Master thesis
Cyber Security

Radboud University

`leak-detect`: Automatic login form leakage detection for website administrators and researchers

Author:
Steven Wallis de Vries
S1011387
`SWallisDeVries@science.ru.nl`

First supervisor/assessor:
Güneș Acar
`g.acar@cs.ru.nl`

Erik Poll
`erikpoll@cs.ru.nl`

January 31, 2023

Abstract

Many websites contain third-party trackers, both for advertising and analytics. Previous studies have shown that these trackers often collect personal data of users, such as an email address that was entered into a form. Trackers can also accidentally collect passwords, sometimes because the password was copied to a DOM attribute in the document. Regularly, website administrators do not have a full overview of all the data that is leaked on their website to trackers. For this study, we developed `leak-detect`, a tool that website administrators can use to determine if trackers on their site collect personal data filled into forms. Compared to previous studies, this tool has the advantage that it is self-contained, with built-in leak and tracker detection, and it has some extra features such as shadow DOM support and detection of passwords leaking to attributes. We used this tool to conduct a crawl of the top 30K websites with login forms, and found the most-leaking parties to be Facebook, Google, and AtData. We discovered that the React framework caused most leaks of passwords to attributes. On 12 websites, we found unintentional password leaks to third parties. At least two of these were fixed due to our disclosures. Finally, we make some recommendations on how to prevent leakage of personal data.

Contents

1 Introduction **2**

2 Preliminaries **4**
- 2.1 Page structure 4
- 2.2 Data collection 4
- 2.3 Tracking identifiers 5
- 2.4 Minified script bundles and source maps 5
- 2.5 URLs and domain names 5

3 Related work **7**

4 `leak-detect` **10**
- 4.1 Requirements 10
- 4.2 Crawler behavior and implementation 11

5 `value-searcher` **15**
- 5.1 Operation 15
- 5.2 Comparison with LeakDetector 16

6 Crawl setup **18**

7 Results **20**
- 7.1 Request leaks 21
- 7.2 DOM leaks 26
- 7.3 Password leaks 28

8 Discussion **32**
- 8.1 Comparison with other studies 32
- 8.2 Limitations 34
- 8.3 `leak-detect` evaluation 38
- 8.4 Future work 38
- 8.5 Recommendations 39

9 Conclusion **42**

Bibliography **44**

Appendices **48**

A Example `leak-detect` result summaries **49**
- A.1 Summary with password leak to tracker 49
- A.2 Summary with DOM attribute leak 51

B Extra figures **54**

C Password leaks **55**
- C.1 Timeline regarding Crash In You leak 55
- C.2 Found password leaks 57

Chapter 1

Introduction

Websites may include third-party trackers for various purposes, such as advertising and analytics. With a lot of websites offering services free of charge, many of them rely on advertisers for their income. To make it more appealing for companies to advertise on a given platform, the platform tries to maximize the relevance of advertisements to users, by tracking the user's interests. This can be done by looking at what other websites they visit and what they do there. For example, what products a user has bought online lately. To track someone over longer periods of time and across devices, different sessions may be linked using personal identifiers, such as some hash value derived from the user's email address or phone number.

Besides that, websites use analytics services to get insight into what demographics visit, how these users got to the website, and how they behave. As an example, take a website offering paid services that wants to increase the number of people buying their product. To find out if users can easily navigate the site and find what they are looking for, many such commercial websites are now using session replay scripts. These scripts can record every interaction of a user with the site, from mouse movements and clicks to what they enter into an input field. This data is sent to and stored at the third party that delivers the session replay script, not at the first party. Among these scripts are Hotjar, Yandex Metrika, Microsoft Clarity, and FullStory [1–4]. The number of websites using these scripts has increased substantially in the last couple of years [5]. Regularly, these scripts will also capture personally identifiable information (PII), such as an email address, or they might even accidentally record credit card numbers or passwords [6]. Note that because of the nature of session replay, this data will be captured even before the user submits the form. Usually there is the option to mask information that the user enters into input fields [1], but leaks may still occur in other ways. For example, if text on the page contains sensitive data and page contents are not masked. The default privacy settings differ per tool.

Regularly, website administrators are not well aware of all personal data that third parties on their site collect [6–8]. Hence, one of our research questions was:

> How can we develop a tool for website administrators to gain more insight into where personal data entered into forms is sent?

To this end, we developed `leak-detect`, a tool that automatically interacts with a web page and observes information sent to third parties. More specifically, it detects, fills, and submits forms with a dummy email address and password, and searches for these values in outgoing

[1] For example, for Hotjar: `https://help.hotjar.com/hc/en-us/articles/360048489874-Site-Settings#data_suppression`

web requests. To enable searching for encoded information in web traffic, we also developed `value-searcher` and integrated it with `leak-detect`. `leak-detect` is intended as a tool that is easy to use for developers, with a readable output format, but also easy to use for researchers in large studies. It improves on crawlers used in previous studies by being a self-contained package including leak detection and tracker labeling, supporting shadow DOM, detecting leaks to DOM attributes [2], capturing stack traces for leaks, supporting manual interaction, and more. We used the tool to answer the following questions:

- What are the trackers that leak the most credentials from web forms, per leak timing (e.g. before or after form submission)?
- How common are unintentional password leaks to third parties, and how often are these caused by leaks to DOM attributes?
- How prominent are DOM attribute leaks, and what is the most common cause for these?

We expected the answer to the first question to be in line with recent similar studies [6, 9], where AtData and Facebook ranked high for leaks before and after submission respectively. Regarding the second and third question, we know that previous studies found multiple password leaks [6, 7], with the study from Senol et al. finding 49 leaks among 100K websites even before submission. However, we did not know if in our crawl the number of unintentional password leaks per website would be smaller because of fixes deployed by trackers, or larger because of the growing amount of tracking scripts and because in our study we also submit the form. Senol et al. mention that most of the leaks were caused by a session replay script from Yandex, and almost all affected sites were built using the React JavaScript framework. We expected that these password leaks might be due to DOM attribute leaks caused by React.

To answer these questions, we crawled a set of 30K web pages with login forms to check for email addresses and passwords being sent to third parties, and will present the results of the crawl in this thesis. We found 2 071 pages with values leaking to tracking parties, some even before form submission, and including 15 unintentional password leaks to trackers and a malicious domain.

[2] If a password leaks to a DOM attribute, this means it is written to a portion of the page markup that may be collected by a tracker script, causing an unintentional password leak, see Chapter 2

Chapter 2

Preliminaries

First, we will introduce some relevant web concepts regarding page structure, data collection, and domains.

2.1 Page structure

At the basis of a web page lies an HTML document. When parsed by the browser, this produces a tree of elements, called the DOM (Document Object Model). Some example elements would be an input field or a paragraph of text. Each element can have associated attributes, such as a default value or a color. Dynamic content on websites is enabled by JavaScript. With JavaScript, developers can use the DOM API to list, add, remove, modify, or observe elements and their attributes and input values. A relatively new concept is shadow DOM [10]. It can be used to create a 'shadow root' on an element, which can contain other elements, similar to a regular element. However, these child elements will not show up in the list of children of the element with shadow root. Also in other aspects the elements inside the shadow root are more isolated from regular child elements. For example, style sheets applied to the rest of the page will not automatically be applied to these elements.

2.2 Data collection

A page may use multiple JavaScript files, including files from third parties. Tracking scripts included on a web page usually have full access to the page, exactly like first-party scripts. This means they can access the full DOM tree and values inside input elements. To a certain extent, this is required for these scripts to operate. However, it also means that they can (accidentally) capture the value of a field containing personal or sensitive data such as a password, street address, or credit card number.

These script can send web requests with data to tracking servers. They can do this in the form of HTTP requests with the `XMLHttpRequest` (XHR) API or the more modern `fetch` API, or by loading a hidden dummy image or script with the data to transmit in the URL of the image. In general, the data to transmit can be put into the URL, the body (with XHR or `fetch`), or a header. Headers usually transmit metadata such as the type of data inside the body and cookies that have been set for the request domain. Tracking scripts can also use the beacon API to transmit a small amount of data in the body of a request, using the `navigator.sendBeacon` function [11]. Contrary to requests with `XMLHttpRequest` and `fetch`, the response of a beacon request is discarded and the request is guaranteed to be sent even if the page is closed, which is useful for tracking. Scripts that want to send multiple packets of data over the same connection can use the WebSockets API, which sets up a two-way communication channel between the browser and the server. Lastly, a website may use the `ping` attribute on a hyperlink element to easily track which links a user clicks. This attribute can contain a list of URLs to notify when the user clicks the link. A number of these requests may include a `Referer`[sic] header, which includes the URL or domain of the page from which the request originated. Information in requests is often encoded, for example, inside a JSON structure, a URL parameter, or using some compression algorithm.

2.3 Tracking identifiers

An easy way for trackers to link multiple events to the same user is to use cookies. Tracking cookies are unique identifiers stored in the browser and transmitted with each request to the tracking server in the `Cookie` header. Trackers can also use fingerprinting, where they try to uniquely identify each user by features such as their screen size, browser version, operating system, installed fonts, etc. This is sometimes called stateless tracking, as opposed to stateful cookie-based tracking [12].

Some trackers use an email address that the user enters into a form to identify the user [9]. This allows them to link different sessions and track users across devices. Regularly, they collect a hash of the email address, instead of the email address itself. A cryptographic hash is a fixed-length byte string, generated from an input string of arbitrary length, such as an email address. It must not be easily reversible, so it should be hard to determine the value that was hashed from just seeing the hash. It should also be hard to find two strings with the same hash. However, email addresses are usually fairly predictable and sometimes public, so finding the email address corresponding to a given hash is not that hard [13]. Like plain email addresses, hashes allow different tracking parties to exchange data about the same user.

2.4 Minified script bundles and source maps

Multiple JavaScript files including frameworks and libraries may be combined into one minified script bundle, which usually means that all code of multiple scripts including libraries was put on one line, where variable and function names are shortened to a couple of characters to reduce file size. To find the original source file and location corresponding to a location in the minified script, and thus still enable debugging, one can use source maps. Source maps are files that link locations in a minified script to locations in the original file(s). These files are mostly used by the developers of the website when debugging, but they may also be hosted by the server along with the minified files.

2.5 URLs and domain names

Resources on the web such as web pages or endpoints of tracking servers are identified by URLs. A URL can be split into components such as scheme, host, and path. The scheme, also called protocol, is usually `https` or `http`, or for WebSockets `wss` or `ws`. The variants suffixed with ‘s’ are secure versions featuring encryption and authentication. The host can be an IP address or domain, such as `www.tyranidslair.blogspot.co.uk`. The top-level domain (TLD) of this domain is `uk`. However, one cannot just register names directly under `uk`, only under `co.uk`. Hence, we call `co.uk` a public suffix. A list of public suffixes is available online [14]. This list also contains `blogspot.co.uk`, which means that this is the full public suffix of `www.tyranidslair.blogspot.co.uk`. While `co.uk` is part of the ICANN (Internet Corporation for Assigned Names and Numbers) section of the public suffix list, `blogspot.co.uk` is part of the private section. This is because each domain under the public suffix is owned by a different person. The public suffix is used by browsers for some security policies [15], and is sometimes also called the effective top-level domain. We call `tyranidslair.blogspot.co.uk` the registrable domain, or the effective top-level domain plus one, eTLD+1. Unless specified otherwise, in this study we will use the eTLD+1 of a domain, as it should be the most general domain to identify an entity.

Chapter 3

Related work

Multiple studies related to web tracking and the leakage of PII have been conducted before. We mention these in chronological order, along with relevant information, such as, where applicable, the list of websites crawled, the browser and method to control it, methods to capture requests and web API calls, followed hyperlinks, filled fields, if the form was submitted, how leaks were detected, and how third parties and trackers were labeled. We do not discuss results here, as we will do that in Chapter 8.

In 2015, Starov et al. [16] investigated leakage of email addresses to third parties through contact forms on the Alexa top 100K websites [17]. To find the contact forms, their crawler searches for hyperlinks containing the word 'contact'. It then detects and fills contact forms, where it tries to deduce the type of each field to decide what kind of structure the data to be filled should have. After filling, it submits the form and tries to determine if this was successful by looking at the text of the page that the browser navigated to. To detect leaked email addresses, they perform three crawls per form, where the first two use the same email address and the third uses a different address. They then look at query parameters, headers, and request body, and if a parameter that had the same value for the first two crawls gets a different value in the third crawl, it is flagged as potentially leaking and inspected manually later. Third parties are identified by looking at their eTLD+1, but they also check if the IP address is in the same Autonomous System as the IP address of the visited website, to prevent CNAME cloaking [18]. For the crawl they used PhantomJS [19], which is a headless WebKit browser with JavaScript support. PhantomJS is controlled using GhostDriver [20], which is a PhantomJS backend for a precursor of the WebDriver protocol. PhantomJS does not support all web technologies normal browsers do, such as WebGL and HTML5 video, so it is a less realistic testing environment, which may influence the behavior of trackers. Since 2018, it not actively developed anymore. In the study, all traffic is routed through the BrowserMob proxy [21] to detect leakage.

For their 2016 web tracking study, Englehardt and Narayanan crawled the Alexa top 1M websites to investigate the prominence of trackers on these pages and the behavior of these trackers, including their use of tracking cookies and fingerprinting. Some of the conducted crawls investigate stateful tracking using cookies by not clearing the cookies between page visits. To recognize trackers on a certain web page, they use the EasyList and EasyPrivacy tracking protection lists [22]. The crawler does not interact with the page. To perform the crawl, they created the OpenWPM framework [12], which uses Firefox controlled by the Selenium WebDriver [23]. OpenWPM is still actively developed as of 2022. They opted for a full-fledged browser instead of PhantomJS to provide a more realistic environment and to be able to test with all web APIs that a tracker may use for fingerprinting. Additionally, this allows for evaluating browser extensions such as tracker blockers. Firefox is run in headful mode, but the window is rendered into a virtual frame buffer. To protect against crashes and hangs, they run multiple processes and a central controller handling retries in case of crashes. mitmproxy [24] is used to capture web traffic. They built upon the now discontinued FourthParty Firefox extension [25] to log access to certain JavaScript APIs.

In 2020, Chatzimpyrros et al. quantified leakage of PII from registration web forms [26]. They crawl the Alexa top 200K websites and follow links to registration pages by searching for keywords. They then search for a registration form on these pages and fill the form fields with name, email address, ZIP code, and credit card info. The type of each field is determined by looking at keywords in DOM attributes. Relevant keywords for each field type were based on those collected in a small crawl of 500 websites and translated to multiple languages using WordNet [27]. The form is abandoned when an unknown field or Captcha is found. In any case, forms were not submitted in the crawl. They then searched for leaks of the PII that they filled to third parties. More specifically, they only searched the request URL for values and encodings or hashes of these, up to 3 layers deep. To confirm the leaks of encoded or hashed values, they also decoded parts of the URL to search for the value. Third parties are recognized by looking at the 'domain', but it is unclear if this means just the eTLD+1 or the full domain name. They use a Firefox browser controlled by Selenium for the crawl, with the BrowserMob proxy to intercept requests.

For their 2020 study, Acar et al. built upon OpenWPM to research multiple ways of data leaks to third parties on websites [7]. One of these ways, which is relevant for our study, is whole-DOM exfiltration, the practice of a third party sending the entire page contents over to their own server. Usually, this is done by session recording services to be able to accurately display the page when replaying the recording. If the page then contains personal information like the name or email address of the user, or maybe a credit card number entered into a form by the user, this information may be sent over to the third party. Note that normally fields values cannot be seen directly in the DOM tree, but some websites copy field values to an attribute on the element, which trackers can then collect [28]. This may also be done by web frameworks, without the knowledge of the developer of the website. For the study, Acar et al. inserted an extra element containing some made-up name into the DOM. They examined the Alexa top 15K sites plus a sample of 15K sites from the top 15K–100K and 20K from the top 100K–1M, totaling 50K sites. On each website the crawler also visits 5 other linked pages. To detect leaks of the inserted name in encoded form, they use a method based upon one developed by Englehardt et al. [29]. The method tries to recursively decode substrings of the web request to find the value sought after, or an encoding of it (e.g. a hash), where decoding and encoding are limited to a certain depth. However, some scripts split the page over multiple requests. To still detect DOM exfiltration by these scripts, they first detect pages with scripts generating requests with a size larger than the compressed size of the page. They then insert a 200 KB chunk of data into these pages and measure the difference in request size. If it increased by a similar size, the initiating script was flagged as leaking.

Dao and Fukuda looked at leakage of PII from sign-up forms in 2021 [9]. Using a Firefox browser, they manually visited 404 shopping sites from the Tranco top 10K websites [30]. They then navigated to a sign-up page and filled fields like username, real name, phone, email address, date of birth, gender, and postal address. After that, they submitted the form. This manual approach gave them the advantage that they could evade bot detection. Additionally, they could click account creation confirmation links that they received via email. They then logged in to the websites via another browser and visited a product page. Finally, they searched captured requests to third parties for leaked values and encodings or hashes of these, up to 3 levels deep, but it seems that they do not try to decode parts of the request, meaning that they would miss Base64-encoded and/or compressed structures (e.g. JSON) containing a leaked value. They distinguish third parties by looking at the eTLD+1, but they also look at CNAME records and match the values with cloaking block lists from NextDNS [31], AdGuard [32], and their previous study [33].

In 2022, Kats et al. [34] investigated leaks of search terms to third parties when using internal website search functionality on the Tranco top 1M websites. To identify third-party domains, they just look at if the eTLD+1 of two domains is different. Search fields are detected by looking at if their attributes contain the term 'search', but also translations of the term into languages identified by the HTML `lang` attribute of the page, using translations of Firefox's interface [35]. These are then filled and submitted by pressing the enter key. Their crawler also looks at embedded search functionality specified via various site search description schemas. They search URL, `Referer` header, and body for requests made for the entered search term. For leaks via URL query parameters, they try to exclude legitimate search providers by looking at the parameter name where the search term occurs (e.g. `q` or `query` is probably intended) and requests containing just a substring of the search term, which may indicate auto-complete suggestions by a search provider. To find leaks in requests, they did also decode Base64-encoded requests. More complex requests they tried to decode using Ciphey [36]. For the crawler, they use headless Chromium controlled by Puppeteer [37] via the Chrome DevTools Protocol [38].

In the same year, Senol et al. published a similar study to ours, measuring email and password leakage to third parties before form submission on the Tranco top 100K websites [6]. They first try to find form fields on the landing page. If it has no fields, their crawler tries to find hyperlinks to login or registration pages and clicks up to 10. As soon as it encounters a page with a form, it stops. Because email fields do not always have their type attribute set to 'email', it uses a pre-trained Mozilla Fathom model [39] to also detect email fields with a type of 'text'. Fathom is a "supervised-learning system for recognizing parts of web pages" [40], which is also used in the Firefox browser [1]. It then fills detected email and password fields and observes web requests made, without submitting the form. Fields inside shadow DOM are not filled. To detect leaks, they used an improved version of the method used in the 2020 study from Acar et al. [7]. In some crawls they also simulated the "show password" feature provided by some websites and browser extensions by changing the type of the password field to 'text', to observe if this would cause additional leaks. In some crawls they also tried to interact with Cookie Management Prompts using Consent-O-Matic [41], either accepting or rejecting cookies. To determine which domains are third parties on a website, they use the eTLD+1 and the DuckDuckGo Tracker Radar entity map [42], which groups multiple domains owned by the same company. For labeling tracker domains they consider any domain flagged by one of Disconnect, WhoTracks.me [43], DuckDuckGo [44], or by one of uBlock Origin's [45] default block lists, including EasyList, EasyPrivacy, and Peter Lowe's ad and tracking server list [46]. For leaking third parties not flagged as tracker they manually decided if these were actually trackers. They did not take into account CNAME cloaking. Their crawler is based on DuckDuckGo's Tracker Radar Collector [47], which uses a headless Chromium browser under the hood, controlled by Puppeteer.

[1] For example, for detecting 'new password' fields: `https://searchfox.org/mozilla-central/source/toolkit/components/passwordmgr/NewPasswordModel.jsm`

Chapter 4

`leak-detect`

We developed `leak-detect` [1] to be a standalone tool that website administrators can use to check if a page on their website leaks credentials of a user to a third party, and to see if passwords are leaked to DOM attributes, which may indirectly cause leaks if third-party scripts read these. Besides that, `leak-detect` can be used by researchers like us for large-scale crawls to determine the most prominent leaking trackers, causes of attribute leaks, encodings of leaked values, the prominence of password leaks, etc.

4.1 Requirements

To accomplish these goals, `leak-detect` had to satisfy a number of functionality requirements. It should be able to automatically find, fill, and submit forms with email address or username and password fields. It should capture web requests and detect possibly-encoded values that it filled in to the fields. It should also detect leaks of the password to DOM attributes and should record script access to the input field values ('value sniffs').

While developing, we had two use cases in mind: that of the website administrator and that of the researcher. For the website administrator, the tool should be easy to set up. We focus on making it usable for someone who has some programming knowledge, but a more user-friendly wrapper could be developed in the future. For the administrator, it would be practical to have a standalone tool. That is, crawling and request leak detection should be combined into the same program. Besides that, the output should be human-readable. Lastly, for debugging and locating forms that are hard to find or fill for the crawler, there should be a manual mode where the website administrator can interact with the browser. For the researcher, including for our own 30K-website crawl, there needs to be a batch crawl option, such that a list of websites can be crawled, preferably with some amount of parallelism. This crawl should produce detailed output in machine-readable format. An error that occurs during crawling of a website should not halt the full batch crawl. If a batch crawl is somehow still interrupted, it should be possible to resume the crawl without losing all progress. For the researcher, it is also useful to be able to save the configuration and versions of the tool and of other components involved to make the crawl easily reproducible.

[1] `https://github.com/stevenwdv/leak-detect`

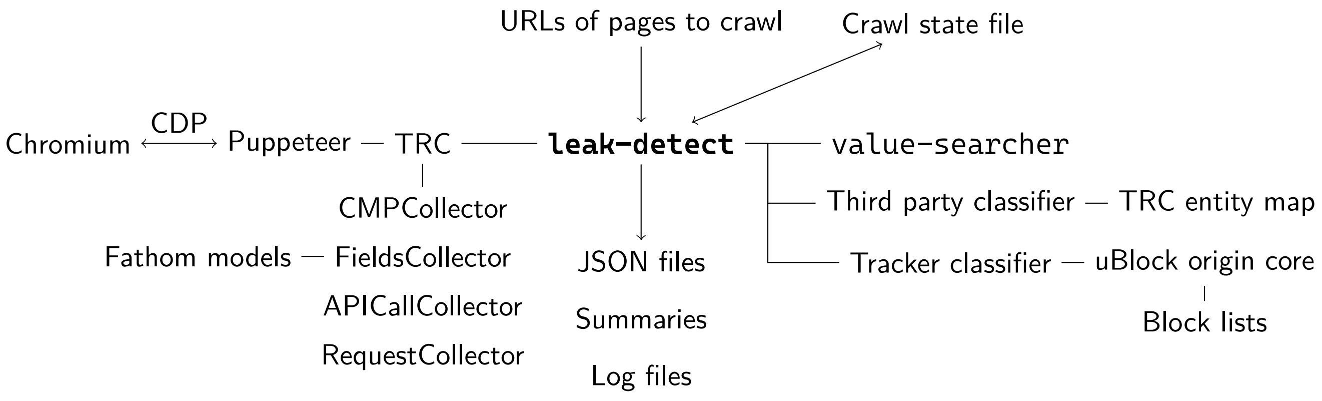

Figure 4.1: `leak-detect` structure

4.2 Crawler behavior and implementation

`leak-detect` is implemented as a JavaScript Node.js package. It uses a headless Chromium browser controlled via Puppeteer. The package also uses DuckDuckGo Tracker Radar Collector (TRC) on top of Puppeteer to provide request collection and the infrastructure for our page interaction code. More precisely, it uses a fork of TRC that we created to add some functionality, fix some issues, and update Puppeteer [2]. We chose TRC because it had been successfully used by Senol et al. in their study [6]. Besides that, we were told that Selenium, an alternative to Puppeteer, crashed regularly, and it would require us to write an equivalent of TRC for Selenium. Another alternative, OpenWPM, is more complicated, not cross-platform, and uses Firefox instead of Chrome, while most users in the real world use Chrome. In some cases, adjustments to Puppeteer itself or fixes for bugs in Puppeteer or Chromium were needed, in which case we submitted an issue to the project [3].

In short, the crawler visits a URL, fills username and password fields on the page, submits the corresponding form, and may follow login/registration links to other pages and fill forms there. Meanwhile, it listens for web requests leaking the filled username and password, sniffs of input field values, and leaks of the password to DOM attributes. After the crawl is complete, it gives a report containing information on leaking parties. Figure 4.1 gives an overview of the components of the crawler.

Finding and filling fields

Email and username fields are detected through existing Mozilla Fathom models [39, 48], which are evaluated in the page context. The email field detector is currently used in the Firefox Relay add-on, it is necessary because email fields cannot always be detected by their `type` attribute, as email fields on some websites have a type of `text` instead of `email`. From now on, we will use the terms 'email' and 'username' interchangeably. It can also find fields inside frames or shadow DOM. Frames with a different eTLD+1 are skipped by default, because we think that they have a high chance of not containing a login form for the top website. If the URL that we tried to visit redirects to a different domain, we also skip crawling the website. For detecting email and username fields within shadow DOM, we use our fork of Fathom,

[2] `https://github.com/stevenwdv/tracker-radar-collector`

[3] Puppeteer: `https://github.com/puppeteer/puppeteer/issues?q=author:stevenwdv -reason:"not planned"`,
Chromium: `https://bugs.chromium.org/u/1134224695/hotlists/devtools?can=1`

fathom-shadow [4]. We wanted to be able to look into shadow DOM, because it is an emerging technology and previous research skipped shadow DOM when finding forms. If a page tries to create a closed shadow root, the crawler changes the call to create an open shadow root instead, such that we can still look into it. We try to exclude hidden fields by using a function from Fathom, which checks if the element has a nonzero size, is not above or to the left of the screen, and is not transparent, among other things. It also looks at the ancestor elements. Depending on the configuration, the crawler may fill just one or multiple forms. We group fields in the same form and fill all of them, or fill all fields that are not inside any form. In the newest version of `leak-detect`, we give preference to forms containing password fields when it is configured to fill just one form, to prevent filling a newsletter form when a login form is available, for example.

Capturing web requests and value sniffs

Requests are continuously logged by the request collector from TRC. This collector was modified by Senol et al. [6] to add WebSocket, request header, and POST data (request body) support, together with some other improvements. On top of that, we added support to capture the full request initiator call stack, and fixed some small issues. Field value sniffs are logged by our modified API call collector, which listens for access to the `value` property. We modified this collector to add the ability to capture custom data when an API is accessed and to capture the full call stack. We also added support to more easily customize the observed APIs and to pause the debugger in Chromium's developer tools when an API is accessed.

Detecting password-to-attribute leaks

In Chapter 3 we introduced leaks to DOM attributes, as mentioned by Englehardt et al. [28]. We wanted to detect leaks of passwords to attributes before these are actually collected by a tracker, such that leaks to trackers cannot be introduced if a new tracking script that collects attributes is later added to the site.

We listen for leaks of passwords to DOM attributes using two methods. First, we use a `MutationObserver` to observe the changes to attributes and added elements for the DOM tree of the frame containing the password field, including shadow DOM. This method enables listening for changes to attributes in the whole tree at once. However, it does not allow for capturing the stack trace of the code that changed the attribute. That is why we implemented a second method that instructs the `DOMDebugger` of CDP to pause whenever an attribute is changed. This allows us to capture the stack trace and resume afterwards, but it requires us to specify the elements of which the attributes should be observed; it does not support observing the whole tree at once. Hence, we observe the elements that are most likely to have the passwords written to an attribute, based on our pilot crawl. First, we observe the password input to be filled. Besides that, we observe changes to attributes of other input fields in within the same form element, if any, because we saw that a number of sites leaked the password to other input elements. However, the stack trace often gives locations inside a minified script bundle, which usually means that the original source file and function names are gone. This means that we would not be able to determine if a JavaScript library caused the leak. To still try to recover the original script name and location, we go through each script in the leak stack trace and check if it has an associated source map. If it has one, we use that to map the position in the minified script back to the original script. When determining if an attribute

[4] https://github.com/stevenwdv/fathom-shadow

leaks, we check if it contains the password or a JSON-encoded version of the password. Newer versions of `leak-detect` also detect URI-encoded passwords.

Submitting forms

After filling a form, the crawler tries to submit the form by pressing enter inside a form field while still listening for leaks. Each form is only submitted once. By default, it also inserts and clicks a dummy button before submission to see if some scripts just mistake any button for a form submit button. Facebook was previously observed collecting hashed email addresses from forms after any button was clicked [6].

If configured to fill multiple forms, the crawler returns to the landing page and fills and submits the next form. Login and registration links (and buttons) to follow are selected in the same way as in the study from Senol et al. Pages with a different eTLD+1 are skipped as they are likely not login pages for the current website.

Detecting request leaks

After crawling a page, requests are searched for possibly-encoded leaked filled values using our `value-searcher` (see Chapter 5). This package is inspired by the LeakDetector used in the study from Senol et al. It is implemented in JavaScript, such that it was easy to integrate this in the crawler. We search the request body, URL, `Referer` and `Cookie` headers, and headers starting with `X-`. Other headers are excluded to reduce the searching time, and the `Referer` and `Cookie` headers should be the ones that leak most often, as they can contain the URL of the visited website and values of tracking cookies. Headers starting with `X-` are custom headers, and we wanted to discover if there were trackers using these to transmit the filled values.

Labeling third parties and trackers

The crawler also automatically labels third-party and tracker domains of scripts in request URLs and input value sniff stacks. Third parties are labeled by first looking at the corresponding entity from the entity map from DuckDuckGo's Tracker Radar [42], or at the eTLD+1 if that fails. Trackers are labeled using a number of block lists: EasyList, EasyPrivacy [22], URLHaus Malicious URL blocklist [49], Peter Lowe's Ad and tracking server list [46], and uBlock Origin filters, badware risks, privacy, and unbreak lists [45]. These are the default lists used by uBlock Origin 1.45 excluding the quick fixes and resource abuse lists. We did not use the lists to block trackers, only to classify them. When determining tracker status, we pass the full request URL, the resource type, and the URL of the page to uBlock Origin, so we look at more than just the domain.

Crawl output

After the crawl, a file containing all results in machine-readable JSON format can be saved, together with a log file and a human-readable summary. The summary contains the most important information from the JSON file and presents it as a timeline of events, like filled fields and detected leaks, with more details below that. See Appendix A for some example summaries.

Batch crawls

It is possible to perform a batch crawl of multiple URLs with a single command, by passing a file containing a list of URLs. Results are saved to separate files for each website. A number of websites may be crawled in parallel, where all crawls run in the same Node.js process, but multiple browser processes are used. We do not use multiple Node.js processes like some other studies, because the latest version of Puppeteer is stable enough that it does not cause fatal errors. Each crawl uses a fresh separate browser context, such that crawls are stateless. An error in one crawl should normally not influence other crawls, unless the crawler ran out of memory. If the crawler is interrupted during a batch crawl, it will remember which URLs it fully crawled and which it has not finished yet, so that it can continue where it left off. It also makes sure that no two crawler processes can accidentally run at the same time using the same results folder.

Cookie prompts and bot detection

DuckDuckGo Autoconsent [50], built into recent versions of TRC, is used to automatically interact with cookie management prompts (CMPs). By default, it is configured to accept all cookies. The crawler changes the user agent to be Chrome 93 instead of 'HeadlessChrome'. The newest version also sets the nonstandard `window.chrome` property, which is normally present in headed Chrome, and unsets `navigator.webdriver` and overwrites some other properties that TRC normally overwrites to evade simple bot detection. Due to an oversight, this was excluded from the version used for the main crawl.

Crawl options and additional features

Various crawl options can be specified. For example, the crawler can also be configured to change the type of a password field that it fills to 'text', to simulate a 'show password' feature and check if third-party scripts handle this properly.

It is also possible to specify a number of actions to be performed before searching and filling fields, which can be useful when crawling a website where one has to click multiple buttons to reach the form. These interactions can be recorded with the Chrome DevTools Recorder and are replayed using Puppeteer Replay [51]. There is also an option for manual headed mode, where one has the ability to manually fill forms that the crawler cannot find itself.

Chapter 5

`value-searcher`

To make the crawler into a standalone tool that could detect leaks of email addresses and passwords itself, it was practical to have the part that searches for leaks implemented in the same programming language as the crawler, i.e. JavaScript. The script used to check for leaks in the crawl from Senol et al. [6] was written in Python [52], but they did translate part of it to JavaScript for their LeakInspector browser add-on [53]. However, this version still lacked some compression support, had only limited lz-string support, and seemed to have some potential bugs [1], some resulting from the translation to JavaScript. Additionally, we thought it would be faster to use built-in Node.js functions for Base64 and hashing. Hence, we wrote our own `value-searcher` package [2], including a lot of unit tests and integration tests to make sure that it works correctly and can find the values that it should find.

5.1 Operation

`value-searcher` still uses the idea of recursive decoding, but instead of splitting on some delimiters, it uses regular expressions (RegEx) to find substrings to decode. For example, it can find substrings that may be Base64-encoded or URI-encoded data. The idea behind this was that it may be better in finding values in encoded substrings delimited with unrecognized characters. We chose to use recursive decoding instead of a method like the one from Starov et al. [16], because it does not require multiple crawls of the same website, and we think it is better at detecting leaks, especially in encoded structures in which parts change every request. It supports decoding the following encodings commonly used on the web: Base64, HEX, URI, JSON string, HTML entities, `application/x-www-form-urlencoded`, `multipart/form-data`, lz-string, GZIP, ZLIB, raw DEFLATE, and Brotli. For Base64, multiple dialects are supported, including the URL-safe dialect which uses `-_` instead of +/ as digit 62 and 63. It also supports searching for hashed values. Adding custom encodings is also easily possible. For encodings operating on strings, we assume the strings are encoded using UTF-8, the character encoding commonly used on the web. The searcher can be instantiated with a list of encoders and decoders and a maximum depth for decoding. Additionally, a maximum encoding depth can be specified. Encoding is mostly meant for non-reversible encoders, most notably hashes, such that it can also find these. In the LeakDetector from previous studies, the resulting encodings are called the 'precomputed pool'. The encoding depth can be set to be greater than 1 such that hashed Base64-encoded values are also found, for example. The operation

[1] For example, see `https://github.com/leaky-forms/leak-inspector/commit/ddce65c3`, `7453d419`, `https://github.com/leaky-forms/leak-inspector/pull/19`

[2] `https://github.com/stevenwdv/value-searcher`

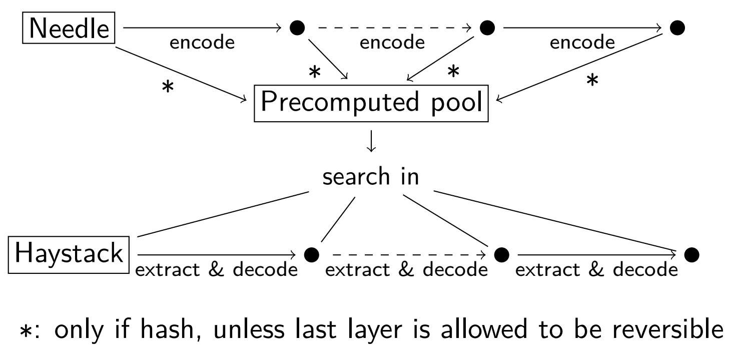

Figure 5.1: `value-searcher` operation. Values are recursively encoded/decoded with multiple encoders/decoders, this is repeated for all combinations. The 'Needle' is the value to search for, the 'Haystack' is the buffer we search in, e.g. a web request

can be seen in Figure 5.1. By default, the code makes sure that the outer encoding layer of each value in the precomputed pool is non-reversible. In practice, this means values in the precomputed pool are always hashes of some possibly-encoded value, because normally it can just decode encoded substrings using reversible encodings, such as Base64, so these do not need to be in the precomputed pool. In some cases, however, the RegEx method may not be able to properly isolate a Base64-encoded string that is prefixed by more potentially-valid Base64 characters, for example. In this case, it can be useful to allow the outer encoding to be reversible, such that the precomputed pool will include a Base64-encoded version of the value to search for, for example. This behavior is more similar to what LeakDetector does.

5.2 Comparison with LeakDetector

To evaluate the performance of `value-searcher`, we compared it with both LeakDetector versions (Python and JavaScript) [3]. First of all, we verified that `value-searcher` can find all leaks that LeakDetector could, using the dataset from the paper from Senol et al. Due to the problem of poorly-delimited Base64 substrings mentioned above, it failed to find 5 leaks out of 12 621, unless the outer encoding in the precomputed pool was allowed to be reversible. In the latter case, it successfully found all leaks, at least after some small fixes. `value-searcher` would be able to decode some cases that LeakDetector cannot decode, such as HEX-encoded values, escaped values in JSON strings, HTML-escaped values [4], and compressed data inside a `multipart/form-data` POST request.

Speed

`value-searcher` can sometimes take a while to find values, especially in large complex structures that do not contain the search value it can take a while to come to this conclusion, because it tries to decode a lot of parts first, due to the RegEx method. We did a speed comparison between `value-searcher` with reversible encoding enabled, `value-searcher` without, the Python LeakDetector.py, and leak_detector.js from LeakInspector. We ran LeakDetector.py with Python 3.10.8. For leak_detector.js, we compare a version using Node.js APIs like `Buffer` and `crypto`, and a version using pure JavaScript, both run under Node.js 19.0.0. We also tested the pure-JavaScript version in the Firefox browser, version 108.0. The results can

[3] See `https://github.com/stevenwdv/evaluate-value-searcher`
[4] LeakDetector can encode these but not decode

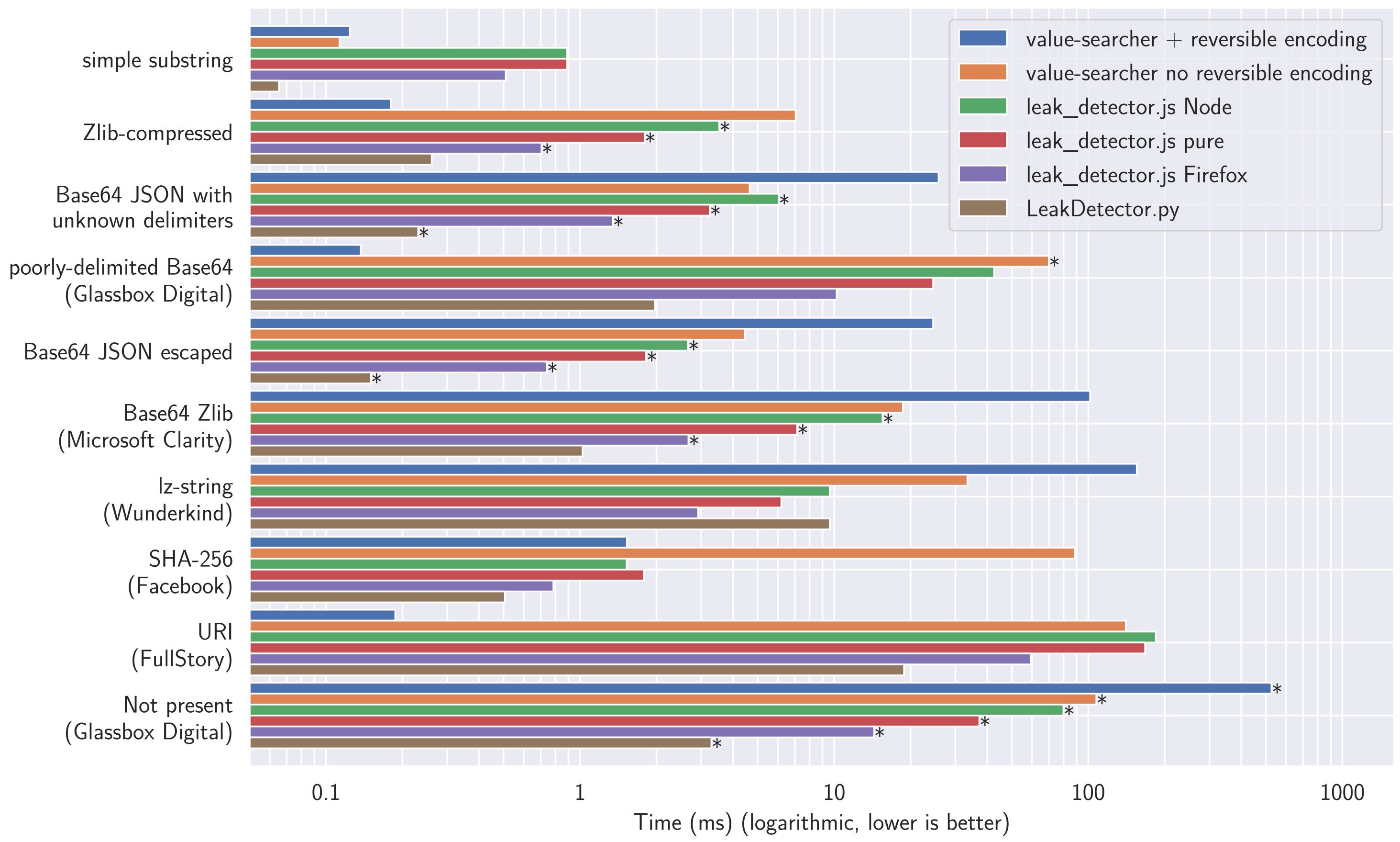

Figure 5.2: Comparison running time of `value-searcher` with LeakDetector, lower is better. $*$ = value not found. For the bottom group, the value was intentionally not present in the encoded string

be seen in Figure 5.2. The time required to build the precomputed pool is not included. It can be seen that LeakDetector.py is often the fastest. Unfortunately, `value-searcher` is often slower than the others. However, as mentioned, it can find some values that the rest cannot. Interestingly, the pure-JavaScript leak_detector.js version is often faster than the version using Node.js APIs, while usually native (C) code is faster than JavaScript. This may be because the script tries to decode a lot of small substrings, calling Node.js APIs in the process, and context switches between JavaScript and native code take time, so staying in JavaScript could be faster. When run in Firefox, which uses the SpiderMonkey JavaScript engine [54], the code is even faster than in Node.js, which uses V8 like Chromium [55]. This was also something we did not expect. It varies which variant of `value-searcher` is faster, with or without ending with a reversible encoding allowed. With the former, it may be able to find an encoded value more quickly without the need for decoding, while with the latter it does not have to search for as many encoded values.

Chapter 6

Crawl setup

We crawled a list of 30K web pages with login forms from a list composed by Roefs [56] using the the Tranco top 100K domains from a Common Crawl snapshot from November/December 2021. The Common Crawl dataset consists of a large number of static downloaded web pages [57]. Roefs used Mozilla Fathom models for login forms [48] to detect the forms in the static DOM of the page. By using this list, we did not have to find pages with login forms ourselves. We chose not to crawl their list of pages with registration forms, because registration forms often have more fields that our crawler may not be able to fill, decreasing the chances of a successful form submit. All crawled pages belonged to different domains (eTLD+1).

To speed up the crawl, we opted to not follow hyperlinks on the crawled pages, and because we crawl pages that were found to include login forms, this should generally not be necessary. Additionally, we instructed the crawler to just fill one form per page. As email address and username, we filled in `leak-detector@example.com`. As password, we entered `The--P@s5w0rd`. We think that these values are distinctive enough such that they will not randomly occur within binary data. The password was chosen such that it satisfies most common password restrictions, such as the requirement for special characters, numbers, and capital letters. We did not change the `type` attribute of password fields.

We searched for values encoded as Base64 (also URL-safe and non-padded Base64), HEX, URI, JSON string, HTML, form data (`application/x-www-form-urlencoded` and `multipart/form-data`), lz-string, GZIP, ZLIB, raw DEFLATE, Brotli, MD5, SHA-1, SHA-256, SHA-512, a salted SHA-256 version, and a custom string substitution. The last two were observed in the paper from Senol et al. [6]. We decode up to 4 layers and encode up to 2 layers for the precomputed pool. We switched on an option to also catch leaked values within poorly-delimited Base64 substrings in requests, for example, as described in Chapter 5.

We ran this crawl from a server in New York, to avoid influence from the GDPR, such as websites collecting less data or refusing to load when accessed from inside of the EU. We first ran a pilot crawl on 1K URLs to make sure that the crawler worked correctly. Based on this, we fixed some issues and made some improvements.

We ran the crawl between October 3 and October 11, 2022, using `leak-detect` versions between `028ee8d3` and `5ed68b2c`, fixing some issues and crashes that we came across along the way. These versions all use Puppeteer 18.0.5, which uses Chromium 106.0.5249.0. The block list versions we used to label trackers are listed in Table 6.1. We used DuckDuckGo's

List	Version
EasyList	Sep 22, 2022, 11:42
EasyPrivacy	Sep 22, 2022, 11:42
Peter Lowe's	Sep 17, 2022, 14:54
uBlock badware	Sep 22, 2022, 05:47
uBlock filters	Sep 21, 2022, 20:22
uBlock privacy	Sep 20, 2022, 14:30
uBlock unbreak	Sep 21, 2022, 16:57
URLHaus	Sep 22, 2022, 00:13

Table 6.1: Tracker list versions used for main crawl, in UTC

entity map version `03b5f725` [1]. We skipped crawling three URLs, because `value-searcher` went out-of-memory checking for leaks on these websites.

We later added the feature to the crawler that captures the stack for DOM leaks using `DOMDebugger`, as mentioned in Chapter 4, and we re-crawled websites where we found DOM leaks in the large crawl with this new version (`leak-detect` `3d5bc0ce`, Puppeteer 18.2.1, Chromium 107.0.5296.0) on October 14 on the same server. We observed that most DOM leaks happened through minified JavaScript bundles, so it was hard to see if it was a library causing these or original code from the website. To mitigate this, we tried to reconstruct the original script name using a source map for the script, if available. We executed a crawl with the updated crawler that uses source maps (`leak-detect` `48876782`) on November 21 using a VPN connection with an exit server in New York.

Finally, we analyzed all the output JSON files with Python to extract relevant statistics, which we will present in the next chapter.

[1] `https://github.com/duckduckgo/tracker-radar/blob/03b5f725/build-data/generated/entity_map.json`

Chapter 7

Results

The crawler was able to load 96% of websites in the main crawl. For the rest, it encountered an error while fetching the document, such as a domain name that could not be resolved or an invalid certificate, or it timed out while crawling the page. A breakdown of fatal error frequencies can be found in Figure 7.1. ‘Operation timed out’ means that the entire crawl took longer than 20 minutes and was aborted, which happened once. HTTP error status codes were not taken into account. We found fields on 74% of the websites that successfully loaded. 6.0% of otherwise successfully loaded websites timed out while loading the main page, and for 2.4% the URL was still `about:blank`, while the others at least partially loaded. In both cases, we still went ahead with the crawl, after instructing the browser to stop loading the page.

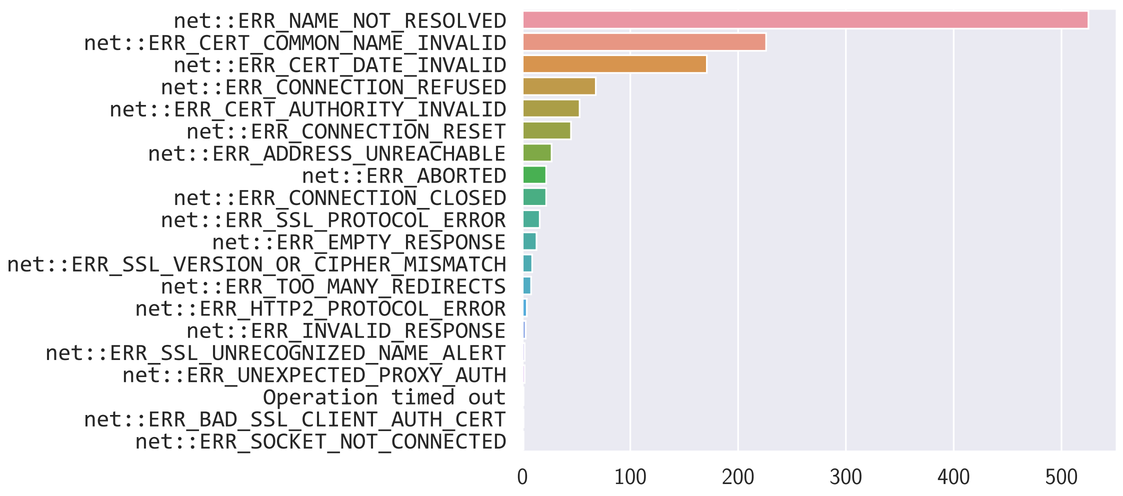

Figure 7.1: Fatal crawl errors [58]. ‘Operation timed out’ means that the entire crawl took longer than 20 minutes.

In total, we filled 21 485 email fields and 19 484 password fields. On 18 695 pages, we filled both an email and a password field, while on 2 127 pages we filled just an email field, see Figure 7.2. We found 26 sites that had fields within frames with the same eTLD+1, and 9 sites with fields inside a shadow tree. We were able to successfully submit 20 997 forms.

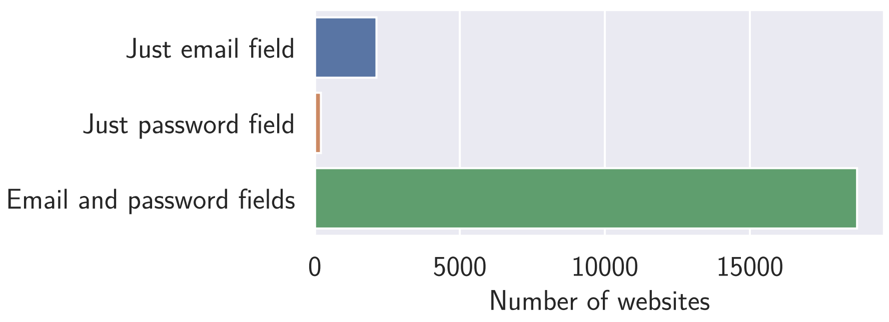

Figure 7.2: Field typed filled by the crawler for each website

7.1 Request leaks

In this section, when we count requests satisfying some criteria, we do not count two requests to the same eTLD+1 from the same website twice, unless they differ in the feature we are looking at. When we talk about trackers here, we mean third-party trackers, as identified by our crawler. Unless specified otherwise, we look at email and password leaks combined.

We found a leak to at least one third party on 63% of crawled URLs without fatal errors. Only looking at tracking parties, we found leaks on 7.2%, or 2 071 pages. In Table 7.1 you can find a breakdown of the two leak categories, email addresses (or usernames) and passwords, and the tracker status of the third party that leaked the value. We can see that most third parties receiving email addresses are trackers. Most of the time, they receive these addresses in hashed form, such that they can use them to identify a user. Passwords are usually not sent to trackers, but they are used by some third-party authentication services, and sometimes passwords may accidentally end up at tracking parties, as we will discuss in Section 7.3.

	email		**password**	
	plain	hashed	plain	hashed
tracker	946	2 177	32	0
non-tracker	442	69	108	2

Table 7.1: Number of websites with request leaks by category, encoding, and third-party tracker status

Leaking domains

In Figure 7.3 you can see that the most frequently occurring tracker domain that values are leaked to is `facebook.com`, which leaks an order of magnitude more often than the runner-up, `google-analytics.com`, even if we add leaks of `google-analytics.com`, `google.com`, and `doubleclick.net`. In Table 7.2 you can see the tracker domains and their corresponding entities per leak timing, where 'post fill' means that the value was leaked just after filling the field, before submission, and 'post submit' means after submission. We found 2 237 post submission leaks to 218 domains on 1 602 websites, and 599 post fill leaks to 79 domains on 328 websites. One can see that AtData causes most leaks directly after filling the form. Facebook collects most email addresses after form submission, followed by Google, TikTok, and AtData. We excluded the leaks which occurred just after clicking the added dummy button from this table, because practically all of these (1 597) were due to Facebook, as can be seen in Figure 7.4. We also found email address leaks on one website before we filled in the email

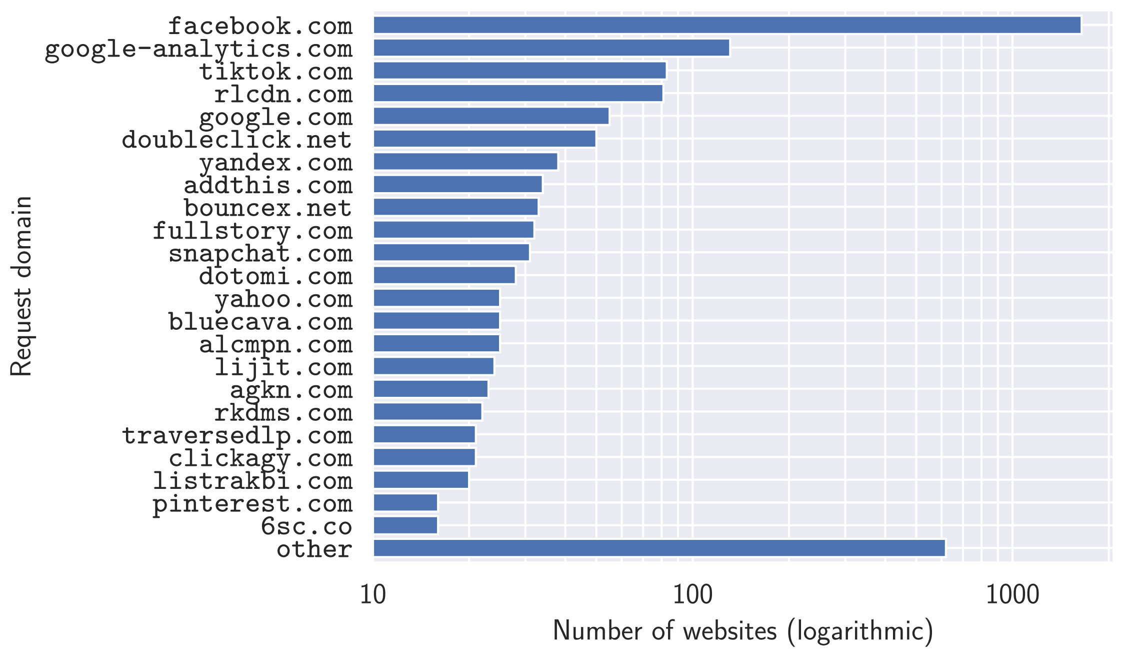

Figure 7.3: Most-leaking tracker domains with the number of websites they leak on

field on the page, which was likely caused by a tracker fingerprinting our browser.

The high position of Google Analytics is interesting, because Google does not allow using it to collect PII, such as email addresses [59]. Most of these addresses were leaked inside the `dl` parameter in the URL, which contains the document URL [60]. This means that the email was put in the URL after filling the form, and then collected by Google Analytics. To a lesser extent, values are also leaked via `el` (Event Label), `utmp` (UTM page path) [61], and some other parameters. We saw that many of these leaks only happen when entering incorrect credentials, like we did, because websites may reload the login page with the email field already filled, using a URL parameter to remember the value. Other causes include the crawler filling a field with an email address while the field is not meant for that, such as with a search field, and the crawler filling newsletter subscription forms instead of a login forms, which sometimes redirect to a page with the email address in the URL.

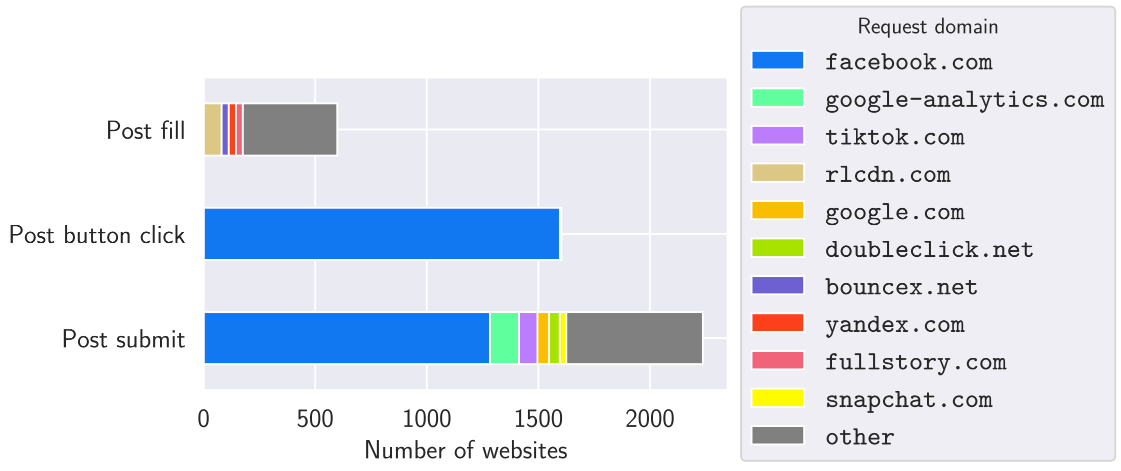

Figure 7.4: Top leaking domains by leak timing

Timing	Request domain	Entity	Website count
Post fill	`rlcdn.com`	AtData [a]	81
	`yandex.com`	Yandex	32
	`bouncex.net`	Wunderkind [b]	32
	`fullstory.com`	FullStory	31
	`addthis.com`	AddThis (part of Oracle)	23
	`agkn.com`	Neustar (part of TransUnion)	23
	`dotomi.com`	Epsilon [c]	23
	`rkdms.com`	Merkle [d]	22
	`lijit.com`	Sovrn [e]	22
	`yahoo.com`	Yahoo	22
	`clickagy.com`	Clickagy (part of ZoomInfo)	21
	`bluecava.com`	Adstra [f]	21
	`alcmpn.com`	Adstra [g]	21
	`traversedlp.com`	Traverse	21
	`listrakbi.com`	Listrak	20
	`pippio.com`	LiveRamp [h]	15
	`shop.pe`	SafeOpt [i]	12
	`salecycle.com`	SaleCycle	11
	`smarterhq.io`	Wunderkind [j]	11
	`recapture.io`	Recapture	9
Post submit	`facebook.com`	Facebook	1 283
	`google-analytics.com`	Google	130
	`tiktok.com`	TikTok	83
	`google.com`	Google	52
	`doubleclick.net`	Google [k]	48
	`snapchat.com`	Snapchat	29
	`bouncex.net`	Wunderkind	25
	`6sc.co`	6sense	16
	`pinterest.com`	Pinterest	16
	`linkedin.com`	LinkedIn	14
	`fullstory.com`	FullStory	13
	`bizible.com`	Adobe Marketo [l]	13
	`permutive.com`	Permutive	13
	`klaviyo.com`	Klaviyo	13
	`addthis.com`	AddThis (part of Oracle)	12
	`criteo.com`	Criteo	11
	`yandex.com`	Yandex	10
	`adnxs.com`	Xandr [m] (part of Microsoft)	10
	`bing.com`	Microsoft	10
	`clarity.ms`	Microsoft	9
	`zenaps.com`	Awin [n]	9

Table 7.2: Top leaking domains by leak timing. Many of these parties were acquired by larger companies in the past or were rebranded.

[a] formerly TowerData, RapLeaf
[b] formerly BounceX, Bounce Exchange
[c] formerly Dotomi
[d] formerly Rimm-Kaufman
[e] formerly Lijit
[f] formerly BlueCava
[g] formerly ALC, American List Counsel
[h] formerly Arbor, Pippio
[i] formerly Shop.pe
[j] formerly SmarterHQ
[k] formerly DoubleClick
[l] formerly Bizible
[m] formerly AppNexus
[n] formerly Affiliate Window, Zenaps

Leaks to `addthis.com`, `dotomi.com`, `lijit.com`, `bluecava.com`, `alcmpn.com`, `agkn.com`, `rkdms.com`, and `traversedlp.com` are often caused by Clickagy, which allows including multiple tracking parties at once. This can be seen in Figure 7.5, which shows initiating scripts for each domain that is being leaked to. See Figure B.1 in Appendix B for a version providing just the portion of requests per domain, without the quantity on a logarithmic axis.

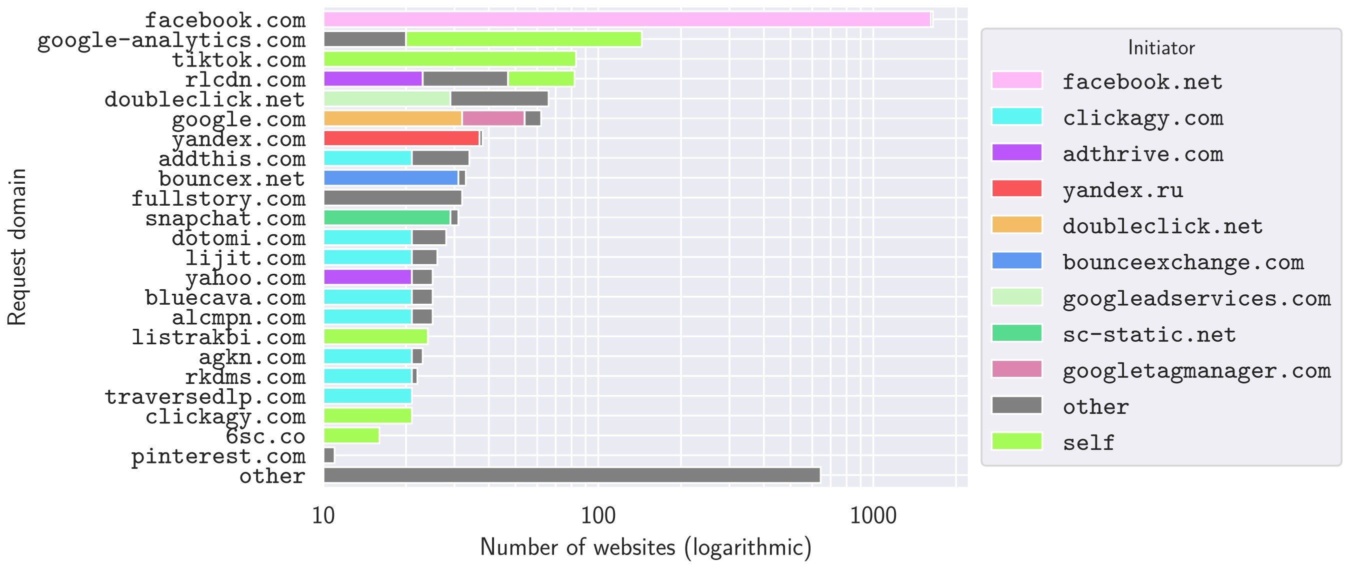

Figure 7.5: Top tracker domains to which data is leaked, including domains of scripts initiating the leaking requests to that tracker ('Initiator' legend). 'self' indicates that the script domain is the same as the tracker domain.

Most pages have just one leaking tracker, with less that one fourth of pages having two or more trackers, although couple of pages have 24 leaking trackers. See Figure 7.6.

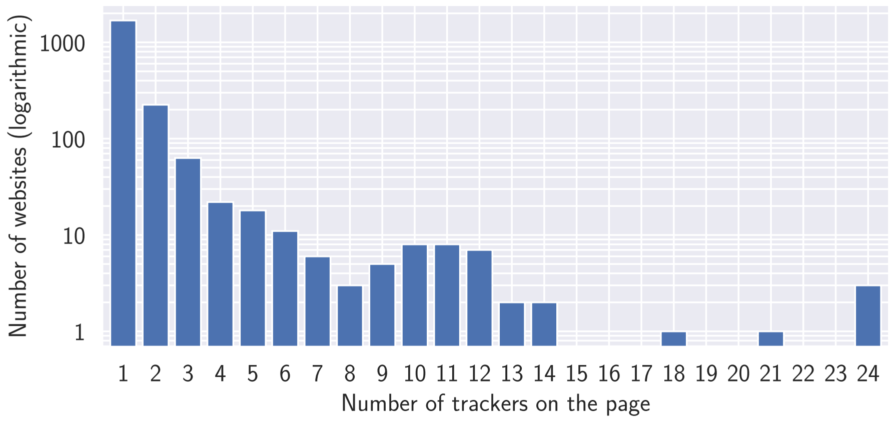

Figure 7.6: Number of leaking trackers per page

Request types and parts

In Figure 7.7 one can see in what kind of requests we found the most leaks and by which parties. We observed 1 211 leaks through the request URL and 1 211 leaks through the request body. Of these, 1 772 were from the Beacon API, most of them to Facebook. However, Facebook can use one of multiple communication methods and chooses which to use based on the browser, and it turns out that our headless browser was not recognized as a regular Chrome browser by Facebook, because we did not set the nonstandard `window.chrome` property. With this property correctly configured, it seems that Facebook prefers using images (tracking pixels) instead, so the graph in Figure 7.7 would look substantially different. We also saw 47 leaks via a request header, but remember that we only search `Referer`, `Cookie`, and `X-`… headers. From these, we saw 27 leaks in the `Referer` header, 6 in the `Cookie` header, and one in a header called `x-mbsy-url`, apparently used by Ambassador (`getambassador.com`).

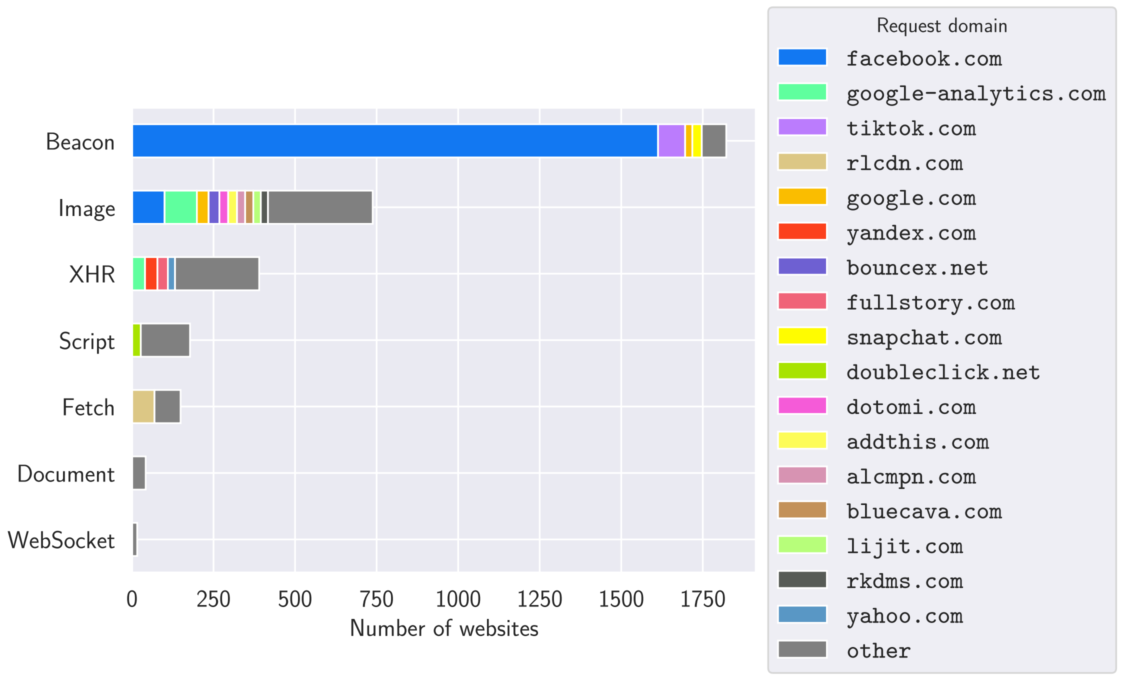

Figure 7.7: Leaking request domain for each request type.

Encodings

Figure 7.8 displays the most frequently used encodings for the leaked values. Most common are SHA-256 hashes, which we found leaked on 1 823 websites. Of these, 1 625 were collected by Facebook. MD5 hashes (336 websites), URI-encoded variants (664), and plain text (188) are also popular. Note that the first (outer) Base64 layer in the figure may not actually be present in the request, as CDP encodes binary request data with Base64. We observed just one value which was encoded with more than 3 layers, namely using 4 URI encode layers. We saw 6 websites leaking values encoded with the substitution cipher mentioned in Chapter 6.

In Figure 7.9 you can see which domains receive the data in hashed form and which do not. We see that TikTok mostly collects hashed email addresses (after submission), and that Google receives mostly unhashed addresses, which makes sense for the unintentional Google Analytics leaks. For leaks directly after filling the field, we saw in Table 7.2 that AtData, Yandex, Wunderkind, and FullStory are the biggest parties. While AtData sends email addresses in hashed form, the other three send the unhashed address.

Protocol

It is interesting how some websites still transmit credentials including passwords over insecure HTTP connections, as can be seen in Figure 7.10. The same is true to a lesser extent for non-

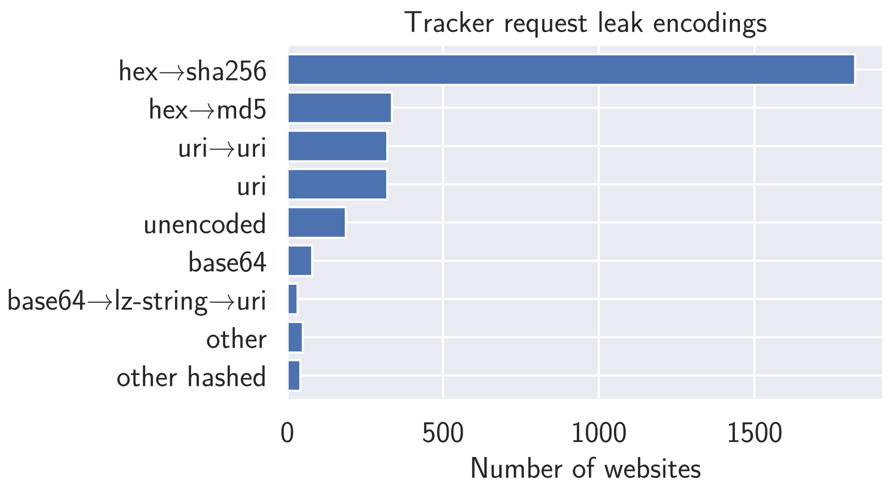

Figure 7.8: Encodings of values leaked to trackers.
Layers are separated by '→' with the outer layer first

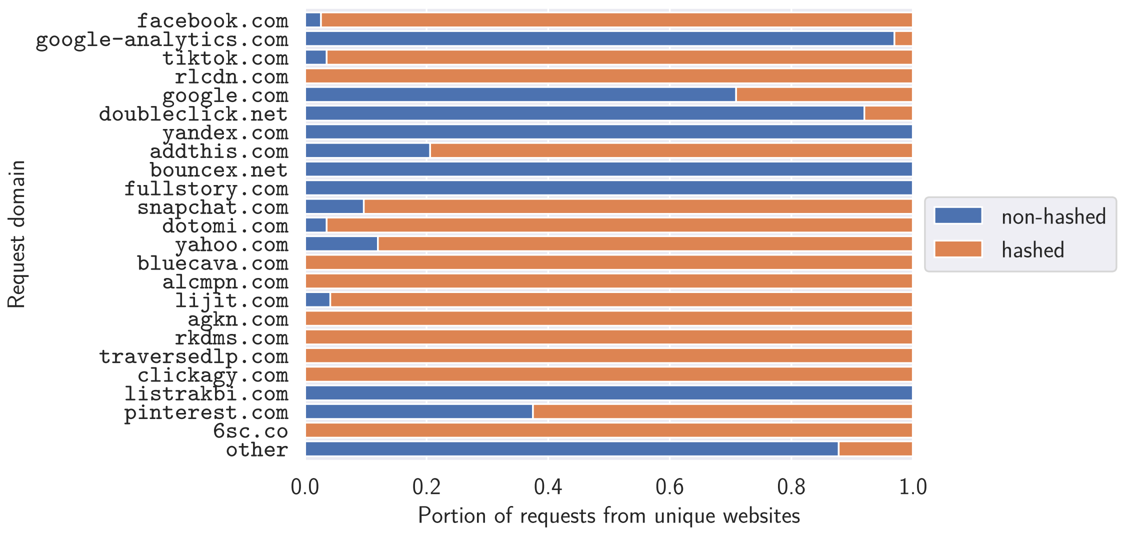

Figure 7.9: Tracker request leak domain with leaked value encoding type

tracker third parties. Also interesting is that most trackers do use secure HTTPS connections, which may be explained by the fact that browsers may block insecure HTTP traffic, which will hinder trackers using it in their operation.

7.2 DOM leaks

We observed 288 pages with DOM password leaks, that is, pages with scripts that copy the password field value to a DOM attribute. We re-crawled these with the newer `leak-detect` version to obtain source-mapped stack traces, as mentioned in Chapter 6. In this crawl, there were 269 pages where we still found DOM leaks. We got 6 fatal errors while performing the crawl (e.g. connection failures like before, but we also count HTTP errors for this crawl). We also observed 14 timeouts while loading main pages, including one where the URL was still `about:blank`.

We found 218 pages with DOM leaks which occurred directly after filling the element, and

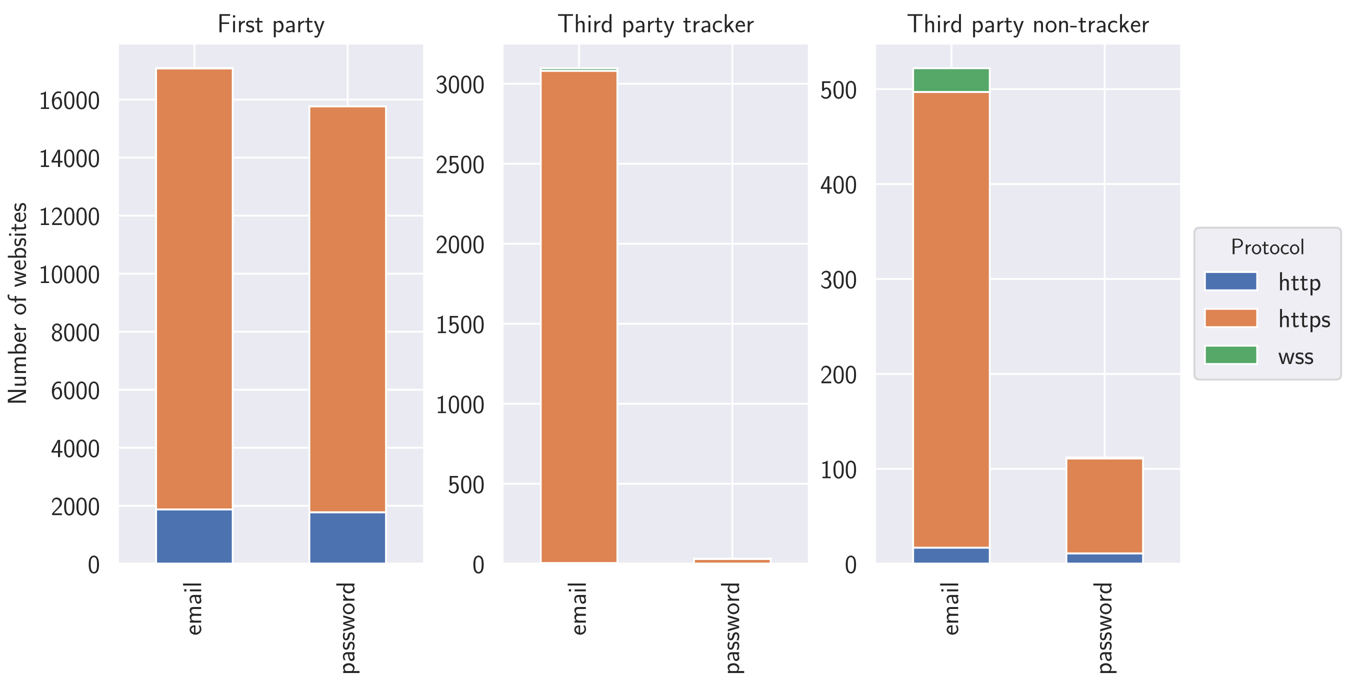

Figure 7.10: Request leak value type and protocol for first-party requests, third-party tracker requests, and third-party non-tracker requests

51 where these only happened after submitting the form. Aside from a couple of leaks, all of these were to the `value` attribute. In Figure 7.11 you can see that most leaks were to an input element that was filled by the crawler. A lower but still significant number of leaks was to some other hidden input field. These leaks were likely intentional in the design of the form. 'unknown' elements indicate elements for which we were not able to capture additional information, because the element was quickly removed or because the page navigated, for example.

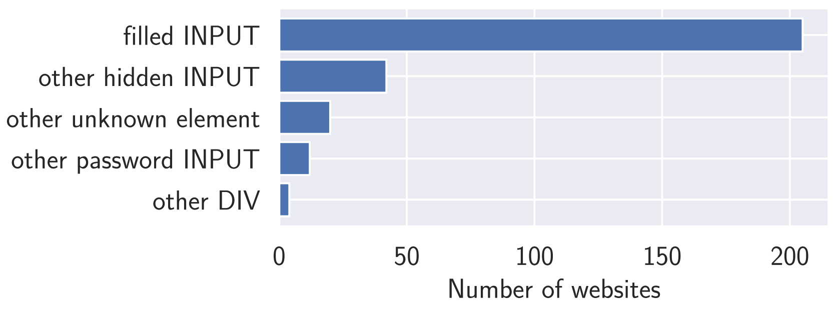

Figure 7.11: Elements with password leaks to DOM attributes

Leaking scripts

We used the stack traces, with source maps, where available, to find which scripts most commonly leaked the password to a DOM attribute. In Figure 7.12 you can see the results. On the top, you can see the script names of the script at the top of the stack, so the script that directly wrote the value to DOM. On the bottom you can see the most commonly occurring scripts anywhere in the leak stack. On the right, source maps were used, while on the left the actual script names are listed. We tried to collapse HEX strings (e.g. `framework-`

d2410d5f375f8476.js becomes framework-*.js), jQuery versions (e.g. jquery-2.1.1.min.js becomes jquery*.js), and react-dom variants (e.g. react-dom.production.min.js becomes react-dom.*.min.js). We can see that framework-*.js ranks very high in the raw script names. Most of these are minified script bundles. Using source maps we can see that most of this is actually from the react-dom [62] package, part of the React framework. The scheduler.production.min.js script we see in the most common scripts is also part of React [1]. We observed that the function of react-dom that leaks the password to DOM is called setDefaultValue [2]. It turns out that this is intended behavior, although the original reason for introducing the behavior is unclear, and it has caused multiple bugs. An issue was filed in the React repository at the end of 2017 which proposes removing this 'attribute syncing' behavior and provides some context [63]. Apparently, the maintainers also know of the security risks of copying a password to an attribute, such as analytics scripts collecting attributes and thus passwords, and CSS key logging. CSS key logging works by matching the value attribute requesting different URLs from a malicious server depending on the value, thus recovering the value by looking at the incoming requests. It is only possible if an attacker can insert a custom CSS style sheet into the page with the form. A pull request that was merged at the end of 2019 disables attribute syncing if the disableInputAttributeSyncing feature flag is enabled [64], but as of version 18.2.0 this is not the case [65] and it is not clear when this will happen. It seems that enabling React feature flags as a consumer is not possible without forking and recompiling the React repository. We also tested other popular JavaScript frameworks (Vue.js, Angular, Svelte, and Preact) and found none that also copy the value to an attribute.

7.3 Password leaks

We found a number of cases where passwords seemed to leak to third parties, but only a small part of these leaks were actually problematic password leaks, see Table 7.3. We manually went through all pages with possible leaks and visited them to determine if the leaks were actual password leaks or not. These leaks are labeled 'True positive' in the table.

There are various reasons for why a request may be wrongly labeled as a password leak. First, many domains that our crawler labeled as third parties are actually also owned by or collaborate with the first party. Second, a number of websites include login forms managed by other entities, such as an event planning service. Lastly, some websites use a third-party service for authentication. These leaks are all put under 'False positive' in Table 7.3. We observed false positives on 104 websites.

We also found 4 websites with leaks that were real, but caused by some obscure web form that a normal user would not come across unless they were given the direct URL. On 5 websites, we detected leaks, but later when we wanted to report them we could not reproduce them anymore. On one website, we observed a leak to tracking software, but we decided not to report the leak because the software was owned by the same company. For one website (labeled 'Unknown' in the table), we were not able to determine with certainty if the transmission of the password was intentional or not.

[1] It is part of the scheduler package, which is internal to React: https://www.npmjs.com/package/scheduler

[2] https://github.com/facebook/react/blob/edbfc639/packages/react-dom-bindings/src/client/ReactDOMInput.js#L415-L431

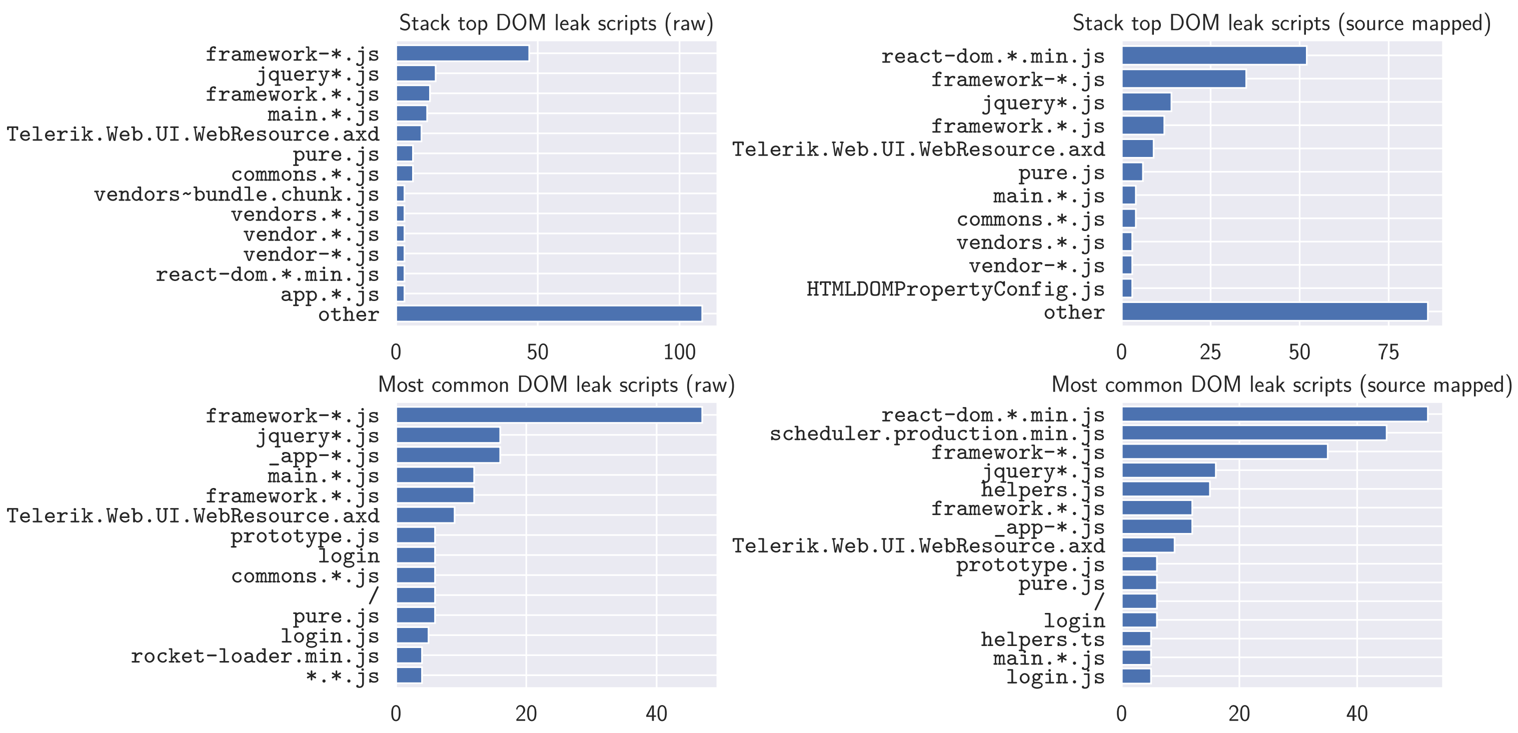

Figure 7.12: Names of scripts causing password leaks to DOM attributes, with hashes and versions stripped. Top: scripts directly writing password to attributes, bottom: scripts occurring anywhere in the stack of the writing call. Left: raw script name from URL, right: script name from source map (or raw name if no source map was available).

Actual leaks

We found 10 websites with actual password leaks using the crawler, and we manually found 5 extra websites with leaks caused by the same third party as one of the other 10 leaks. We found these extra leaks we found by searching for websites with a tracker (WebFX) that we found to be leaking, and by using a web source code search engine, which we will elaborate on below. We reported these leaks to the website administrators and to the collecting party, if they were to blame, but received just 2 messages back. However, it seems that some parties who did not reply have addressed the issue: at the time of writing, we cannot reproduce the password leaks anymore when running `leak-detect` on 4 of the 13 leaking websites which did not reply to our disclosures. One of the parties that did reply to our disclosures, Xsolla, awarded us a bug bounty. A list of password leaks with causes and more information can be found in Section C.2.

While we encountered leaks of passwords to DOM attributes on 288 pages, we only saw one instance where this caused a leak to a tracker, namely the leak from Xsolla to Inspectlet in Section C.2. This leak occurred because the password was written to the custom `data-value` attribute, which Inspectlet did not filter out. Interestingly, while the password was also written to the `value` attribute, this was not what caused the leak, as Inspectlet does not collect this attribute. Looking at the rest of the leaks, we can conclude that most tracking scripts do not collect the `value` attribute anymore, as some used to do [28].

Leak type	Websites
True positive	10
Extra discovered	5
False positive	104
Could not reproduce anymore	5
Obscure form	4
Tracking of same company	1
Unknown	1

Table 7.3: Requests to third-party domains containing password. True positives are actual password leaks discovered by our crawler, extra discovered leaks are related password leaks discovered by us, false positives were intentional transmissions. 'Could not reproduce anymore' means that the leaks were found by our crawler, but when we wanted to report them to the websites, we could not reproduce them anymore.

Leaks to WebFX

We found that a script from WebFX that collects fields values while users are typing, also sent values of password fields to `marketingcloudfx.com`, because they forgot to filter out passwords. The crawler found 2 websites where this script caused a leak. We found one more manually by looking at the crawl results, where on one site we found the tracker, but it was not leaking on the page that we crawled, only on another page. We then used a web source code search engine [66] to find more websites using the tracker, which yielded another 3 websites with password leaks caused by WebFX. WebFX has not reacted to our disclosure.

Leaks to malicious domain CrashInYou.net

We also observed a malicious inline script on the TechPowerUp forum, which reads the user's credentials on submit and transmits them to `crashinyou.net`. The leaking code can be seen in Listing 1. Using the same web source code search engine as before, we also found another site, `pixelmon.ru`, with the same malicious inline script, except it sends the credentials to a different page of `crashinyou.net`.

```
function ssubmit()
{
    var inputa = document.getElementsByName("login")[0].value;
    var inputb = document.getElementsByName("password")[0].value;
    var xhr2 = new XMLHttpRequest();
    var body2 = "user=" + encodeURIComponent(inputa)
            + "&getterr=" + encodeURIComponent(inputb);
    xhr2.open("POST", 'https://crashinyou.net/site_dti2343rewdf.php', true);
    xhr2.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
    xhr2.send(body2);
}
```

Listing 1: Leaking code on TechPowerUp forum to `crashinyou.net` (extra wrapping added)

Fortunately, it seems that `crashinyou.net` is currently down, with an error from Cloudflare that the server is offline. Looking at the Wayback Machine archive, we can see that the malicious code was put in somewhere between Jan 8, 2020 and Feb 23, 2020 [3]. We contacted TechPowerUp and they told us that the malicious code was inserted into the login page template on Feb 6, 2020, which aligns with the archived versions. They also told us that they did notice a specific Russian IP address logging in to "tons of our accounts, like they knew the password". The `crashinyou.net` domain was registered way before that, in 2017, and at first hosted a 'Minecraft checker' form (in Russian), which could apparently be used to search for accounts with compromised credentials for the game Minecraft. Later, the domain was replaced by `crashinyou.com` and later `crashinyou.me`, now offering this service for more games, and also allowing to recover IP addresses besides (possibly hashed) passwords. Apparently, users can buy keys to use the service [4]. The site seems to be run by Ukrainians, judging by the language used in the latest version and comments like "Happy New Year everyone, may all Russians die, Ukraine will win!" (translated from Ukrainian) in their Telegram channel [5].

Now, there are two options: either the `crashinyou.net` domain was bought by an unrelated new party, or it is still owned by the same group. If it is the latter, they might obtain account credentials from hacked forum websites like TechPowerUp to see if some users use the same credentials as in some video games like Minecraft. As mentioned, we found an almost identical malicious inline script on `pixelmon.ru`. This is a web forum related to Minecraft, so it makes sense to collect credentials for Minecraft accounts from here. We expect that there were more websites that contained the malicious script, also judging from URLs of similar endpoints on `crashinyou.net` captured by the Wayback Machine [6]. A timeline of events regarding CrashInYou can be found in Section C.1.

[3] `https://web.archive.org/web/20200701000000*/https://www.techpowerup.com/forums/login/`

[4] `https://telegra.ph/Gajd-CrashInYou-06-26`

[5] `https://t.me/s/crashinyou1`

[6] `https://web.archive.org/web/*/http://crashinyou.net/*`

Chapter 8

Discussion

8.1 Comparison with other studies

Regarding leaks directly after filling, AtData (`rlcdn.com`) is the largest, which is in line with the results from the US crawl of Senol et al. [6]. Other parties that ranked high in both studies were Yandex, Wunderkind, FullStory, AddThis, Neustar, Yahoo, and Listrak (see Table 7.2). Parties that leaked significantly less in our study were Taboola and AdRoll. Taboola leaked on 3 sites, but only post submit, while AdRoll leaked on 2 sites pre-submit and on 4 sites post submit. Looking at the full list of leaks of the study [1], we can see that we both found many leaks to Clickagy-related trackers AddThis, Epsilon (`dotomi.com`), and Neustar (`agkn.com`) (all exactly 133 in their study), but they found only a couple of leaks to other Clickagy-related trackers Sovrn (`lijit.com`), Adstra (`bluecava.com` and `alcmpn.com`), Merkle (`rkdms.com`), and Traverse (`traversedlp.com`), all of these on the same two websites. Note that while the table in their paper is sorted by prominence, ours is sorted by the number of websites with that tracker, not taking into account the rank of the website. Like them, we still observed Facebook mistaking any button that was clicked for a submit button, causing the email address hash to be transmitted. This also happens when accessing websites from the EU.

The study from Kats et al. lists the parties most frequently leaking search queries via request payload [34]. While this is something different than an email address, we can see that in their study Facebook also ranks high, like it does in our study after submitting the field. However, session replay company Hotjar, their runner-up, ranks much lower in our study, with just 3 leaks in total. Xandr (`adnxs.com`), Criteo, Sovrn (`lijit.com`), and Google Analytics rank relatively high in both studies.

The study of PII leakage from sign-up forms by Dao and Fukuda gives some relevant statistics for shopping sites [9]. Again, for them Facebook also ranks high. In their study, Criteo, Pinterest, and Snapchat rank much higher than in our study, although we did find some leaks to these domains. Apparently, these trackers are more common on shopping sites. AtData (`rlcdn.com`) ranks a lot lower than for us. They observed most leaks through the URL, while we observed Facebook extensively leaking via the request body using the Beacon API, but we will add a caveat in Section 8.2. Like them, we also observed that many of the leaks used SHA-256 hashing.

Acar et al. list parties that transmit the whole DOM of the page [7]. We also observed some scripts extracting at least parts of the DOM. For example, we observed a password leak where

[1] `https://homes.esat.kuleuven.be/~asenol/leaky-forms/#findings`

Yandex Metrika collects the whole DOM tree, including the value attribute of a hidden field. Yandex and FullStory both rank high for post fill leaks in our study, but the remaining parties did not rank high. It seems that these may not collect input values, or at least not anymore. We only observed a couple of leaks from SessionCam and Hotjar, and could not find any leaks to SmartLook. It may be that they do not collect the literal value anymore, but they may use equal-length masking.

Chatzimpyrros et al. discuss PII leakage from registration pages before submission, and they list the top 10 third-party domains that collect unencoded email addresses [26]. Although they claim these are trackers, it seems to us that at least some of these are benign parties like website or form builders. Of the leaking parties they found, we also found Google, AdRoll, Cloud-iQ, and HubSpot (`hsforms.com`), but we did not mark `hsforms.com` as a tracker. Looking into the leaks to `hsforms.com`, all leaks in our study were to an endpoint called `emailcheck`, which seems to perform some verification of an email address entered into a newsletter form, which is not blocked by uBlock Origin and may be necessary for the form to work.

Englehardt and Narayanan did not fill forms in their study, but only tested for the presence of trackers on a page. They found Google Analytics and other Google domains to be the ones occurring most often, followed by Facebook, Twitter, and Xandr (`adnxs.com`) [12]. We found all these parties leaking in our study, although Twitter caused leaks on just 8 pages. They also showed how 82.9% of third-party services on the top 55K sites only supported insecure HTTP, instead of HTTPS, although third parties on websites with a higher rank generally did offer HTTPS. We observed that for our dataset almost all trackers used HTTPS and the overwhelming majority of third-party non-trackers used HTTPS.

Starov et al. investigated leakage from contact forms by third parties [16]. They found a number of unintentional leaks via the `Referer` header, mostly to Google and Facebook, while we found such leaks on just 27 sites. This is caused by a change in the default policy for sending the `Referer` header: since 2017, Chrome only sends the host name for requests to third parties, instead of the full URL. However, we did observe many leaks through the document URL parameter in Google Analytics requests, which is also caused by PII being present in the page URL. We did not find many such leaks to Facebook; most data sent to Facebook was sent in hashed form in the request body. Starov et al. also list third parties that were found intentionally leaking PII before and after submission, but their ranking looks different from ours. First, they do not list Facebook here, while for us this was the top leaking party of email hashes. This may be caused either by a change in how Facebook operates or by the difference in how they detect leaks. Some other top leaking parties in their study were Adobe (`marketo.com` and `bizible.com`), Salesforce, HubSpot, MailChimp (`list-manage.com`), and AWeber. We found barely any leaks from SalesForce, but found more leaks from the rest. However, we marked most leaks from Salesforce, HubSpot, and MailChimp as not tracking-related based on the full URLs. For pre-submit leaks, they list SessionCam as the top leaking party, although they count just 5 leaks in a set of 100K websites. We observed just 2 websites where SessionCam leaked, of which one leak was before submission. For the other pre-submit domains listed we have none in common. It seems that either the number of pre-submission leaks increased since 2016, or that their leak detection method using multiple requests was less reliable.

8.2 Limitations

There are of course a number of limitations to our study. In this section we will list these, focusing on the crawl that we performed, but also mentioning general limitations of `leak-detect`.

Pages crawled

First, we look at the pages that were included in the crawl. As mentioned, the set with URLs that were crawled was composed the year before we conducted the crawl, and experienced a number of pages where the browser could not successfully establish a connection with the server, usually because the domain name could not be resolved or the certificate was invalid. In one case, the crawl of a website timed out, which means that it took longer than 20 minutes to crawl the site, because the crawler got stuck somewhere. Additionally, some pages with forms may have moved between the labeling of the dataset and the crawl, and the version of the crawler used for the main crawl did not check for HTTP error status codes like 404 Not Found, 500 Internal Server Error, or 403 Forbidden; it would just fail to find any fields on such pages.

There are also some limitations related to the creation of the dataset from Roefs. First, it uses the Common Crawl dataset, which means only static forms are included [67], so forms that would be dynamically generated with JavaScript are excluded. This unfortunately also means most forms inside shadow DOM were excluded, as shadow roots could traditionally only be created using JavaScript, and the declarative shadow DOM feature that allows creating them using just HTML is still relatively new, supported in Blink-based browsers (Chrome, Edge, Opera, ...) since mid-2021 and not supported in Safari or Firefox [68]. Forms inside frames were also skipped. This means that the number of sites with fields inside shadow DOM (and also inside frames) may be a lot higher than we observed, which we did not realize at first. The list of course also excludes forms that Fathom did not recognize. To detect the forms, Fathom searches for keywords like 'login', 'signin', and 'username', but this set is not complete and does not include non-English terms, which means that forms in a foreign language, where also parts of the source code are not in English, will not be included. For more limitations, see Roefs's thesis [56]. The list also includes URLs of obscure pages with forms, which actual users of the website will not normally come across. For our crawl, we used the login forms list, not the registration form list, so registration forms are underrepresented in our results. The login form list contained 30K websites, which is smaller than most web tracking studies used.

For some pages, we experienced a timeout while loading the main page, in which case we just tried to continue with the crawl anyway. In some cases the page was partially loaded, in other cases the URL was still `about:blank`, see Chapter 7. This means that our crawler will have missed a number of forms which were not loaded yet.

If the page when loading redirected to another domain which the crawler would label as a third party, we would not fill fields on that page. This may mean that we excluded some pages of companies that changed their name or were merged into another company. As mentioned in Chapter 6, we also skipped crawling 3 websites due to out-of-memory problems in `value-searcher`.

It is likely that we failed to crawl a number of pages because our crawler was detected as a bot and thus blocked from accessing the website. As mentioned in Chapter 4, due to an oversight, some properties giving away that our browser is controlled by a crawler were not overwritten by us. This may have lead to some more scripts detecting our crawler as such, although more advanced detection mechanisms will not be fooled by changing these properties anyway.

We only performed a crawl originating from New York, while results from the EU may look substantially different, due to the GDPR, but also because some websites block access from certain countries. This difference can be seen in the study from Senol et al. The crawl to record source-mapped DOM leak stack traces was conducted at a later time, on a different machine than the main crawl, and using a VPN to New York instead of a machine physically located there, which may have influenced the results slightly. We did observe that we failed to find 19 DOM leaks that we found before. This was at least partially caused by connection failures, timeouts, and server errors. However, we think that the different environment should not impact the stack trace of the DOM leaks and the attribute and element that is leaked to in most cases.

In our crawl, we did not make the crawler click login or registration links, but in general there are also some limitations with this feature. First, it will not find all such links, because some links will only have hints to their purpose in a language that the tool does not recognize. This means that the tool is likely to find less forms on these pages than on others when following links is turned on. It may also follow links that are not actually links to login or registration forms, but this should mostly influence just the crawl time and not leak statistics. Lastly, it will not find links inside shadow DOM.

Finding fields

If the crawler successfully ends up on a page with a login or registration form, it may still not find it. For example, maybe a button had to be clicked first to make it visible, or the form could be visible but Fathom does not recognize the email or username field. When identifying email and username fields, Fathom looks at some English keywords, which, as mentioned before, will mean we miss some fields on pages in a foreign language. It may also be that in some cases we enter an email address into a field which was not meant for that, which can cause extra leaks. On the other hand, password fields should always be detected correctly using their `type` attribute with value ‘password’. In some cases, we also saw that the crawler filled an unexpected form. For example, it filled a field to subscribe to a newsletter, instead of the login form in the center of the page. Because we instructed the crawler to fill just one form, this means that the login form will be skipped. On 2 127 websites we filled just an email or username field (see Figure 7.2), which could indicate we missed the main form, or that the form was split in multiple stages, where the username has to be submitted first. We did not fill fields inside frames with a different eTLD+1, even though in some cases a first party owns multiple domains, and they could place a form inside a frame with a different domain.

Filling fields

Before we can fill a form, all popups that cover it need to be closed. We tried to close cookie prompts, but have probably missed some that were not recognized by DuckDuckGo Autoconsent. We have also likely failed to close some popups that were not cookie-related, which means we could not fill forms on these pages.

The crawler cannot complete forms separated in multiple stages, where it would first need to fill in the email address and submit it, and only then fill in the password. For these forms, the crawler would not fill the password field. We also did not attempt to fill other field types than email address, username, or password, such as real name, street address, telephone number, and credit card information, although it would also be interesting to know how many third parties collect these.

We do not change the type of password inputs to ‘text’ to simulate using a ‘show password’ button, if the field has one, or using a browser add-on which shows passwords. It is likely that some trackers will accidentally collect the password in these cases as they would not recognize the field as a password field.

Submitting form

After filling, we try to submit the form by pressing enter inside the first recognized field. We only filled forms using a nonexistent account with an email address ending in `@example.com`. This means that we do not exactly know what would happen when we use an existing account with a valid email address. It may be that on some websites a third party only collects data on a successful login attempt, but we also saw pages that on an unsuccessful login put the email address in a parameter in the URL to pre-fill the field for the next login attempt, causing a leak that would not occur with valid credentials.

The submit action itself may also fail. For example, because not all fields were filled, fields were filled with data with an incorrect format, or a checkbox was not checked. Other websites may not accept submission by pressing enter, or they may require a user to solve a captcha, which we did not attempt. However, even if the submission failed, some third parties may still decide to collect the filled values.

Finding leaks

Meanwhile, we capture requests and search for the filled values inside these requests using `value-searcher`, but some leaks may be missed. For example, values encoded with unsupported encodings or using an unknown salt, encrypted values, values encoded using too many layers, encoded structures that were poorly delimited such that we could not extract them, or if the local part and domain of the email address were encoded separately. However, we observed just one value encoded with more than 3 layers, namely 4 URI encode layers, which is the maximum number of layers we decode. This value could also be found with 3 decode layers and 1 encode layer (for the precomputed pool), so we think that using 3 decode layers is likely enough in almost all cases. We only search for the first occurrence of a (hash of the) value, so if it was encoded using multiple methods, we just record the first one that we find. This could also mean that we think just a hash was collected while the request (possibly unintentionally) also contained the unhashed value somewhere. Values that were leaked by only sending separate keystrokes are also not detected.

Many tracking scripts batch events like mouse movements, clicks, and keystrokes into larger packets, which has two consequences for our request leak detection. Firstly, it may be that the packet happened to be sent after submitting the form, even though it would be sent before submission if we would have waited longer or if we had unfocused the page, for example. This means that some cases where we recorded a leak to be ‘post submit’, it could have been ‘post fill’. Secondly, it might mean that some trackers did not send the batched information at all, because they thought that they would be able to do that later-on, but then we closed the browser. In a more recent `leak-detect` version, we simulated unfocusing and refocusing the page after submission, because we observed that some trackers would always send the batched events after unfocusing the page. On at least one site we saw that clicking a field with a filled value caused a request with the value to a third party, but we only click the first field after filling, to then press 'Enter' in the field to submit the form, which means that we would usually miss password leaks that are caused in this way.

Regarding field value sniffs by scripts, we only recorded access to the `value` property, while there are other ways to obtain the value in a field, such as using key events or `FormData`. Another way in which we could miss some events is if the website opens a new tab or popup window. In this case, the crawler may fail to record the first couple of requests and sniffs. For some DOM leaks, we failed to capture the stack or element for the leak, for example, this may occur because the page navigated directly after the leak, because the element with the was removed from the page, or because the element was not a neighboring field (in case of the missing stack trace).

When counting leaking sites, we count distinct eTLD+1 values. However, it may be that the same party includes the same leaking tracker on different subdomains which have a different eTLD+1, which means we would count the tracker more times than we would want to.

Labeling third parties and trackers

We try to detect third parties based on the DuckDuckGo entity map and the eTLD+1, but we observed domains that were labeled as third parties, while they were actually owned by the same party as the website itself. In some cases, the domains were identical except for the country suffix (e.g. `.com` versus `.co.uk`), in other cases they were harder to recognize. On the other hand, we may also have mislabeled some third parties as first parties if CNAME cloaking was used. In Chapter 7 we only looked at third-party trackers, so we would miss tracking subdomains even if they were included in some block list.

As mentioned, trackers were detected using various block lists. While elaborate, these lists are not exhaustive and will only help us to detect known trackers, so we will mislabel some tracking parties as non-tracker, although these are likely mostly trackers that are not widely used.

Other limitations

Lastly, we mention some more general issues with the crawler. Something one essentially has to live with when building a crawler that interacts with the page is that the page can change in unexpected ways while you are interacting with it, or that fields may have different identifiers when reloading the page. We make heavy use of scripts executed in the page context, to search for fields or gather field properties, for example. These scripts may run into issues if scripts from the page overwrite certain methods or fields. For example, during our pilot crawl we encountered multiple pages where a library overwrote the `entries()` method for arrays with something that behaved differently. We fixed this instance, but it is far from the only example. Things like this lead to errors while finding fields, which means we were not able to fill all fields on such pages.

We mentioned that we did not change certain properties giving away that the browser visiting the page was not a regular Chrome browser, and that this could mean that anti-bot protection scripts could easily block our crawler. However, it also lead to Facebook using a different method to transmit data, as mentioned in Chapter 7, likely because it thought we were using a different browser. This may also have been the case for other scripts.

The crawler collects stack traces for requests, value sniffs, and DOM attribute leaks, but source maps are only used for the latter, not for request or sniff call stacks.

8.3 `leak-detect` evaluation

In this section we briefly look at how `leak-detect` satisfies the requirements from Section 4.1.

We successfully developed a tool that can find, fill, and submit forms with email address, username, and password fields. It can then search for leaks to third parties and trackers, leaks to DOM attributes, and sniffs of field values. The tool can be used by a website administrator, and to set it up one just has to install Node.js, clone the GitHub repository, run the 'install' command, and then run the tool with the URL of the website. While the interface of the tool could be improved for website administrators that do not have experience with using the command line, we think that it should be fairly easy to use for a developer. The summary feature provides the user with the most important findings of the tool, such as leaks to third parties and trackers. The statistics and timeline of events that it provides are easy to read for anyone, but it also gives some details with stack traces for web developers to help pinpoint the cause of the leak. There is also a manual headed mode using which a developer can easily see the leak being reproduced, and they can even manually fill forms that `leak-detect` failed to find or properly fill.

Researchers can use `leak-detect` for parallel web crawls, like we did, by providing a list of URLs via the batch option. Detailed results including web requests, leaks, and fields that were filled are saved in machine-readable JSON files, which can be analyzed using the favorite programming language of the researcher. In our crawl, Puppeteer itself never crashed. Any errors that did occur were handled without interrupting the batch crawl, except for 3 out-of-memory errors caused by `value-searcher`. The crawler keeps track of crawled URLs such that no significant progress is lost when the crawler is interrupted or encounters an out-of-memory error. We did not find time to implement a feature to also save the full crawl configuration for reproducibility.

8.4 Future work

`leak-detect`

There are a number of improvements that could be made to `leak-detect`. For example, it cannot automatically find forms hidden behind two or more clicks, which can be necessary when a login button opens a popup where the user first has to choose from multiple authentication methods. It could also be interesting to check for sites that log passwords to the console. Similar to with DOM leaks, some performance measurement scripts may accidentally collect the password in this way. We saw this actually happening by third party Rollbar on the Oribi website. Even though Rollbar tries to mask passwords, it fails in some instances. Related, it would be useful to detect PII being sent in the URL of a first-party request, through a GET form, for example, because this often triggers leaks to third parties. This information may already be present in the JSON output file but is not reported in the summary, and it may be that a leak in the body of the request hides the fact that a leak also occurred via the URL. To give a more complete picture of leakage, it would be very interesting to add support for more field types that the crawler can fill, such as real name, street address, and credit card information, and observe leaks for these fields. For a web developer using the tool, it would also be useful to support source maps for the request and value sniff stacks.

Other useful improvements include the ability to specify all crawler options via one configuration file, instead of just collector options via a file and the rest via the command line, like it is now. This would also make it easier to record the configuration used for a crawl for reproducibility. For this, it would also be useful to save the `leak-detect` version used and block list versions used. The interface could also be improved, especially for people without programming knowledge. One way to accomplish this would be to develop a website which internally uses `leak-detect` on which someone could enter a URL to check for leaks. For more possible `leak-detect` improvements, see the corresponding GitHub issue [2].

`value-searcher`

`value-searcher` could also be improved. For example, it caused some out-of-memory issues in our crawl for specific websites, and it could be sped up. These two issues could likely be solved at once by switching to WebAssembly. The code could be written in a compiled language like Rust and used from JavaScript. This would also make it possible to use `value-searcher` in the browser, outside of Node.js. Besides improvements in speed, some functionality could be added, such as returning all matches if there are multiple (e.g. hashed and plain), and returning intermediate encoded values and the index of the match. See the GitHub issue for more [3].

Crawl

Regarding the crawl, it may be nice to repeat it after some additions have made to the crawler, like those mentioned above or those that we added after our crawl was complete, and to then see what the difference is with our results. For example, filling forms with password fields in the presence of multiple forms, setting the `window.chrome` property, and unfocusing the page after submission. It could also be configured to change the password field type to ‘text’, to compare the amount of leaks in this case with the study from Senol et al. [6]. It would also be interesting to execute a crawl on homepages of the Tranco top 100K websites, with login/register link clicking enabled. This may lead to discovering more fields within shadow DOM and frames. It could also be configured to fill multiple forms.

8.5 Recommendations

In this section we list some recommendations for website administrators and tracking parties to prevent undesired leakage of PII.

[2] `https://github.com/stevenwdv/leak-detect/issues/11`
[3] `https://github.com/stevenwdv/value-searcher/issues/3`

Recommendations for website administrators

A good practice is to not put tracking scripts on pages of the website with forms where the user enters PII, such as login forms. Especially session replay scripts should be avoided as they might capture field values. Note that this may be trickier to accomplish if the login form is available as a popup on any page. In this case, the form could be put into a frame with a different domain, such that scripts on the main page cannot access it. If some of the functionality of these trackers is absolutely required, they should be configured to not capture the values of input fields. However, note that this may not protect against leaks via DOM attributes. If at all possible, this should be configured for all pages. Where this is not possible, masking should be enabled for the sensitive fields. To prevent scripts from accidentally leaking PII, input field values should not be copied to the DOM. This is often something done by the React JavaScript framework, and we would recommend that they remove this behavior. We also observed cases where the server of the website put the password in the HTML code. This is also something to avoid, because some scripts will exfiltrate this data. Other things to avoid are sending PII in the URL (via a GET form) and logging PII in the console, for example by logging the web request. In all these cases, scripts can capture and collect the PII. Website administrators should also regularly check for malicious scripts operating on login and registration pages, such as the one we observed on the TechPowerUp forum. Lastly, we would recommend running `leak-detect` on the pages with login or registration forms to verify that PII is not leaked, and to be alert for disclosures of password leaks from researchers like us.

Recommendations for tracking and analytics parties

We would advise tracking and analytics parties to not collect values of input fields, as they may contain PII, or to at least mask the contents. Note that not collecting values of just fields with type ‘password’ is not enough to avoid collecting passwords, as a ‘show password’ feature might change the type to ‘text’. Values may also leak indirectly through DOM attributes, so we would advise not collecting attributes on input fields, but as some attributes may affect how the page is displayed, this may be not possible for a session replay script. Currently, at least some scripts just skip the `value` attribute (see Section 7.3), but this does not catch passwords leaked through other attributes such as `data-value`. Hence, we recommend composing a list of attributes that can be safely captured, such as `style`. This list will not include attributes such as `value` and `data-value`. Another, possibly complementary, method would be to filter out attributes that contain the value of the input field that they are on. However, this could fail to detect rare cases where the value is encoded in some way.

More robust defenses

In the end, the problem is that third-party scripts included directly on a page have the same permissions as first-party scripts. They can access everything in the page and make web requests from the origin of the first party. It might be possible to better isolate these scripts by putting them inside invisible `<iframe>` elements with a different domain on the page, as such frames cannot access data in their parent page, and passing only the required data to these frames, by using `window.postMessage`. Another option in the future may be to use ShadowRealms [69], which are like JavaScript sandboxes, and to only pass the required data.

However, both of these solutions are likely unpractical in many cases. For example, session replay scripts often need access to the whole DOM, and with these isolated approaches, the DOM tree would need to be passed to these scripts anyway. It is also a question which script would pass the required data to these tracking scripts. If another script of the tracking party is used for this, it just moves the issue, and the only result is that now setting up the service on a website has become more difficult.

Chapter 9

Conclusion

We built `leak-detect`, a tool that can be used to detect leakage of email addresses, usernames, and passwords to third parties. It can automatically examine one or more websites, filling and submitting forms on these, and it searches for leaks of the filled values in web requests using our `value-searcher`. Besides that, it detects leaks of the filled password to attributes in the DOM and detects sniffs of field values by scripts. It can also classify URLs as third parties or trackers. When compared to crawlers used in previous studies such as the one from Senol et al. [6], our tool has a number of advantages. Firstly, it can detect leaks to DOM attributes, supports filling fields inside shadow DOM, and can fill username fields besides just email address fields. Furthermore, it can submit the form and detect leaks after submission, and can also detect leaks after clicking a button unrelated to submission. Useful for website administrators is that it is self-contained, with integrated leak detection and third party / tracker classification, collection of stack traces for all leak types, even using source maps for DOM leaks, and support for manual interaction in headed mode and playback of Chrome DevTools recordings. For researchers, it has an option to easily resume interrupted batch crawls. Our leak detection module `value-searcher` intentionally is a separate package, such that it can be used in other studies, possibly with some improvements.

We used `leak-detect` to examine 30K pages with forms for leaks to trackers. We found 2 071 pages with leaks to tracking parties. Of these, 1 560 pages leaked values after the form was submitted. Most common were leaks of email address hashes to Facebook. In total, we found 1 644 leaks to Facebook, of which 1 597 occurred after clicking a dummy button unrelated to submission that we added. This shows that Facebook does not have a way to distinguish a form submission button from any other button. The second-largest leaking party was Google, which leaked on 152 websites with all its domains combined, 130 leaks of which were to Google Analytics, which does not allow collection of personal data. We also observed 328 pages with pre-submission leaks, most of these (81) to AtData (`rlcdn.com`). The crawl even lead to us discovering 15 websites with password leaks, which we reported. Of these leaks, 6 have since been addressed, although we did not get a reply from 4 of these parties. We found 288 pages with passwords leaking to DOM attributes, leaks to the `value` attribute being the most common. The largest part of these leaks was caused the React framework, but most tracking scripts seem to properly filter out the `value` attribute from data they collect, because we found no password leaks caused in this way. However, we did find one password leak caused by a website leaking to a different attribute.

We think that `leak-detect` could give website administrators more insight into which parties collect what information on their website, and that in this way unintended leaks can be addressed and unsuspecting users protected. Besides that, a batch crawl like we conducted in our study can expose the most common causes for leaks, including password leaks and DOM attribute leaks, such that website administrators and web framework maintainers can address these. We hope that in this way `leak-detect` can help to make the web a little more secure for everyone.

Bibliography

[1] *Hotjar*. https://www.hotjar.com/.

[2] *Yandex Metrika*. https://metrika.yandex.ru/.

[3] *Microsoft Clarity*. https://clarity.microsoft.com/.

[4] *FullStory*. https://www.fullstory.com/.

[5] *Distribution for websites using Analytics technologies*. BuiltWith. https://trends.builtwith.com/analytics/. Look at individual techologies for statistics over time.

[6] Asuman Senol, Gunes Acar, Mathias Humbert, and Frederik Zuiderveen Borgesius. "Leaky Forms: A Study of Email and Password Exfiltration Before Form Submission." In: *31st USENIX Security Symposium (USENIX Security 22)*. 31st USENIX Security Symposium (Aug. 10–12, 2022). Boston, MA, USA: USENIX Association, Aug. 2022, pp. 1813–1830.

[7] Gunes Acar, Steven Englehardt, and Arvind Narayanan. "No boundaries: data exfiltration by third parties embedded on web pages." In: *Proceedings on Privacy Enhancing Technologies*. Vol. 2020. 4. Sciendo, Oct. 1, 2020, pp. 220–238. DOI: 10.2478/popets-2020-0070. https://petsymposium.org/popets/2020/popets-2020-0070.php.

[8] Gunes Acar, Steven Englehardt, and Arvind Narayanan. *Website operators are in the dark about privacy violations by third-party scripts*. https://freedom-to-tinker.com/2018/01/12/website-operators-are-in-the-dark-about-privacy-violations-by-third-party-scripts/.

[9] Ha Dao and Kensuke Fukuda. "Alternative to third-party cookies: investigating persistent PII leakage-based web tracking." In: *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. CoNEXT '21: The 17th International Conference on emerging Networking EXperiments and Technologies (Dec. 7–10, 2021). Virtual Event, Germany: ACM, Dec. 3, 2021, pp. 223–229. DOI: 10.1145/3485983.3494860. https://dl.acm.org/doi/10.1145/3485983.3494860.

[10] *Using shadow DOM*. https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM.

[11] *Beacon API*. https://developer.mozilla.org/en-US/docs/Web/API/Beacon_API.

[12] Steven Englehardt and Arvind Narayanan. "Online Tracking: A 1-million-site Measurement and Analysis." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16: 2016 ACM SIGSAC Conference on Computer and Communications Security (Oct. 24–28, 2016). Vienna, Austria: ACM, Oct. 24, 2016, pp. 1388–1401. DOI: 10.1145/2976749.2978313. https://dl.acm.org/doi/10.1145/2976749.2978313.

[13] Matthias Marx, Ephraim Zimmer, Tobias Mueller, Maximilian Blochberger, and Hannes Federrath. “Hashing of personally identifiable information is not sufficient.” In: *Sicherheit*. Vol. 2018. Lecture Notes in Informatics. Konstanz, Germany: Gesellschaft für Informatik e.V., Mar. 22, 2018, pp. 55–68. DOI: 10.18420/sicherheit2018_04. http://dl.gi.de/handle/20.500.12116/16294.

[14] *Public Suffix List*. Mozilla. https://publicsuffix.org/.

[15] *HTML Standard: Sites*. WHATWG. https://html.spec.whatwg.org/multipage/browsers.html#sites.

[16] Oleksii Starov, Phillipa Gill, and Nick Nikiforakis. “Are You Sure You Want to Contact Us? Quantifying the Leakage of PII via Website Contact Forms.” In: *Proceedings on Privacy Enhancing Technologies*. Vol. 2016. 1. De Gruyter Open, Jan. 1, 2016, pp. 20–33. DOI: 10.1515/popets-2015-0028. https://petsymposium.org/popets/2016/popets-2015-0028.php.

[17] *Alexa top 1M*. Alexa Internet. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip.

[18] Yana Dimova, Gunes Acar, Lukasz Olejnik, Wouter Joosen, and Tom Van Goethem. “The CNAME of the Game: Large-scale Analysis of DNS-based Tracking Evasion.” In: *Proceedings on Privacy Enhancing Technologies*. Vol. 2021. 3. Sciendo, July 1, 2021, pp. 394–412. DOI: 10.2478/popets-2021-0053. https://petsymposium.org/popets/2021/popets-2021-0053.php.

[19] *PhantomJS*. https://phantomjs.org/.

[20] *GhostDriver*. https://github.com/detro/ghostdriver.

[21] *BrowserMob Proxy*. https://bmp.lightbody.net/.

[22] *EasyList*. https://easylist.to/.

[23] *Selenium WebDriver*. https://www.selenium.dev/documentation/webdriver/.

[24] *mitmproxy*. https://mitmproxy.org/.

[25] *FourthParty*. http://fourthparty.info/.

[26] Manolis Chatzimpyrros, Konstantinos Solomos, and Sotiris Ioannidis. “You Shall Not Register! Detecting Privacy Leaks Across Registration Forms.” In: *Computer Security*. Vol. 11981. Lecture Notes in Computer Science. Cham: Springer International Publishing, Feb. 21, 2020, pp. 91–104. DOI: 10.1007/978-3-030-42051-2_7. http://link.springer.com/10.1007/978-3-030-42051-2_7.

[27] George A. Miller. “WordNet: A Lexical Database for English.” In: *Communications of the ACM* 38.11 (Nov. 1, 1995), pp. 39–41. DOI: 10.1145/219717.219748. https://wordnet.princeton.edu/.

[28] Gunes Acar, Steven Englehardt, and Arvind Narayanan. *No boundaries for credentials: New password leaks to Mixpanel and Session Replay Companies*. https://freedom-to-tinker.com/2018/02/26/no-boundaries-for-credentials-password-leaks-to-mixpanel-and-session-replay-companies/.

[29] Steven Englehardt, Jeffrey Han, and Arvind Narayanan. “I never signed up for this! Privacy implications of email tracking.” In: *Proceedings on Privacy Enhancing Technologies*. Vol. 2018. 1. De Gruyter Open, Jan. 1, 2018, pp. 109–126. DOI: 10.1515/popets-2018-0006. https://petsymposium.org/popets/2018/popets-2018-0006.php.

[30] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. "Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation." In: *Proceedings 2019 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium (Feb. 24–17, 2019). San Diego, CA, USA: Internet Society, 2019. DOI: 10.14722/ndss.2019.23386. https://tranco-list.eu/.

[31] *NextDNS CNAME Cloaking Blocklist*. NextDNS. https://github.com/nextdns/cname-cloaking-blocklist.

[32] *CNAME-cloaked trackers*. AdGuard. https://github.com/AdguardTeam/cname-trackers.

[33] Ha Dao, Johan Mazel, and Kensuke Fukuda. "CNAME Cloaking-Based Tracking on the Web: Characterization, Detection, and Protection." In: *IEEE Transactions on Network and Service Management* 18.3 (Apr. 13, 2021), pp. 3873–3888. DOI: 10.1109/TNSM.2021.3072874. https://ieeexplore.ieee.org/document/9403411.

[34] Daniel Kats, David Luz Silva, and Johann Roturier. "Who Knows I Like Jelly Beans? An Investigation Into Search Privacy." In: *Proceedings on Privacy Enhancing Technologies*. Vol. 2022. 2. Sciendo, Apr. 1, 2022, pp. 426–446. DOI: 10.2478/popets-2022-0053. https://petsymposium.org/popets/2022/popets-2022-0053.php.

[35] *Firefox translations*. Mozilla. https://pontoon.mozilla.org/projects/firefox/.

[36] *Ciphey*. https://github.com/Ciphey/Ciphey.

[37] *Puppeteer*. Google. https://pptr.dev/.

[38] *Chrome DevTools Protocol*. Google. https://chromedevtools.github.io/devtools-protocol/.

[39] *Private Relay email detector*. Mozilla. https://github.com/mozilla/fx-private-relay-add-on/blob/2022.5.18.515/src/js/other-websites/email_detector.js.

[40] *Fathom*. Mozilla. https://github.com/mozilla/fathom.

[41] *Consent-O-Matic*. Aarhus University. https://consentomatic.au.dk/.

[42] *Tracker Radar entity map*. DuckDuckGo. https://github.com/duckduckgo/tracker-radar/blob/main/build-data/generated/entity_map.json.

[43] *WhoTracks.Me*. Ghostery. https://whotracks.me/.

[44] *DuckDuckGo's Web Tracker Blocklist*. DuckDuckGo. https://github.com/duckduckgo/tracker-blocklists/tree/main/web.

[45] *uBlock Origin*. https://github.com/gorhill/uBlock.

[46] Peter Lowe. *Peter Lowe's Ad and tracking server list*. https://pgl.yoyo.org/adservers/serverlist.php.

[47] *Tracker Radar Collector*. DuckDuckGo. https://github.com/duckduckgo/tracker-radar-collector.

[48] *Login forms detector*. Mozilla. https://mozilla.github.io/fathom/zoo.html#login-forms.

[49] *URLHaus Malicious URL Blocklist*. Abuse.ch URLhaus. https://gitlab.com/malware-filter/urlhaus-filter.

[50] *Autoconsent*. DuckDuckGo. https://github.com/duckduckgo/autoconsent.

[51] *Puppeteer Replay*. Google. https://github.com/puppeteer/replay.

[52] Asuman Senol, Gunes Acar, Mathias Humbert, and Frederik Zuiderveen Borgesius. *LeakDetector, Python version*. https://github.com/leaky-forms/leaky-forms/tree/main/leak-detector.

[53] Asuman Senol, Gunes Acar, Mathias Humbert, and Frederik Zuiderveen Borgesius. *LeakDetector, JavaScript version*. https://github.com/leaky-forms/leak-inspector/tree/main/background_scripts/leak_detector.

[54] *SpiderMonkey JavaScript engine*. Mozilla. https://spidermonkey.dev/.

[55] *V8 JavaScript engine*. Google. https://v8.dev/.

[56] David Roefs. "Large-scale login and registration form identification using the Common Crawl dataset." MA thesis. Radboud University, May 2022.

[57] *Common Crawl*. https://commoncrawl.org/.

[58] *Chromium net_error_list.h*. Google. https://source.chromium.org/chromium/chromium/src/+/main:net/base/net_error_list.h.

[59] *Google Analytics Terms of Service*. https://marketingplatform.google.com/about/analytics/terms/us/ (visited on 01/24/2023).

[60] *Google Analytics Measurement Protocol Parameter Reference*. https://developers.google.com/analytics/devguides/collection/protocol/v1/parameters.

[61] *Google Analytics and Matomo tracking in depth*. NoGA. https://nogadata.nl/2020/06/18/ga-and-matomo-tracking.html.

[62] *react-dom*. Meta. https://www.npmjs.com/package/react-dom.

[63] *React issue: Stop syncing value attribute for controlled inputs*. https://github.com/facebook/react/issues/11896.

[64] *React pull request: Stop syncing the value attribute on inputs (behind a feature flag)*. https://github.com/facebook/react/pull/13526.

[65] *React feature flags*. https://github.com/facebook/react/blob/v18.2.0/packages/shared/ReactFeatureFlags.js.

[66] *NerdyData*. https://www.nerdydata.com/.

[67] *Common Crawl: WARC Format*. https://commoncrawl.org/the-data/get-started/#WARC-Format.

[68] *Can I Use: Declarative Shadow DOM*. https://caniuse.com/declarative-shadow-dom.

[69] *ShadowRealm proposal*. https://github.com/tc39/proposal-shadowrealm.

[70] *Tranco top 1M domains, Dec 3, 2022 – Jan 1, 2023*. https://tranco-list.eu/list/3VYXL/1000000.

Appendices

Appendix A

Example `leak-detect` result summaries

A.1 Summary with password leak to tracker

Below is part of the summary output for the crawl of `https://oribi.io/login`, which leaks the password to Rollbar. ‘[…]’ means that text was omitted.

```
Crawl of https://oribi.io/login
Took 44s

══ ⏰ Timeline (see below for more details): ══

⌚8.0s ✒️ fill email field #email
   ⌚13.3s 🔍 📧 email value of field read by script www.clarity.ms/…/clarity.js
   ⌚13.3s 🔍 📧 email value of field read by script oribi.io/…/login-1dc6175bc9a93f3d.js
   ⌚13.3s 🔍 📧 email value of field read by script oribi.io/…/login-1dc6175bc9a93f3d.js
   ⌚13.3s 🔍 📧 email value of field read by script oribi.io/…/main-06c183262e3d9339.js
   ⌚13.6s 🔍 📧 email value of field read by script www.clarity.ms/…/clarity.js
   ⌚13.6s 🔍 📧 email value of field read by script oribi.io/…/_app-13f2c9c38fa150d4.js
⌚13.7s ✒️ fill password field #password
   ⌚13.9s 🔍 📧 email value of field read by script oribi.io/…/main-06c183262e3d9339.js
   […]
   ⌚16.5s 🔍 🔑 password value of field read by script www.clarity.ms/…/clarity.js
   ⌚16.5s 🔍 🔑 password value of field read by script oribi.io/…/login-1dc6175bc9a93f3d.js
   ⌚16.5s 🔍 🔑 password value of field read by script oribi.io/…/login-1dc6175bc9a93f3d.js
   ⌚16.5s 🔍 📧 email value of field read by script oribi.io/…/main-06c183262e3d9339.js
   ⌚16.5s 🔍 🔑 password value of field read by script oribi.io/…/main-06c183262e3d9339.js
   ⌚16.7s 🔍 🔑 password value of field read by script www.clarity.ms/…/clarity.js
   ⌚16.7s 🔍 🔑 password value of field read by script oribi.io/…/_app-13f2c9c38fa150d4.js
⌚21.7s click added button for Facebook tracking detection
   ⌚21.7s 🔍 📧 email value of field read by script connect.facebook.net/…/382622115434455
   ⌚21.7s 🔍 📧 email value of field read by script connect.facebook.net/…/382622115434455
   ⌚21.7s 🔍 🔑 password value of field read by script connect.facebook.net/…/382622115434455
   ⌚21.7s 🔍 🔑 password value of field read by script connect.facebook.net/…/382622115434455
   ⌚21.7s ⚠️ 📤 📧 email sent to www.facebook.com
```

```
⌚24.8s ⏎ submit email field #email
   ⌚24.9s 🔍 📧 email value of field read by script connect.facebook.net/…/382622115434455
   ⌚24.9s 🔍 📧 email value of field read by script connect.facebook.net/…/382622115434455
   ⌚24.9s 🔍 📧 email value of field read by script www.clarity.ms/…/clarity.js
   ⌚24.9s 🔍 📧 email value of field read by script connect.facebook.net/…/382622115434455
   ⌚24.9s 🔍 📧 email value of field read by script connect.facebook.net/…/382622115434455
   ⌚24.9s 🔍 🔑 password value of field read by script connect.facebook.net/…/382622115434455
   ⌚24.9s 🔍 🔑 password value of field read by script connect.facebook.net/…/382622115434455
   ⌚24.9s ⚠️ 📤 📧 email sent to www.facebook.com
   ⌚25.0s 🔍 📧 email value of field read by script oribi.io/…/main-06c183262e3d9339.js
   ⌚25.0s 🔍 🔑 password value of field read by script oribi.io/…/main-06c183262e3d9339.js
   ⌚26.2s ⚠️ 📤 📧 email sent to api.rollbar.com
   ⌚26.2s 🚨 📤 🔑 password sent to api.rollbar.com
   ⌚26.2s 🔍 📧 email value of field read by script oribi.io/…/main-06c183262e3d9339.js
   ⌚26.2s 🔍 🔑 password value of field read by script oribi.io/…/main-06c183262e3d9339.js

══ ⚠️ 📤 Values were sent in web requests to third parties: ══

⌚21.7s 📧 email hash (hex→sha256) sent in url of Image request to 🔺 third party 👁 tracker
"https://www.facebook.com/tr/?[…] " by:
   c (↓ :24:64983)
   y (↓ :24:62575)
   value (↓ :24:30575)
   […]
   https://connect.facebook.net/en_US/fbevents.js :24:45083
   clickFacebookButton (pptr://__puppeteer_evaluation_script__ :8:21)

[…]

⌚26.2s 🚨 🔑 password (plain text) sent in body of XHR request to 🔺 third party 👁 tracker
"https://api.rollbar.com/api/1/item/" by:
   […]
   value (https://oribi.io/_next/static/chunks/pages/_app-13f2c9c38fa150d4.js :1:10172)
   https://oribi.io/_next/static/chunks/pages/login-1dc6175bc9a93f3d.js :1:6210
   l (↓ :1:27400)
   ↓ :1:27188
   https://oribi.io/_next/static/chunks/pages/_app-13f2c9c38fa150d4.js :1:27825
   N (↓ :1:4618)
   s (↓ :1:4857)
   N (↓ :1:4689)
   a (↓ :1:4821)
   ↓ :1:4880
   https://oribi.io/_next/static/chunks/pages/login-1dc6175bc9a93f3d.js :1:4761
   A (https://oribi.io/_next/static/chunks/main-06c183262e3d9339.js :1:101048)

══ ℹ️ 🔍 Field value reads: ══

[…]

⌚21.7s access to 📧 email value of email field "#email" by:
   c (↓:20:9901)
   g (↓:20:10807)
   e (↓:20:8431)
   🔺 third party 👁 tracker https://connect.facebook.net/signals/config/[…]:20:67347
   v (↓:24:75981)
   each (🔺 third party 👁 tracker https://connect.facebook.net/en_US/fbevents.js:24:77527)
   HTMLDocument.<anonymous> (🔺 third party 👁 tracker https://connect.facebook.net/sig-
nals/config/382622115434455?v=2.9.91&r=stable:20:67144)
   HTMLDocument.<anonymous> (🔺 third party 👁 tracker https://connect.facebook.net/en_US/f-
bevents.js:24:45083)
   clickFacebookButton (pptr://__puppeteer_evaluation_script__:8:21)

[…]

📊 Automated crawl statistics:

📝 2 fields found
✒️ 2 fields filled
↩ 1 fields submitted

💧🔍 Leak/sniff statistics:

🚨💧 1 🔑 password leaks to rollbar.com
⚠️💧 3 📧 email leaks to facebook.com, rollbar.com
⚠️💧 1 🔑 password to console leaks
⚠️🔍 9 🔑 password value sniffs by third parties / trackers
ℹ️🔍 6 📧 email value sniffs by third parties / trackers
```

A.2 Summary with DOM attribute leak

Below is a summary from a crawl of `https://www.hellofresh.com/login`. We can see that the password is leaked to DOM by `framework-e4a10fa639c31078.js`. In the source-mapped stack trace below the timeline we can see that the original name of the responsible script was `react-dom.production.min.js`.

```
Crawl of https://www.hellofresh.com/login
Took 56s

══ ⏰ Timeline (see below for more details): ══

⌚11.7s ✒️ fill email field #hf-form-group-username-input
   ⌚17.0s 🔍 📧 email value of field read by script www.hellofresh.com/…/framework-
e4a10fa639c31078.js
   […]
⌚17.3s ✒️ fill password field #hf-form-group-password-input
   […]
   ⌚19.9s 🔍 🔑 password value of field read by script www.hellofresh.com/…/frame-
work-e4a10fa639c31078.js
   ⌚20.0s ⚠️ 🔑 password written to DOM by script www.hellofresh.com/…/framework-
e4a10fa639c31078.js
   […]
⌚25.1s click added button for Facebook tracking detection
```

```
⌚28.1s ⏎ submit email field #hf-form-group-username-input
   ⌚28.3s 🔍 📧 email value of field read by script www.hellofresh.com/…/framework-
e4a10fa639c31078.js
   ⌚28.3s 🔍 🔑 password value of field read by script www.hellofresh.com/…/frame-
work-e4a10fa639c31078.js
   […]
   ⌚28.4s ⚠️ 📤 📧 email sent to tr.snapchat.com
   […]

═══ ⚠️ 🔑 Password was written to the DOM: ═══

⌚20.0s to attribute "value" on element "#hf-form-group-password-input" by:
   bs (↓ :38:106)
   bb (↓ :36:258)
   ab (↓ :234:113)
   ij (webpack://_N_E/node_modules/react-dom/cjs/react-dom.production.min.js :257:224)
   b (webpack://_N_E/node_modules/scheduler/cjs/scheduler.production.min.js :19:278)
   Nf (↓ :123:322)
   gg (↓ :253:268)
   Uj (↓ :244:347)
   c (webpack://_N_E/node_modules/react-dom/cjs/react-dom.production.min.js :124:104)
   b (webpack://_N_E/node_modules/scheduler/cjs/scheduler.production.min.js :19:278)
   Nf (↓ :123:322)
   gg (↓ :124:39)
   jg (↓ :123:422)
   ig (↓ :245:113)
   ay (↓ :245:113)
   Ib (↓ :50:75)
   Mb (↓ :51:72)
   dO (↓ :51:72)
   jd (↓ :76:247)
   yc (webpack://_N_E/node_modules/react-dom/cjs/react-dom.production.min.js :75:122)
   b (webpack://_N_E/node_modules/scheduler/cjs/scheduler.production.min.js :19:278)
   Nf (↓ :123:322)
   gg (↓ :293:9)
   Hb (webpack://_N_E/node_modules/react-dom/cjs/react-dom.production.min.js :74:257)
   b (https://www.hellofresh.com/assets/releases/web-infra/_next/static/chunks/pages/_app-
e069072fd3c7a024.js :13:2877)


If a script then extracts the DOM it might leak the password in a web request

═══ ⚠️ 📤 Values were sent in web requests to third parties: ═══

⌚28.4s 📧 email hash (hex→sha256) sent in body of Ping request to 🔺 third party 👁 tracker
"https://tr.snapchat.com/collector/prep_mapping" by:
   Tt (↓ :1:5689)
   Pt (↓ :1:6699)
   ↓ :1:22963
   https://sc-static.net/scevent.min.js :1:22996
   b (https://www.hellofresh.com/assets/releases/web-infra/_next/static/chunks/pages/_app-
e069072fd3c7a024.js :13:2877)
```

```
══ ℹ️ 🔍 Field value reads: ══
[…]


📊 Automated crawl statistics:

📑 2 fields found
🖋️ 2 fields filled
↵ 1 fields submitted

💧🔍 Leak/sniff statistics:

⚠️💧 1 📧 email leaks to snapchat.com
⚠️💧 1 🔑 password to DOM attribute leaks
```

Appendix B

Extra figures

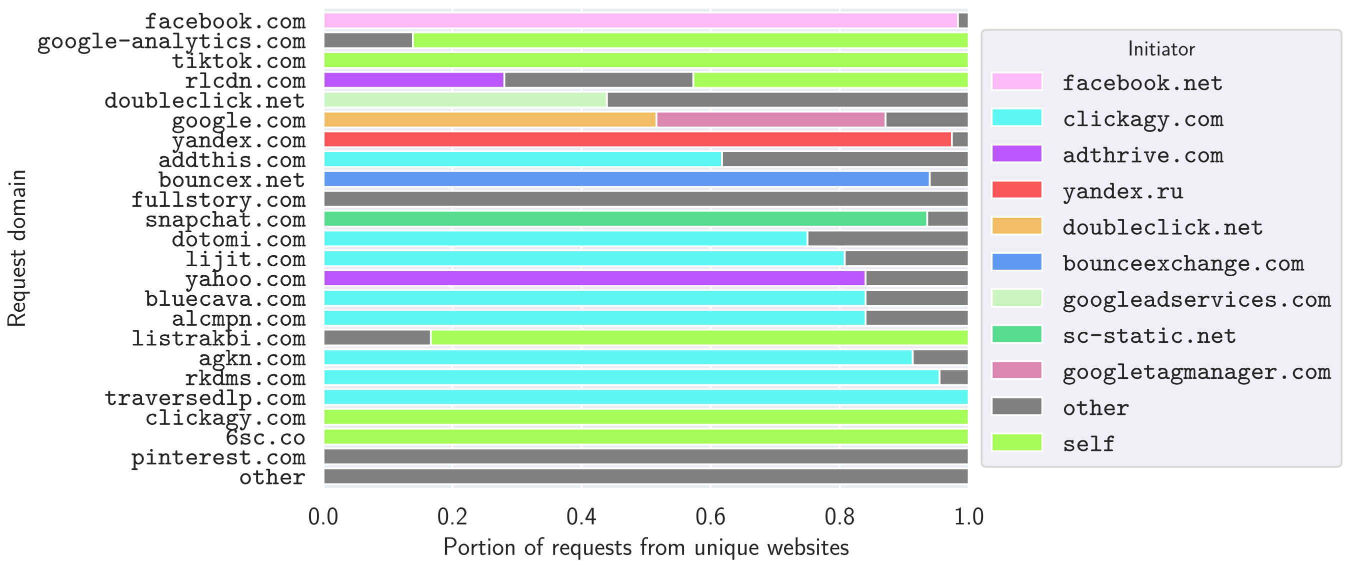

Figure B.1: Caption for non-log: Domains of scripts initiating requests for top leaking trackers. 'self' indicates that the script domain is the same as the tracker domain.

Appendix C

Password leaks

C.1 Timeline regarding Crash In You leak

1. Aug 8, 2017: `crashinyou.net` is registered [1]
2. Aug 13, 2017: Site is captured by the Wayback Machine, showing that it hosted some software, including a Minecraft cheating tool [2]
3. Oct 12, 2017: Site now contains some files, which include the tool from before and some files with a website name and the word 'Database' in the name, possibly indicating a user account leak. However, we could not find a malicious script in archived versions of these websites
4. Jan 22, 2018: Site is first captured with 'Site temporarily unavailable' page from hosting company
5. Mar 5 & 16, 2018: Site now contains a 'Minecraft checker' form requiring a nickname and key, which searches various account databases, some of which include passwords
6. Mar 21, 2018 up to Sep 5, 2019: Site is captured being suspended again
7. Jun 25, 2018 up to Jul 2, 2018: Forum threads sharing Minecraft accounts and mentioning `crashinyou.net` were active [3]
8. Oct 18, 2018: `crashinyou.com` is first captured, containing a layout copied from `crashinyou.net` [4]
9. Jun 5, 2019: `crashinyou.me` is registered
10. Jun 13, 2019: `crashinyou.com` is captured containing just the text '`crashinyou.me`'
11. Jul 14, 2019: `crashinyou.com` is suspended
12. Jul 28, 2019: Malicious script is added to `pixelmon.ru`
13. Sep 3, 2019: `crashinyou.me` is first captured, containing a checker form with a different layout [5]

[1] We determined this using WHOIS
[2] `https://web.archive.org/web/*/http://crashinyou.net/`
[3] `https://zelenka.guru/threads/518419/`, `https://zelenka.guru/threads/518552/`
[4] `https://web.archive.org/web/*/http://crashinyou.com/`
[5] `https://web.archive.org/web/*/http://crashinyou.me/`

14. Dec 23, 2019: `crashinyou.net` is first captured hosting an endpoint similar to the one used on the TechPowerUp forum

15. Feb 6, 2020: Malicious script is inserted into template of TechPowerUp forum login page, according to forum software

16. Feb 23, 2020: TechPowerUp forum is first captured containing malicious script [6]

17. Dec 7, 2020: Endpoint on `crashinyou.net` used for TechPowerUp is first captured, but with 404 code

18. Nov 30, 2021: Empty `crashinyou.net` homepage is captured

19. Mar 14, 2022: `crashinyou.me` redirects to `premium.crashinyou.me`

20. May 21, 2022 up to 5 Dec 2022: Endpoint used for TechPowerUp is captured again, as empty page

Empty pages might also be caused by anti-bot defense mechanisms from Cloudflare.

[6] `https://web.archive.org/web/20200223214241/https://www.techpowerup.com/forums/login/`

C.2 Found password leaks

Tranco rank [a]	First party	Third party	First-party response	Third-party response	Only on login error
2 308	*Oribi*	Rollbar	✗ None	–	Yes
5 389	TechPowerUp	*CrashInYou.net* (malicious)	✓ Fixed	–	No
9 943	Princeton Review	*WebFX* (marketingcloudfx)	No (automatic)	✗ None	No
17 282	Xsolla	Inspectlet	✓ Fixed	None	No
23 004	BDO USA [b]	*WebFX*	None, but tracker was removed from site	None	No
34 803	NoRedInk [b]	*WebFX*	–	✗ None	No
78 539	Workspot	ZoomInfo (insent.ai)	None, but does not happen anymore	No (automatic)	No
81 297	*Wacoal Japan*	Facebook	None, but seems to be fixed	–	Possibly
92 361	Vacheron Constantin	*WebFX*	None, but tracker was removed from site	None	No
123 198	Minsk Branch of the Belarusian Chamber of Commerce and Industry	Yandex	✗ None	–	Possibly
175 764	*Pixel Joint*	Google Analytics	✗ None	–	Possibly
242 031	Adbeat	Autopilot (Ortto)	✗ None	–	No
N/A	Skin Type Solutions [b]	*WebFX*	–	✗ None	No
N/A	River NYC [b]	*WebFX*	–	✗ None	No
N/A	PixelMon.ru [b]	*CrashInYou.net*	✗ None	–	No

'Only on login error' means that the leak only occurs with invalid credentials

First party	Third party	Timing	Cause
Oribi	Rollbar	post submit	Request is logged on failed login, log is recorded
TechPowerUp, PixelMon.ru	*CrashInYou.net*	post submit	Inline script collecting credentials
Princeton Review, ...	*WebFX*	post fill	Script collects any values for input events
Xsolla	Inspectlet	post fill	Script collects `data-value` DOM attribute
Workspot	ZoomInfo	post submit	Form submission collection
Wacoal Japan	Facebook	post submit	Server puts POST parameters in `<meta>`
Minsk Branch of the Belarusian Chamber of Commerce and Industry	Yandex	post submit	Server puts password in hidden field on submission
Pixel Joint	Google Analytics	post submit	GET form
Adbeat	Autopilot	post submit	Form submission collection

The party clearly at fault is *italicized*

[a] Rank in Tranco Dec 3, 2022 – Jan 1, 2023 top 1M list [70]
[b] Found manually