never visited before
site visited, but cookie expired
site visited, but does not use cookies
site visited, but uses only secure cookies
cached or preloaded HSTS info
browser acquired cookie at an earlier visit
site on HSTS preload list
Yes
No
user visited site before
history status unclear

Fig. 1. History stealing flow chart on users browser reaction.

site scripting (XSS). A *secure cookie* is only presented by the client to the server on encrypted connections (i.e., HTTPS). Cookies can also be bound to a specific domain and path where it will be also used for subordinate domain names and paths (e.g., a cookie set for the domain *example.com* is also used by the browser for *docs.example.com*).

IV. History Stealing in Wi-Fi Captive Portals

Wi-Fi hotspots are a popular method to access the Internet in public places. They are typically faster than mobile data connections and do not count towards a monthly mobile data plan. For foreign visitors, they are often the only way to save on excessive data roaming fees.

Most users are familiar with captive portals. At the first request, the web browser is redirected to a portal page that in many cases contains the terms and conditions, a word from the sponsor, and – if applicable – payment options.

A. Stealing History

Just like most other browser-based history stealing attacks, this attack scheme requires a list of URLs or domains of interest, and testing each of them for their occurrence in the browser history. The history stealing method works by inserting image references such as `<img href="http://somedomain.com/nonexisting-file?customtag">` for each site into the captive portal's landing page. The client will look up the domain and try to fetch the alleged external resource. Each of these requests is (again) intercepted by the captive portal. The chosen filename and/or an added tag ensures that the portal can identify these requests, record the

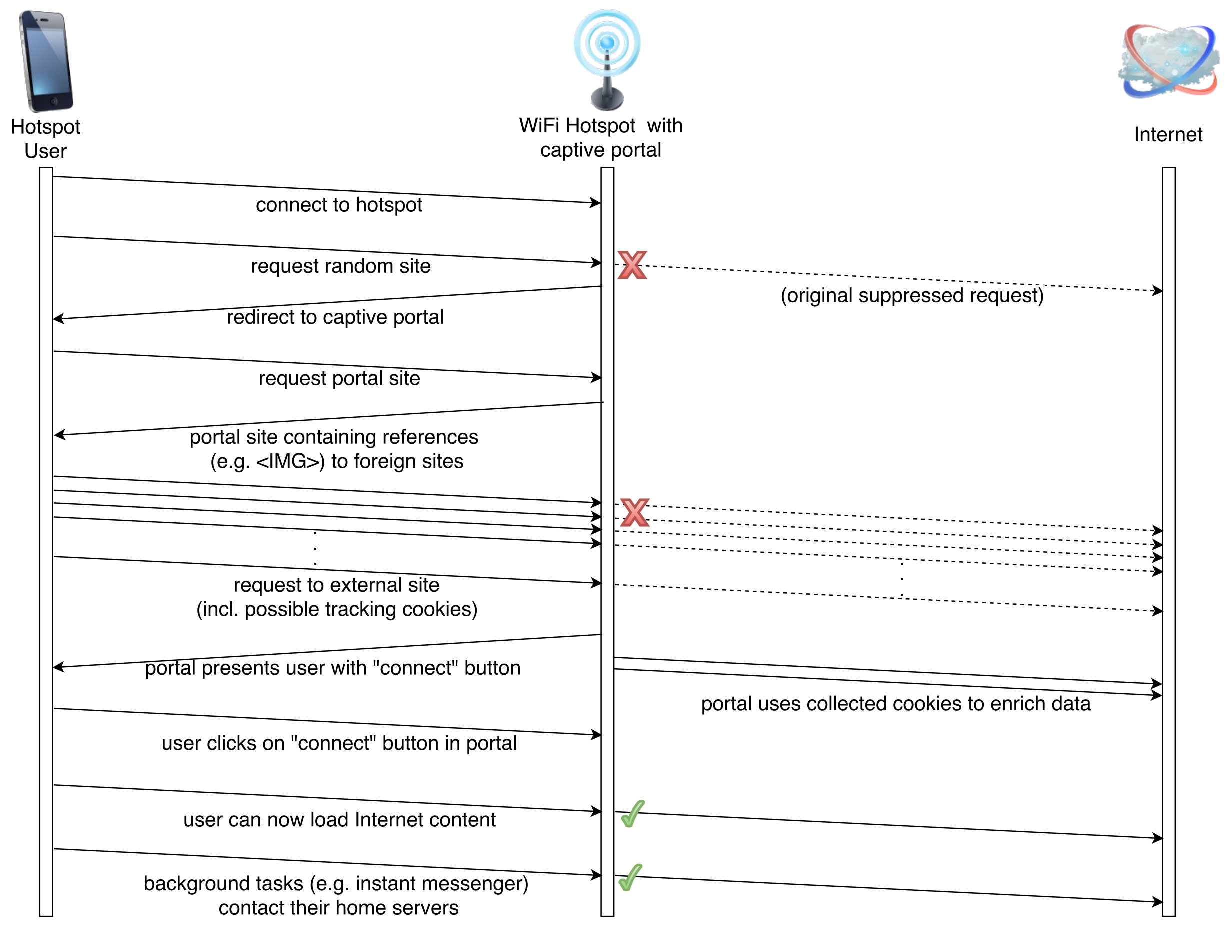
Hotspot User
WiFi Hotspot with captive portal
Internet
connect to hotspot
request random site
(original suppressed request)
redirect to captive portal
request portal site
portal site containing references (e.g. <IMG>) to foreign sites
request to external site (incl. possible tracking cookies)
portal presents user with "connect" button
portal uses collected cookies to enrich data
user clicks on "connect" button in portal
user can now load Internet content
background tasks (e.g. instant messenger) contact their home servers

Fig. 2. History stealing scenario in a public Wi-Fi hotspot setting.

request (and its cookies) and return a dummy file (e.g., a 1×1 pixel image or an empty document).

The browser will include cookies for that site into the request if they are available in the local cookie store. Thus, a cookie included in the provoked request is proof of the user visiting the site before (Figure 1). A missing cookie could indicate that (i) the user has not visited the website, (ii) the website is not setting any cookies, (iii) the cookies already expired, or (iv) the website is setting cookies for a restricted path or sub-domain, or just for secure connections (*secure* cookie).

Websites trying to protect their users using HTTP Strict Transport Security (HSTS) [9] can also accidentally leak history information. HSTS allows websites to declare that browsers should only use encrypted connections. This information is transferred at the first visit of a browser to a particular website. Apart from about 6,500 sites on the public *preload HSTS list* used by particular browsers [10], a cached HSTS entry indicates that the user visited the site before.

Even though the staged requests are intercepted, they take some time. A waiting screen (e.g., "Please wait while you are connected to the Internet") can put off the user only for a limited time before she/he unnervingly closes the window (or the app). We therefore also inject these tagged image references into later HTTP requests with HTML content using a transparent man-in-the-middle proxy on the gateway.

Image (`<img>`) tags are not the only option to generate external requests by the browser. Basically any method used in Cross-Site-Request-Forgery (XSRF) attacks works. Image tags have the advantage of not requiring any Javascript on the client side. They work considerably faster than `<iframe>` which produce considerable overhead at the client.

The attack heavily relies on the usage of long-term cookies by websites. We elaborate on the prevalence of long term cookies in Section VI.

B. *Unintentional Connection*

Once a Wi-Fi network is known to a phone or tablet, every further connection will be made automatically in most cases (e.g., Android, iOS).

For our attack, we will either operate a Wi-Fi hotspot that users actively connect to or simulate a known one (*Evil-Twin* attack [5]). The Wigle project [11] provides a collection of popular Wi-Fi names (SSID). For example *attwifi*, *BTOpenzone*, *public*, *Guest*, *Free Public WiFi* or *BTWIFI* are often used for public hotspots. Default Wi-Fi names are also a good guess, as many smart phone users might have used them once before: *linksys*, *default*, *dlink*, *belkin54g*, or *ZyXEL*.

V. Avoiding Minimalistic Browsers for Captive Portal Login

Computer users will typically use their default desktop browser to be captured and perform the login procedure. Thus, they expose the history of a full-fledged browser that is most likely used for their day-by-day browsing.

In contrast, modern mobile phone operating systems try to detect captive portals and offer the user a way to quickly log on with a minimalistic browser. This browser does not share the history with the main browser.

Android and iOS perform different connectivity checks and display a notification if they assume that user interaction is needed. For example, iOS performs a captive portal test since version 4 and Android since version 4.2. Before Android 5.0, the default system browser was used to load the captive portal which directly exposes the history. Since Android 5.0, the operating system starts a captive portal browser, basically a lightweight browser in privacy mode (e.g., without history).

To prevent the usage of a stripped-down browser, the attacker can fool the online check into believing it is connected. This way, the victims will not get a notification and will use the default browser for the captive portal, thus exposing their actual browsing history.

As stated above, this does not typically apply for users with a desktop browser, as they will use their default browser. However, we know of two exceptions: Chrome OS/Chromium and Mac OS X since 10.7. Both perform the exact same connectivity test as their mobile OS counterparts and can be circumvented in the same manner.

A. *Circumventing the Connectivity Test on Android*

Android, Chromium [12], and Desktop Chrome [13]) creates am HTTP request to one of the Google servers and checks for a specific return code. A plain captive portal will try to redirect the user during this request and rewrite the return code.

The connection manager binds an HTTP client to a specific interface (e.g. Wi-Fi) and tries to request `http://clients3.google.com/generate_204`. A response with a 204 HTTP status code [14, Section 10] indicates an open Internet connection, anything else a captive portal. The attacker can test if the circumvention was successful by testing for the User-Agent string of the lightweight browser (and other minor differences [15]).

B. *Circumventing the Connectivity Test on iOS*

Apple iPhones use a very similar technique to Android. Apple's Captive Network Assistant (CNA) downloads URLs with a known content such as `http://captive.apple.com/hotspot-detect.html` or `http://www.apple.com/library/test/success.html`. The list of tested URLs changed significantly between different versions of the operating system and expanded considerably in later versions. However, the CNA uses a very distinct User-Agent string ("wispr") which has proven to be a good indicator [16]–[18].

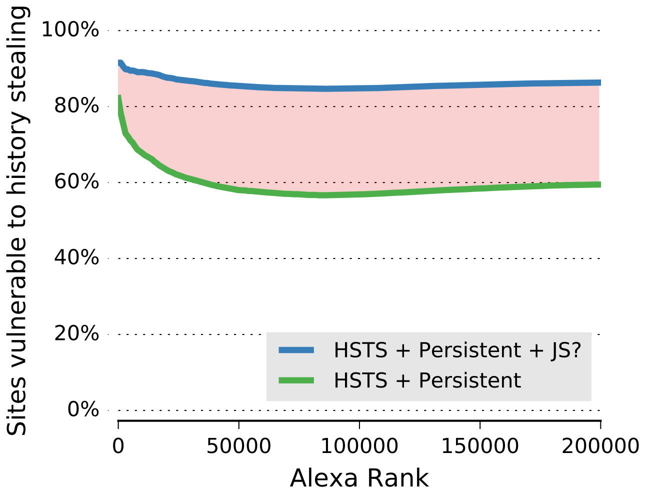

Fig. 3. Percentage of scanned sites that are vulnerable to our history stealing attack in the Alexa Top $200,000$, grouped by steps of 1,000 pages (cumulated). The lower bound represents all pages that either use HSTS or set a persistent cookie. The upper bound additionally includes cookies set through Javascript (includes session cookies). The real number of vulnerable sites is somewhere in the highlighted area.

VI. Assessment of Applicability

To get an impression of the impact of our cookie-based history stealing attack, we analyzed how many webpages actually either use persistent cookies or set up HSTS on their servers. Therefore, we crawled Alexa Top 200,000 websites with a headless browser and recorded the network traffic.

There are two ways how a cookie can be set, either through the `Set-Cookie` header or by using Javascript's `document.cookie` property. Furthermore, a persistent cookie needs to have an expiration date set which lies in the future[1] otherwise the cookie will be deleted as soon as the browser session is closed. Additionally, for the cookie stealing attack to work, the secure flag of the cookie must not be set.

A. *Lower Bound Estimation*

Cookies set directly through the server are easy to detect and filter according to the above mentioned criteria. They can be observed in the server response without the need to evaluate Javascript. However, they only determine the lower bound of pages vulnerable to our history stealing attack.

B. *Upper Bound Estimation*

For an upper bound estimation we also included cookies for which we did not observe the `Set-Cookie` header, but the cookie appeared in a request from the browser to the server. We removed all session (non-persistent) cookies that were set by the server. However, the remaining set will include persistent and non-persistent session cookies set by Javascript. The persistence of a cookie is not indicated by the browser in the requests sent to servers. Therefore, the upper bound overestimates the number of sites by those which use Javascript to set session cookies, but no persistent cookies.

[1]Past dates are used to delete such cookies prematurely.

A short manual inspection showed that many of the Javascript cookies we observed are set by tracking scripts like the __utma persistent cookie set by *Google Analytics*[2]. Therefore the real numbers are more likely closer to the upper than the lower bound.

Additionally, the crawling technique might have missed cookies set under rare conditions (such as sub-pages, configuration, tracking opt-out cookies) making the real numbers even higher.

C. Methodology

We set up headless browsers on Amazon EC2 Spot instances to visit the Alexa Top 200,000 websites in an attempt to determine the pervasiveness of persistent cookies.

For every website, we crawled the main page and three randomly selected sub-pages and collected the corresponding responses. We then extracted the headers from each of the requests and searched for the `Set-Cookie` headers. If these headers are found in one of the responses, the corresponding expiry date is set to a future date[3], and the `secure` flag is not set, we count the page as setting a persistent cookie. Furthermore we also look for the `Strict-Transport-Security` header to see if HSTS is enabled for the page. for the bounds calculation, entries in the HSTS pre-load list used by several browsers were excluded from the set of possibly vulnerable pages.

D. Results

As the graph in Figure 3 shows, even without considering cookies set through Javascript, on average about 59.47% of all pages are either using HSTS or are setting at least some kind of persistent cookie. For the top 1,000 pages this persistent cookie usage spikes up to 82.24%. As mentioned before, these results only indicate a lower bound. If we also count cookies that were set through Javascript, the numbers for vulnerable pages rise to 91.52% for the top 1,000 pages and 86.31% on average.

These results clearly indicate that there exists a large attack surface for the captive-portal-based history stealing attack.

VII. Proof-of-Concept Implementation

We created a proof-of-concept privacy-invading Wi-Fi hotspot that might also be used for raising awareness in educational settings or to perform public demonstrations (cf. [19]). It implements the history stealing attack as well as several additional attacks and provides a per-user result page showing in plain language which data has been automatically collected.

Figure 4 gives an overview of the implementation structure. It runs on a virtual machine connected to an USB Wi-Fi adapter[4] in access point (AP) mode. A set of Python scripts extends a Wi-Fi Router/NAT iptables setup.

[2]https://developers.google.com/analytics/devguides/collection/analyticsjs/cookie-usage?hl=en (Accessed: 2016–01–29)

[3]includes the *Expiry* and the *Max-age* option

[4]TP-LINK TL-WN722N

HTTP traffic is diverted to a man-in-the-middle (MITM) proxy that can intercept and manipulate passing HTTP requests. This is the main component of the history stealing attack (see Section VII-A).

To further demonstrate other risks of public hotspots, we also added a small number of additional attacks. A number of passive attacks is implemented by simple packet sniffing on unencrypted non-HTTP protocols, such as plain-text passwords or DNS-snooping (see Section VII-B). This includes the detection of installed applications based on simple network patterns, deriving the full name and gender of the user from the device name or through third-party websites (using the captured cookies). We also added active exploits for security vulnerabilities (e.g., WebView) by injecting code into the HTTP traffic.

A. Implementation of History Stealing

In principle, there are many ways to compel a browser to make external requests. We tested three methods: Javascript XML-RPC requests, IFrames, and IMG tags. The latter turned out to be the fastest. It is lightweight and facilitates the browsers ability to parallelize requests. These requests are inserted into the portal webpage and/or into other HTML pages loaded via HTTP. These induced requests are detected by the MITM proxy based on marker strings within the request (e.g.,

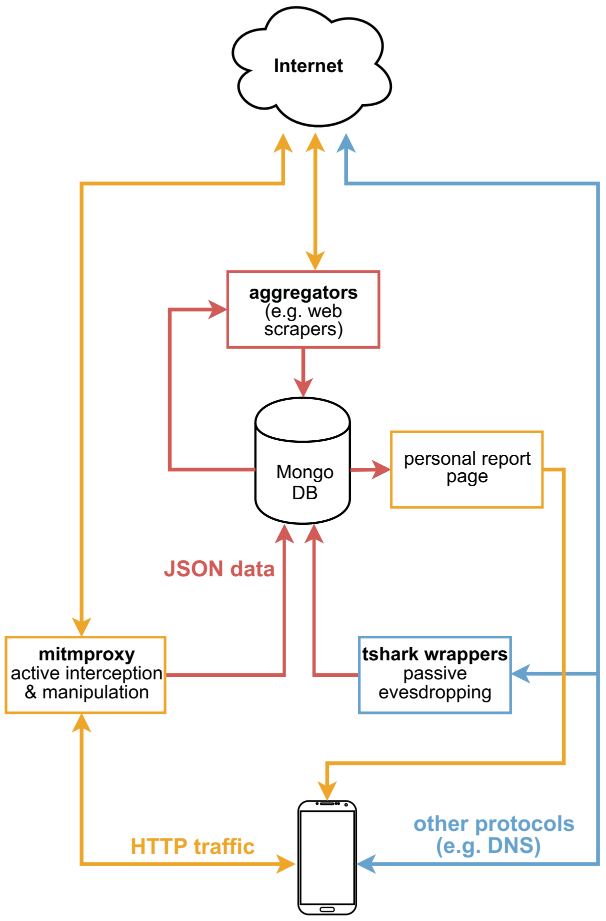

Fig. 4. Structure of the implementation

as part of the URL). The mitm-proxy collects these requests and answers them locally. The latter serves two purposes: (a) it speeds up the requests and (b) it prevents the client from collecting additional cookies through the probing requests.

B. Further Attacks by Malicious Hotspots

For Android phones we implemented sample exploits based on the WebView component [20], [21]. WebView is a UI element which allows developers to simply display local or remote HTML content. It offers a Javascript application bridge that allows arbitrary calls to public functions in the application, including *Java Reflection*. It is still present in today's applications if they are compiled for Android versions below 4.3. One of our test scripts steals the latest photos shot with the phone's camera, while another one lists files from the download folder. Additionally, a network traffic sniffer extracts the listened-to music titles from Shazam's [22] traffic.

C. Personal Data Enrichment

Based on the data acquired by the MITM proxy and passive sniffing, small demonstration scripts try to enrich the dataset with additional information from other external sources. Amazon's public profile feature can be used to acquire the user's full name, even if she or he is not actively logged in, since it is enabled by default. Subsequently we use the name to determine the gender of the user via a public demographic database API.

1) Amazon Name Disclosure: Amazon.com has three states for user sessions: A user can be either *logged in*, *half logged in*, or *not logged in* at all. The half-logged-in state is a specialty of Amazon. Amazon recognizes – with high probability – based on an earlier long term plain-text cookies a particular user even if the user's session has expired. Amazon allows the user to add products to the cart, but will ask for the user's password before any further action. By requesting `http://amazon.com/gp/profile` (or the appropriate localized Amazon site) a public profile including wish lists, reviews, and (by default) the full name of the user can be accessed. A user can turn off the visibility of most elements in the account settings.

2) Gender Estimation: Genderize.io [23] offers a simple service to look up the gender of a person based on their first name. We display this information in the user report.

3) Device (and User) Name: During setup, iOS suggests to use *[Firstname]'s iPhone* as the device name. This name is broadcast on the Wi-Fi using mDNS as part of the zero-config peer discovery protocol *Bonjour*. It offers another way to get hold of a user's first name.

D. Device Parameters

During the captive portal phase – and later due to HTTP injection – the browser can be tricked into loading and executing arbitrary Javascript code. They can read out device parameters (model and brand, screen size, installed extensions), but also track a user's site visits (scrolling, clicking).

E. Installed Applications based on Network Traffic

Some ad networks and most instant messengers regularly connect to their *home* servers. In the latter case, this is necessary to ensure that incoming instant messages are delivered over the current Internet connection. Therefore, most such applications subscribe to system messages indicating a change in the network connectivity. Connecting to a Wi-Fi network (like our malicious one) is such an event. While many instant messengers switched to encrypted traffic, the target of the traffic (and the plain text DNS lookup before that) give away enough information to identify the application[5]. We included a few handwritten rules that identify WhatsApp, Facebook Messenger, Instagram, Snapchat, Skype, and several other services. Other works, such as NetworkProfiler [24] and AppScanner [25] could easily enhance this detection.

F. Report Page

The report page sorts the collected information by its technicality and privacy invasiveness. Thus, it presents high-level information (e.g. the user's real name) at the top, and more technical details (e.g. history) below. This allows a non-technical user to read the important information first.

VIII. Limitations and Future Work

One of the major speed limitations of our implementation is the DNS lookup round-trip time that each new site requires. We can pre-load them into the cache on our hotspot, but not to the client. For the browser, each reference needs to look like a new site. Depending on the used mobile phone the attack performs up to roughly 50-100 history tests per second. Several optimization ideas (such as prematurely killing connections as soon as data is collected) are left for future work.

We are currently injecting history stealing references into the portal and into HTTP traffic. However, we do not catch HTTPS connections and therefore cannot detect cached HSTS entries. Additionally, we experimented with SSL-stripping and SSL-proxying, however, the failure rate was so high that we are currently not using it. It would require extensive white- or black-listing approaches to only affect sites and apps that do no or insufficient certificate validation. We see this as a positive sign of increased awareness among developers.

A limitation of our current hardware setup is the usage of the *TL-WN722N* Wi-Fi stick as an access point. In general, the chipset has good Linux support, but the firmware is limited to seven clients in access point mode.

Since our implementation is meant to evolve to an educational or demonstrative tool (e.g. for school children, students, or not so technology versed people) we like to add more automated aggregation scripts and app reconnaissance mechanisms such as NetworkProfiler [24] or AppScanner [25]. This should serve the goal of increasing awareness regarding the risks of public Wi-Fi hotspots.

[5]This also works in many cases where the application is using a cloud messaging service such as Google Cloud Messaging (GCM), because it needs to communicate the GCM token to its home server.

IX. Conclusion

History stealing is one of the most privacy-invasive attacks. Several methods have been developed in the past, most of which have to probe each individual URL and are now mitigated by current browsers.

We described a site-level history stealing attack which is easy to perform by Man-in-the-Middle attackers and (even more easily) by operators of captive portals, such as those found in many public Wi-Fi hotpots. Captive portals enjoy a privileged network position easing the implementation of the attack. It affects notebooks, mobile phone, and tablet users alike. The attack is difficult to prevent even for VPN users.

The attack operates by supplying the client browser with a vast numbers of external references (e.g., images) which it will try to download. It is not necessary to filter out these requests at the gateway, but it speeds up the attack. During these requests, the browser will use cookies from its persistent cookie database. A presented cookie for a particular domain is a clear indicator that the user has visited that site before. We also described how a cached HSTS entry reveals a user's past visit.

We further investigated the prevalence of such long-term cookies to estimate the attack surface. Out of the Alexa Top 1,000 sites, between 82.24% and 91.52% are affected (depending on how they set their cookies). The rate approaches 59.47% to 86.31% for the Alexa Top 200,000 dataset.

As a proof of concept we implemented this attack together with several other privacy-invading attacks to create an environment where participants can educate themselves about data leakage in untrusted Internet environments. Additional scripts enrich this data with information from other sites (e.g. Amazon's real name disclosure). It is meant as a tool to create awareness regarding privacy issues for people without a technical background.

As an immediate countermeasure, we encourage web developers to switch all their sites to HTTPS, optionally add them to the HSTS preload list, and to only use secure cookies. Additionally, all operating systems should offer a stripped-down browser to log on to captive portals with a less predictable connectivity check.

Acknowledgment

This work is loosely inspired by Sergey Bratus' Tweet about captive portals [26].

This research was partially funded by the Internet Foundation Austria (IPA) through their *netidee* program and by the *COMET K1* program of the Austrian Research Funding Agency (FFG) and the Austrian Science Fund (FWF). Additional support provided by the Josef Ressel Center for User-friendly Secure Mobile Environments (u'smile).

References

[1] D. Baron, "Bug 147777 - :visited support allows queries into global history," 2002, http://bugzilla.mozilla.org/show_bug.cgi?id=147777, accessed 2016-01-21.

[2] J. Ruderman, "Bug 57351 - css on a:visited can load an image and/or reveal if visitor been to a site," 2000, http://bugzilla.mozilla.org/show_bug.cgi?id=57351, accessed 2016-01-21.

[3] Z. Braniecki, "CSS allows to check history via :visited," https://bugzilla.mozilla.org/224954, 2003.

[4] 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*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 270–283. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866339

[5] F. Lanze, A. Panchenko, I. Ponce-Alcaide, and T. Engel, "Undesired relatives: protection mechanisms against the evil twin attack in IEEE 802.11," in *Proceedings of the 10th ACM symposium on QoS and security for wireless and mobile networks*. ACM, 2014, pp. 87–94.

[6] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel, "A Practical Attack to De-anonymize Social Network Users," in *Security and Privacy (SP), 2010 IEEE Symposium on*, May 2010, pp. 223–238.

[7] S. Lee, Y. Kim, J. Kim, and J. Kim, "Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities," in *Security and Privacy (SP), 2014 IEEE Symposium on*, May 2014, pp. 19–33.

[8] A. Barth, *HTTP State Management Mechanism*, RFC 6265, Internet Engineering Task Force (IETF) Std., 2011.

[9] J. Hodges, C. Jackson, and A. Barth, *HTTP Strict Transport Security (HSTS)*, RFC-6797, Internet Engineering Task Force (IETF) Std., 2012.

[10] "HSTS pre-load submission," https://hstspreload.appspot.com/, accessed 2016-01-31.

[11] "WiGLE Stats," https://wigle.net/stats#ssidstats, accessed 2015-01-25.

[12] "Network Portal Detection - The Chromium Projects," https://www.chromium.org/chromium-os/chromiumos-design-docs/network-portal-detection, accessed 2015-01-20.

[13] "Google Chrome Privacy Whitepaper," 12 2015, https://www.google.com/chrome/browser/privacy/whitepaper.html, accessed 2015-01-20.

[14] "Hypertext Transfer Protocol - HTTP/1.1- Section 10, Status Code Definitions," RFC 2616, 1999, https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

[15] "Is it possible to detect the Android captive portal browser?" http://stackoverflow.com/questions/32950326/is-it-possible-to-detect-the-android-captive-portal-browser, accessed 2015-01-20.

[16] "iOS 7 and captive portal - a guide to captive portal requirements," http://blog.tanaza.com/blog/bid/318805/iOS-7-and-captive-portal-a-guide-to-captive-portal-requirements, accessed 2015-01-20.

[17] "iOS7 and captive portals-changes to apple request URL," http://stackoverflow.com/questions/18891706/ios7-and-captive-portals-changes-to-apple-request-url, accessed 2015-01-20.

[18] M. Strubel and S. Pierre, "iOS9 & Android with offline networks," http://librelist.com/browser//off.networks/2015/10/14/ios9-android-with-offline-networks/, accessed 2015-01-20.

[19] R. Balebako, J. Jung, W. Lu, L. F. Cranor, and C. Nguyen, "Little brothers watching you: Raising awareness of data leaks on smartphones," in *Proceedings of the Ninth Symposium on Usable Privacy and Security*. ACM, 2013, p. 12.

[20] M. Neugschwandtner, M. Lindorfer, and C. Platzer, "A View to a Kill: WebView Exploitation," in *Presented as part of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2013.

[21] N. Bergman, "Abusing WebView JavaScript Bridges," http://d3adend.org/blog/?p=314, Dec. 2012, accessed: 2014-09-15.

[22] "Shazam - Music Discovery, Charts & Song Lyrics," http://www.shazam.com/, accessed 2016-02-01.

[23] "Determine the gender of a first name," https://genderize.io/, accessed 2016-01-20.

[24] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, "Networkprofiler: Towards automatic fingerprinting of android apps," in *INFOCOM, 2013 Proceedings IEEE*. IEEE, 2013, pp. 809–817.

[25] V. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic," in *1st IEEE European Symposium on Security and Privacy (Euro S&P 2016)*, March 2016, to appear.

[26] S. Bratus, "As more sites go HTTPS & more Wi-Fi goes captive portal, I find myself treasuring short names of plain old HTTP sites that get MITMed faster," Nov 2015, https://twitter.com/sergeybratus/status/664182669119483904.