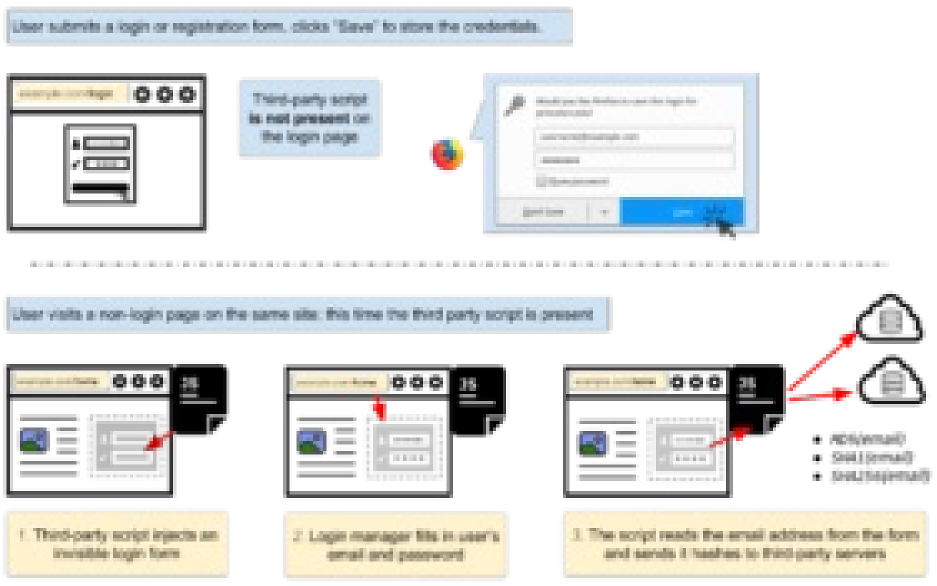
# No boundaries for user identities: Web trackers exploit browser login managers

DECEMBER 27, 2017 BY GUNES ACAR 4 COMMENTS

*In this second installment of the **No Boundaries series**, we show how a long-known vulnerability in browsers' built-in password managers is abused by third-party scripts for tracking on more than a thousand sites.*

by Gunes Acar, Steven Englehardt, and Arvind Narayanan


User submits a login or registration form, clicks "Save" to store the credentials.
Third-party script is not present on the login page
User visits a non-login page on the same site: this time the third party script is present
MD5(email)
SHA1(email)
SHA256(email)
1. Third-party script injects an invisible login form
2. Login manager fills in user's email and password
3. The script reads the email address from the form and sends it hashes to third-party servers

We show how third-party scripts exploit browsers' built-in login managers (also called password managers) to retrieve and exfiltrate user identifiers without user awareness. To the best of our knowledge, our research is the first to show that login managers are being abused by third-party scripts for the purposes of web tracking.

The underlying vulnerability of login managers to credential theft **has been known for years**. Much of the past discussion has focused on password exfiltration by malicious scripts through cross-site scripting (XSS) attacks. Fortunately, we haven't found password theft on the 50,000 sites that we analyzed. Instead, we found tracking scripts embedded by the first party abusing the same technique to extract emails addresses for building tracking identifiers.

The image above shows the process. First, a user fills out a login form on the page and asks the browser to save the login. The tracking script is not present on the login page [1]. Then, the user visits another page on the same website which includes the third-party tracking script. The tracking script inserts an invisible login form, which is automatically filled in by the browser's login manager. The third-party script retrieves the user's email address by reading the populated form and sends the email hashes to third-party servers.

You can test the attack yourself on our live **demo page**.

Search this website … Search

## What We Discuss

AACS bitcoin CD Copy Protection censorship CITP Competition Computing in the Cloud Copyright Cross-Border Issues cybersecurity policy DMCA DRM Education Events Facebook FCC Government Government transparency Grokster Case Humor Innovation Policy Law Managing the Internet Media Misleading Terms NSA Online Communities Patents Peer-to-Peer Predictions Princeton Privacy Publishing Recommended Reading Secrecy Security Spam Super-DMCA surveillance Tech/Law/Policy Blogs Technology and Freedom Virtual Worlds Voting Wiretapping WPM

## Contributors

Select Author...

## Archives by Month

- 2017: J F M A M J J A S O N D
- 2016: J F M A M J J A S O N D
- 2015: J F M A M J J A S O N D
- 2014: J F M A M J J A S O N D
- 2013: J F M A M J J A S O N D
- 2012: J F M A M J J A S O N D
- 2011: J F M A M J J A S O N D
- 2010: J F M A M J J A S O N D
- 2009: J F M A M J J A S O N D
- 2008: J F M A M J J A S O N D
- 2007: J F M A M J J A S O N D
- 2006: J F M A M J J A S O N D

00:00 / 00:00

We found two scripts using this technique to extract email addresses from login managers on the websites which embed them. These addresses are then hashed and sent to one or more third-party servers. These scripts were present on 1110 of the Alexa top 1 million sites. The process of detecting these scripts is described in our measurement methodology in the Appendix 1. We provide a brief analysis of each script in the sections below.

**Why does the attack work?** All major browsers have built-in login managers that save and automatically fill in username and password data to make the login experience more seamless. The set of heuristics used to determine which login forms will be autofilled **varies by browser**, but the basic requirement is that a **username and password field be available**.

Login form autofilling in general doesn't require user interaction; all of the major browsers will autofill the username (often an email address) immediately, regardless of the visibility of the form. Chrome doesn't autofill the password field until the user clicks or touches anywhere on the page. Other browsers we tested [2] don't require user interaction to autofill password fields.

Thus, third-party javascript can retrieve the saved credentials by creating a form with the username and password fields, which will then be autofilled by the login manager.

**Why collect hashes of email addresses?** Email addresses are unique and persistent, and thus the hash of an email address is an excellent tracking identifier. A user's email address will almost never change — clearing cookies, using private browsing mode, or switching devices won't prevent tracking. The hash of an email address can be used to connect the pieces of an online profile scattered across different browsers, devices, and mobile apps. It can also serve as a link between browsing history profiles before and after cookie clears. In a **previous blog post on email tracking**, we described in detail why a hashed email address is not an anonymous identifier.

**Scripts exploiting browser login managers**
***List of sites embedding scripts that abuse login manager for tracking***
"*Smart Advertising Performance*" and "*Big Data Marketing*" are the taglines used by the two companies who own the scripts that abuse login managers to extract email addresses. We have manually analyzed the scripts that contained the attack code and verified the attack steps described above. The snippets from the two scripts are given in Appendix 2.

*The scripts that use login manager to extract email addresses present on a total of 1110 of the top 1 Million Alexa sites.*

**Adthink (audienceinsights.net):** After injecting an invisible form and reading the email address, Adthink script sends MD5, SHA1 and SHA256 hashes of the email address to its server (secure.audienceinsights.net). Adthink then triggers another request containing the MD5 hash of the email to data broker Acxiom (p-eu.acxiom-online.com).

The Adthink script contains very detailed categories for personal, financial, physical traits, as well as intents, interests and demographics. It is hard to comment on the exact use of these categories but it gives a glimpse of what our online profiles are made up of:

```
birth date, age, gender, nationality, height, weight, BMI (body mass index), hair_color (black, brown,
blond, auburn, chestnut, red, gray, white), eye_color (amber, blue, brown, grey, green), education,
occupation, net_income, raw_income, relationship states, seek_for_gender (m, f, transman, transwoman,
couple), pets, location (postcode, town, state, country), loan (type, amount, duration, overindebted),
insurance (car, motorbike, home, pet, health, life), card_risk (chargeback, fraud_attempt), has_car(make,
model, type, registration, model year, fuel type), tobacco, alcohol, travel (from, to, departure,
return), car_hire_driver_age, hotel_stars
```

The categories mentioned in the Adthink script include detailed personal, financial, physical traits, as well as intents, interests and demographics (**Link to the code snippet**).

**OnAudience (behavioralengine.com):** The OnAudience script is most commonly present on Polish websites, including newspapers, ISPs and online retailers. 45 of the 63 sites that contain OnAudience script have ".pl" country code top-level domain.

The script sends the MD5 hash of the email back to its server after reading it through the login manager. OnAudience script also collects browser features including plugins, MIME types, screen dimensions, language, timezone information, user agent string, OS and CPU information. The script then generates a hash based on this browser fingerprint. OnAudience **claims to use anonymous data only**, but hashed email addresses are not anonymous. If an attacker wants to determine whether a user is in the dataset, they can simply hash the user's email address and search for records associated with that hash. For a more detailed discussion, see our **previous blog post**.

OnAudience marketing material that advertises "billions of user profiles".

**Is this attack new?** This and similar attacks have been discussed in a number of **browser bug reports** and **academic papers** for at **least 11 years**. Much of the previous discussion focuses on the security implications of the current functionality, and on the security-usability tradeoff of the autofill functionality.

Several researchers showed that it is possible to steal passwords from login managers through cross-site scripting (XSS) attacks [3,4,5,6,7]. Login managers and XSS is a dangerous mixture for two reasons: 1) passwords retrieved by XSS can have more devastating effects compared to cookie theft, as users **commonly reuse passwords across different sites**; 2) login managers extend the attack surface for the password theft, as an XSS attack can steal passwords on any page within a site, even those which don't contain a login form.

**How did we get here?** You may wonder how a security vulnerability persisted for 11 years. That's because from a narrow browser security perspective, there is no vulnerability, and everything is working as intended. Let us explain.

The web's security rests on the Same Origin Policy. In this model, scripts and content from different origins (roughly, domains or websites) are treated as mutually untrusting, and the browser protects them from interfering with each other. However, if a publisher directly embeds a third-party script, rather than isolating it in an iframe, the script is treated as coming from the publisher's origin. Thus, the publisher (and its users) entirely lose the protections of the same origin policy, and there is nothing preventing the script from exfiltrating sensitive information. Sadly, direct embedding is common — and, in fact, the default — which also explains why the vulnerabilities we exposed in our **previous post** were possible.

This model is a poor fit for reality. Publishers neither completely trust nor completely mistrust third parties, and thus neither of the two options (iframe sandboxing and direct embedding) is a good fit: one limits functionality and the other is a privacy nightmare. We've found repeatedly through our research that third parties are quite opaque about the behavior of their scripts, and at any rate, most publishers don't have the time or technical knowhow to evaluate them. Thus, we're stuck with this uneasy relationship between publishers and third parties for the foreseeable future.

**The browser vendor's dilemma.** It is clear that the Same-Origin Policy is a poor fit for trust relationships on the web today, and that other security defenses would help. But there is another dilemma for browser vendors: should they defend against this and other similar vulnerabilities, or view it as the publisher's fault for embedding the third party at all?

There are good arguments for both views. Currently browser vendors seem to adopt the latter for the login manager issue, viewing it as the publisher's burden. In general, there is no principled way to defend against third parties that are present on some pages on a site from accessing sensitive data on other pages of the same site. For example, if a user simultaneously has two tabs from the same site open — one containing a login form but no third party, and vice versa — then the third-party script can "reach across" browser tabs and exfiltrate the login information **under certain circumstances**. By embedding a third party *anywhere* on its site, the publisher signals that it completely trusts the third party.

Yet, in other cases, browser vendors have chosen to adopt defenses even if necessarily imperfect. For example, the HTTPOnly cookie attribute was introduced to limit the impact of XSS attacks by blocking the script access to security critical cookies.

There is another relevant factor: our discovery means that autofill is not just a security vulnerability but also a privacy threat. While the security community strongly prefers principled solutions whenever possible, when it comes to web tracking, we have generally been willing to embrace more heuristic defenses such as blocklists.

**Countermeasures.** Publishers, users, and browser vendors can all take steps to prevent autofill data exfiltration. We discuss each in turn.

Publishers can isolate login forms by putting them on a separate subdomain, which prevents autofill from working on non-login pages. This does have drawbacks including an increase in engineering complexity. Alternately they could isolate third parties using frameworks like **Safeframe**. Safeframe makes it easier for the publisher scripts and iframed scripts to communicate, thus blunting the effect of sandboxing. Any such technique requires additional engineering by the publisher compared to simply dropping a third-party script into the web page.

Users can install ad blockers or tracking protection extensions to prevent tracking by invasive third-party scripts. The domains used to serve the two scripts (behavioralengine.com and audienceinsights.net) are blocked by the EasyPrivacy blocklist.

Now we turn to browsers. The simplest defense is to allow users to disable login autofill. For instance, the Firefox preference `signon.autofillForms` can be set to false to disable autofilling of credentials.

A less crude defense is to require user interaction before autofilling login forms. Browser vendors have been reluctant to do this because of the usability overhead, but given the evidence of autofill abuse in the wild, this overhead might be justifiable.

The upcoming **W3C Credential Management API** requires browsers to display a notification when user credentials are provided to a page [8]. Browsers may display the same notification when login information is autofilled by the built-in login managers. Displays of this type won't directly prevent abuse, but they make attacks more visible to publishers and privacy-conscious users.

Finally, the "**writeonly form fields**" idea can be a promising direction to secure login forms in general. The **briefly discussed proposal** defines ways to deny read access to form elements and suggests the use of **placeholder nonces to protect autofilled credentials** [9].

## Conclusion

Built-in login managers have a positive effect on web security: they curtail password reuse by making it easy to use complex passwords, and they make phishing attacks are harder to mount. Yet, browser vendors should reconsider allowing stealthy access to autofilled login forms in the light of our findings. More generally, for every browser feature, browser developers and standard bodies should consider how it might be abused by untrustworthy third-party scripts.

---

**End notes:**

[1] We found that login pages contain 25% fewer third-parties compared to pages without login forms. The analysis was based on our crawl of 300,000 pages from 50,000 sites.
[2] We tested the following browsers: Firefox, Chrome, Internet Explorer, Edge, Safari.
[3] **https://labs.neohapsis.com/2012/04/25/abusing-password-managers-with-xss/**
[4] **https://www.honoki.net/2014/05/grab-password-with-xss/**
[5] **https://web.archive.org/web/20150131032001/http://ha.ckers.org:80/blog/20060821/stealing-user-information-via-automatic-form-filling/**
[6] **http://www.martani.net/2009/08/xss-steal-passwords-using-javascript.html**
[7] **https://ancat.github.io/xss/2017/01/08/stealing-plaintext-passwords.html**
[8] "User agents MUST notify users when credentials are provided to an origin. This could take the form of an icon in the address bar, or some similar location." **https://w3c.github.io/webappsec-credential-management/#user-mediation-requirement**
[9] Originally proposed in **https://www.ben-stock.de/wp-content/uploads/asiacss2014.pdf**
[10] **https://jacob.hoffman-andrews.com/README/2017/01/15/how-not-to-get-phished.html**

---

## APPENDICES

### Appendix 1 – Methodology

To study password manager abuse, we extended OpenWPM to simulate a user with saved login credentials and added instrumentation to monitor form access. We used Firefox's **nsILoginManager** interface to add login credentials as if they were previously stored by the user. We did not otherwise alter the functionality of the password manager or attempt to manually fill login forms. This allowed us to capture actual abuses of the browser login manager, as any exfiltrated data must have originated from the login manager.

We crawled 50,000 sites from the Alexa top 1 million. We used the following sampling strategy: visit all of the top 15,000 sites, randomly sample 15,000 sites from the Alexa rank range [15,000 100,000), and randomly sample 20,000 sites from the range [100,000, 1,000,000). This combination allowed us to observe the attacks on both high and low traffic sites. On each of these 50,000 sites we visited 6 pages: the front page and a set of 5 other pages randomly sampled from the internal links on the front page.

The fake login credentials acted as bait, allowing us to introduce an email and password to the page that could be collected by third parties without any additional interaction. Detection of email address collection was done by inspecting JavaScript calls related to form creation and access, and by the analysis of the HTTP traffic. Specifically, we used the following instrumentation:

1. **Mutation events** to monitor elements inserted to the page DOM. This allowed us to detect the injection of fake login forms. When a mutation event fires, we record the current call stack and serialize the inserted HTML elements.
2. Instrument HTMLInputElement to intercept access to form input fields. We log the input field value that is being read to detect when the bait email (autofilled by the built-in password manager) was sniffed.
3. Store HTTP request and response data, including POST payloads to detect the exfiltration of the email address or password.

For both JavaScript (1, 2) and HTTP instrumentation (3) we store JavaScript stack traces at the time of the function call or the HTTP request. We then parse the stack trace to pin down the initiators of an HTTP request or the parties responsible for inserting or accessing a form.

We then combine the instrumentation data to select scripts that:

1. inject an HTML element containing a password field (recall that the password field is necessary for the built-in password manager to kick in)
2. read the email address from the input field automatically filled by the browser's login manager
3. send the email address, or a hash of it, over HTTP

To verify the findings of the automated experiments we manually analyzed sites that embed the two scripts that match these conditions. We have verified that the forms that the scripts inserted were not visible. We then opened accounts on the sites that allow registration and let the browser store the login information (by clicking yes to the dialog in Figure 1). We then visited another page on the site and verified that browser password manager filled the invisible form injected by the scripts.

**Appendix 2 – Code Snippets**

Code snippets from OnAudience (left) and Adthink (right) that are responsible for the injection of invisible login forms.

FILED UNDER: PRIVACY TAGGED WITH: DATA PRIVACY, LOGIN MANAGERS, PASSWORD MANAGERS, PRIVACY, WEB PRIVACY, WPM

## Comments

**Eric Mill says:**
December 27, 2017 at 5:48 pm

Anything that adds usability overhead to password manager auto-fill feels like a challenging proposal. (And user opt-out is always a relatively ineffective control to mitigate systemic issues like this.)

But what about auto-applying the write-only property to a form as soon as it's been auto-filled? In other words, once the browser has auto-filled a field, the field is considered to be in a locked-down state with no further DOM access. That could create some publisher pain for those who are using JS to access the email field in legit ways to instrument a better login form, but that would be putting the burden on a small class of websites, and not on users using auto-fill.

Reply

**Gunes Acar says:**
December 28, 2017 at 1:55 pm

That sounds like an interesting idea to explore. One can imagine autofilled credentials are not needed to be checked for password strength or duplicate usernames – common cases of legit script access to login forms. Still, one needs telemetry or web measurement data to back this up.

The question is whether browsers will ever ship write-only elements or similar protections 🙂

Reply

**Peter says:**
December 28, 2017 at 6:37 am

Here, firefox addon Privacy Badger (PB) immediately flagged rawgit.com as a tracker and blocked the sniffer script. Probably it was known before? Sadly, it is not really an option to recommend PB: users do enjoy the faster page-loads, but when a site breaks, and that can happen, they are clueless at first, and then annoyed by the fact, that PB cannot read their minds and so they have to manage something; even though the PB interface is quite easy to use, IMO.

Reply

**Gunes Acar says:**
December 28, 2017 at 2:01 pm

Just to clarify, we use RawGit (rawgit.com) to serve code from GitHub with the right content type.

https://github.com/rgrove/rawgit/blob/master/FAQ.md

RawGit is not responsible for any of the code that they seem to serve. Privacy Badger must have flagged it perhaps because some other sites embed code from GitHub through rawgit.com.

Reply

## Speak Your Mind

Name

Email

Post Comment