The Cookie Hunter: Automated Black-box Auditing for Web Authentication and Authorization Flaws

Kostas Drakonakis*
FORTH ICS, Greece
kostasdrk@ics.forth.gr

Sotiris Ioannidis†
Technical University of Crete, Greece
sotiris@ece.tuc.gr

Jason Polakis
University of Illinois at Chicago, USA
polakis@uic.edu

ABSTRACT

In this paper, we focus on authentication and authorization flaws in web apps that enable partial or full access to user accounts. Specifically, we develop a novel *fully automated* black-box auditing framework that analyzes web apps by exploring their susceptibility to various cookie-hijacking attacks while also assessing their deployment of pertinent security mechanisms (e.g., HSTS). Our modular framework is driven by a custom browser automation tool developed to transparently offer fault-tolerance during extended interactions with web apps. We use our framework to conduct the first automated *large-scale* study of cookie-based account hijacking in the wild. As our framework handles every step of the auditing process in a completely automated manner, including the challenging process of account creation, we are able to fully audit *25K* domains. Our framework detects more than 10K domains that expose authentication cookies over *unencrypted* connections, and over 5K domains that do not protect authentication cookies from JavaScript access while also embedding third party scripts that execute in the first party's origin. Our system also automatically identifies the privacy loss caused by exposed cookies and detects 9,324 domains where sensitive user data can be accessed by attackers (e.g., address, phone number, password). Overall, our study demonstrates that cookie-hijacking is a severe and prevalent threat, as deployment of even basic countermeasures (e.g., cookie security flags) is absent or incomplete, while developers struggle to correctly deploy more demanding mechanisms.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

Black-box Testing; Cookie Hijacking; Authentication; Authorization; Large-Scale Measurement

*Part of this work was completed while at the University of Illinois at Chicago.
†Sotiris Ioannidis is also with FORTH ICS.

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3417869

ACM Reference Format:
Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. 2020. The Cookie Hunter: Automated Black-box Auditing for Web Authentication and Authorization Flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3372297.3417869

1 INTRODUCTION

Web services have become treasure troves of sensitive data, rendering user accounts high-value targets for attackers. Recently, authentication flaws in popular web applications (or "apps") exposed sensitive data and allowed access to critical functionality of millions of accounts [4, 5]. Reports have even implicated nation-state adversaries in attacks that ultimately aimed to steal user credentials [6, 7]. As such, authentication and authorization flaws in web apps are of great importance [89, 98] as they pose a significant threat. However, detecting such flaws is challenging.

As new technologies and features continue to emerge, web apps are becoming increasingly complicated. This complexity is exacerbated by their rapid evolution and the addition of new functionality and modules [35, 39]. This can result in the introduction of semantic bugs whose composite nature [81] renders detection a challenging task [39, 70]. Moreover, the massive codebase that comprises modern web apps is often developed by separate teams, which can have a negative impact [72] and result in fragmented auditing procedures that do not fully capture the side effects that arise from the interoperability of different components. Web apps can also include legacy code, which is often a significant source of new vulnerabilities [33], further complicating internal auditing procedures. To make matters worse, applicable security mechanisms are often deployed in an incomplete or incorrect manner [32, 47, 52, 76, 92]. As a result, external auditing initiatives from researchers can significantly contribute to the overall hygiene of the web ecosystem by discovering vulnerabilities. However, the sheer scale of this issue and the prevalence of obfuscation [78] mandate an automated, black-box dynamic analysis.

In this paper we adopt such an approach and focus on flaws that lead to the exposure of authentication cookies that allow adversaries to access sensitive data or account functionality. While recent studies have demonstrated that such flaws exist even in the most popular websites [30, 44, 77], these studies relied on significant manual effort and were, thus, inherently small-scale covering a very limited number of domains. With surveys reporting that Internet users in the US now have ∼150 password-protected accounts [2], and tens of thousands of websites streamlining account creation through Single Sign-On [44], it is apparent that manual efforts are not sufficient. To that end, we develop a completely

automated black-box auditing framework that detects authentication and authorization flaws in web apps and identifies what sensitive/personal user information can be harvested by attackers. Our system is designed to handle *every step* of the process, including account creation and user-level interactions. Specifically, our framework analyzes the characteristics and infers the access privileges granted to cookies, while also evaluating the deployment of security mechanisms that can prevent cookie-hijacking attacks.

The main design goal of our framework is to automatically audit web apps in a black-box manner, without any prior knowledge of the underlying app's structure or code. The framework is driven by XDriver, our custom browser-automation tool built on top of Selenium, designed for robustness and fault-tolerance during prolonged interactions with web apps. As XDriver is geared towards security-related tasks, we have implemented modules for evaluating security mechanisms that are pertinent to our study (e.g., HSTS). The black-box auditing process is handled by a series of components dedicated to specific phases of our workflow, including components that employ differential analysis and a series of oracles for inferring the account's "state" reached by requests depending on the cookies submitted and the level of account access granted to those cookies. This requires identifying which cookies are used for authentication and exploring the conditions for different attack vectors under which they can be hijacked. Finally, our framework includes a novel module that analyzes web apps and detects personal user data (e.g., name, email, phone number) that is accessible using hijacked cookies. This is achieved through an in-depth investigation that analyzes the app's client-side source, storage, and URL parameters to detect the exposure of sensitive data.

Using our framework we conduct the first *fully automated*, comprehensive, large-scale analysis of cookie hijacking in the wild. First, we crawl 1.5 million domains, and identify over 200 thousand domains that support account creation. Subsequently, our framework manages to fully audit almost *25 thousand* (~12%) of the domains, requiring 8.5 minutes per domain on average. Our experiments reveal that 50.3% of those domains expose their cookies under different scenarios and, thus, suffer from authentication or authorization flaws. To make matters worse, we find that security mechanisms that could prevent these attacks are not widely adopted (only 11.8% of vulnerable domains do so) or are often deployed in an erroneous manner. In more detail, we find that 10,921 domains expose authentication cookies over unencrypted connections, which can be hijacked by passive eavesdroppers and used to access users' accounts. Moreover, 5,099 domains do not protect their authentication cookies from JavaScript-based access while simultaneously including embedded, non-isolated, third party scripts that run in the first party's origin. With these scripts being fetched from 2,463 unique third party domains, users currently face a considerable risk of malicious, compromised, or honest-but-curious third parties reading their authentication cookies.

Due to the severity of the flaws detected by our system, it is crucial that our findings are made available to developers so they can patch their systems. While we have notified several vulnerable domains, finding an appropriate contact point for such a vast number of domains is infeasible; thus, we will set up a notification service that allows developers to access the auditing results. In summary, our main research contributions are:

- We develop a custom browser automation tool that transparently offers robustness during prolonged interaction with web apps. Our tool is tailored for security-oriented tasks and includes modules for assessing relevant security mechanisms. As our system can streamline a wide range of research projects, our code will be made open source.
- We develop a novel framework for the automated black-box detection of flaws in web apps. Our framework incorporates a series of modules and oracles that employ differential analysis for automatically evaluating the feasibility of cookie hijacking attacks under different threat models, and detecting the exposure of personal user data across multiple dimensions. To facilitate further research, we will share our code with vetted researchers upon publication.
- We conduct the largest study of cookie-based authentication and authorization flaws by auditing ~25K domains. Our comprehensive evaluation reveals a plethora of security malpractices and misconfigurations, as 50.3% of the domains are vulnerable to at least one attack.

2 BACKGROUND AND THREAT MODEL

Our framework focuses on detecting authentication and authorization flaws that stem from the incorrect handling or protection of cookies. While cookie hijacking is not a new attack vector, it can still affect even the most popular websites (e.g., Google, Facebook) and expose users to significant threats [77] including complete account takeover [44]. We consider the following types of attackers.

Passive network attacker. This attacker, referred to as an *eavesdropper*, has the ability to intercept and inspect unencrypted HTTP traffic (but does not attempt to modify it). We assume this attacker cannot intercept HTTPS traffic, and do not explore more elaborate, active attacks (e.g., SSL-stripping [60], cookie-overwriting [94]). This means that any cookies that are not protected with the *secure* flag can be intercepted by this attacker when appended to an HTTP request. This can, e.g., occur naturally while a user browses a website (since many websites serve certain resources over HTTP). An important detail that amplifies the practicality of this attack is that even when a domain supports HTTPS, browsers will by default attempt to access the domain over HTTP before being redirected by the web server to HTTPS [77]. While this can be prevented with mechanisms like HSTS, they are still not widely adopted and are often deployed incorrectly [52, 76].

Web attacker. This attacker can execute some JavaScript code within the origin of the web app, e.g., through a cross-site scripting (XSS) attack [45]. Another attack vector is introduced if the web app includes a script from a third party domain without "isolating" it in an iframe, effectively allowing it to execute in the first party's origin [65]; malicious scripts (e.g., malvertising [59]) or compromised script providers can then read first party cookies [18]. We define as third-party any scripts that are loaded from a different domain [73, 82, 83], where the term *domain* will be used to refer to the eTLD+1 domain throughout the paper. Consequently, cookies that are not protected with the *httpOnly* flag will be readable by client-side code and can be obtained by the attacker. We refer to these two attack vectors as *JS cookie stealing*.

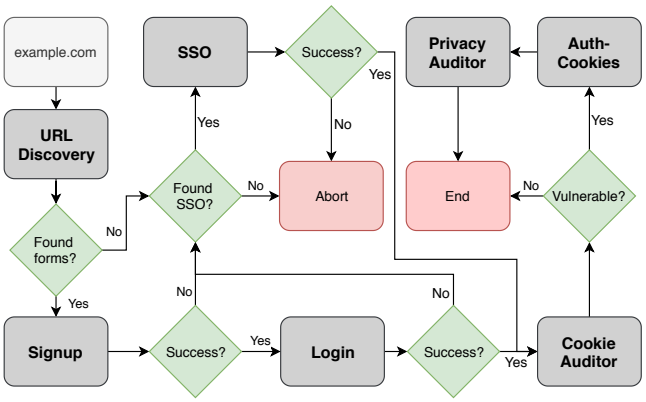

Figure 1: Major phases in our auditing workflow.

It is important to stress that our framework does not search for XSS bugs or malicious third party scripts; our system focuses on automatically inferring the feasibility of stealing authentication cookies through JavaScript due to insufficient protection, and exploring the subsequent privacy implications for users. As such, the numbers reported on JavaScript-based cookie stealing are an upper bound that is contingent on the presence of XSS vulnerabilities or *malicious* third party scripts. Nonetheless, XSS vulnerabilities remain one of the most common attacks against web applications [1] and a plethora of detection systems have been proposed (e.g., [20, 82, 85]). Similarly, recent work has highlighted the prevalence of (suspicious) third party scripts [49, 55].

3 SYSTEM DESIGN AND IMPLEMENTATION

Here we present our framework and the methodology of the core components of our black-box auditing process. Figure 1 depicts a high-level view of the workflow for clarity, and to facilitate presentation. In the following subsections we highlight each component in our pipeline and provide design and implementation details.

3.1 Automated Account Setup

The first phase in our workflow is to automatically create accounts.

URL Discovery. This module follows a straightforward process of crawling domains and terminating when *both* a login and a signup form have been located. As a first step it explores the URLs included in the public dataset by Ghasemisharif et al. [44]. If it does not locate both types of forms, next it will crawl the target web application. The crawl starts at the landing page and goes to a depth of 2 – we opt for a more shallow crawl to reduce the crawl's duration and enable our large-scale study. Our framework collects all links included in each page that point to the same domain, and subsequently visits and inspects them. This step prioritizes links that contain an account-related keyword (e.g., signin, register etc.) and follows a breadth-first search (BFS) approach. If both types of forms are yet to be found, the final step is to collect the first 30 links from the homepage and inspect them, excluding previously-visited URLs. This is based on the intuition that such pages are typically easily accessible to users and not hidden behind multiple menus, and are usually at the top of the page.

For each visited page, we extract any forms that resemble a login or signup process, and a series of heuristics are employed for detecting such forms within a page's code. Specifically, for each form we first count the number of text, email, password, checkbox and radio type input fields. We also check which of those are visible following the custom heuristics proposed by SSOScan [96]. If there are no password fields we skip the form since it probably is not a login or signup form (e.g., contact forms are common). If it contains more than one password field we label it as a signup form since such forms usually require the user to retype the password for verification. If there is a single password field and a single text field we label it as a login form, as this is the typical structure of such forms. If there are more than one text fields or checkbox/radio fields (accounting for the *"remember me"* option in login forms) the form is labeled as a signup form. If the form has a more irregular structure and has not been identified with these heuristics, our system resorts to using two sets of regular expressions (one for login and one for signup) for analyzing the HTML code and detecting elements that allow us to label the form accordingly.

Automated sign up. Automating the account creation process in an application-agnostic way is a challenging task. This is due to the fact that websites have different requirements and constraints regarding the type and format of information for the fields needed for completing the registration. These vary and pertain to the number and type of fields (e.g., email, password, username etc.), as well as to the different restrictions in what is considered a valid input. For instance, a website might consider "+1 012 345 6789" a valid US number while another might require a different format.

The Signup module iterates over the discovered signup pages and attempts to fill each candidate form appropriately. We use a manually-curated set of regular expressions that try to detect what type of information each input element is expecting (e.g., email, postal address, date). We first carefully assign labels to each of the input elements by checking the *for* attribute of *label* elements, since we expect them to be the most descriptive. If there is no match, we move on to the element's HTML code (i.e., its attributes), which can reveal useful information about its type (e.g., an element of type `email` or with a descriptive `id` like `last_name`). If our module has yet to identify what type of information is expected, we consider the text content preceding the element. While this is the most common convention for labeling elements, developers are not constrained and can structure their forms differently. We, thus, follow a conservative strategy and consider these assigned labels as *possible* labels, since we cannot be certain of the form structure – in some cases the input element's accompanying text might be after the element. This is also why we prioritize any previously identified labels, and consider the "possible" labels as a last resort.

If there is still no match, we use Google Translate to translate any labels assigned to the element in English and repeat the aforementioned process. This is needed since our analysis is not limited to English websites and foreign content is common. We refrain from using Google Translate initially, since the previous steps might reveal the type of field, allowing us to avoid the unnecessary API calls. Finally, we resort to either a random string for *text* inputs or a random selection for *select* and *radio* elements. To generate valid inputs after having detected the element's type, we use Python's `Faker` package. We also infer the input's expected size by inspecting

its `size` and `maxlength` attributes and adjust our value accordingly. After filling out the inputs we submit the form. At this point we need to infer whether the signup attempt was successful or not. We employ the following oracle that deems the signup process successful if any step yields a positive result:

- Visit the homepage and check if any of the submitted identifiers appear. The intuition is that if signup failed, websites would not store the provided information. We refrain from making the same check at the landing page after the form submission, since a website might display identifiers in an error message.
- Visit the form's URL and check if it is still displayed. The intuition is that after a successful signup the website will not keep displaying the form. However, we have observed cases where the signup was successful, but the signup form was still displayed.
- Check if we received any emails from the domain. The intuition is that a failed signup attempt would not trigger an email delivery.
- Attempt to login to the website with our automated Login module (described further down). A successful login attempt indicates that the signup was successful.

If the signup is deemed successful we store the filled values and end the signup process. Otherwise, we try to identify any required fields in the form (i.e., by checking for the HTML *required* attribute or an asterisk or the *required* keyword in the element's labels) and attempt to resubmit the form using only those, to reduce the probability of error. If that fails once again, we move on to the next form, until a successful registration is detected or all forms have been processed. After registration we also handle any emails sent by the domain, typically pertaining to account verification, to ensure that our newly created account is valid. As we cannot be certain of those emails' structure or of any action that might be required, we extract and visit all URLs included in the email and try to detect commonly used keywords and phrases pertaining to successful verification. Through empirical analysis we observed that several websites might require the user to additionally click on a button in that page to finish the process. Therefore, if we do not detect any of the above keywords, we resort to clicking all displayed clickable elements in the page.

Automated login. For us to complete the login process, we visit the discovered login URLs (i.e., the ones that contain a login form) and submit each candidate form with our test account credentials. Concluding whether the login attempt has been successful is straightforward in most cases; the login oracle re-fetches the page with the login form and checks whether the submitted form remains in the page. If not, the login attempt is considered successful. During our empirical analysis we observed that several poorly designed websites kept displaying the form even after a successful login; to account for such cases, if the form persists, our login oracle additionally checks if any of our test account's identifiers (e.g., email, username etc.) are now present in the homepage's source code. Similarly, it uses a set of heuristics for detecting whether any logout buttons are displayed in the homepage. If either process yields a result the login is deemed successful.

SSO Fallback. If our system is not able to successfully complete the traditional account creation process, it alternatively identifies whether the app supports Single Sign-On with one of the most popular Identity Providers (*IdPs*) – we currently support Facebook and Google. If SSO elements are discovered it attempts to automatically complete the SSO process using test accounts that have been registered in the IdPs. First we need to identify if the site actually supports SSO; we have created a set of regular expressions that identify potential HTML elements in a page that can be used for performing SSO. The detection of such elements is performed during the execution of the URLDiscovery module. The module terminates if both login and signup forms have been located, regardless of the discovery of potential SSO elements. This is due to the fact that the available SSO options usually accompany the account related forms (if a traditional login scheme is supported). Thus, when locating a login or signup form we also detect if the site also supports SSO.

For each URL, we iterate over the candidate SSO elements and click them. We prioritize elements that are displayed, based on the intuition that sites are usually upfront about the available login options. For displayed elements we use Selenium's `click` method, effectively replicating a user's action. For hidden elements we refrain from trying to make those elements appear, which would involve clicking over other elements and potentially leading to unintended behavior and considerably increasing the process' duration. Instead, we try to trigger their `onClick` method via JavaScript. While this is generally effective, in some cases the candidate element is an outer wrapper element (e.g., a * * element which contains an *<a>* element), and clicking it via JavaScript will not trigger SSO. Thus, for each non-displayed candidate element we also consider its children elements. While this leads to additional elements that need to be tested, we can quickly click on elements and decide if one is an actual SSO element; the overhead induced by this approach is negligible in practice.

The straightforward approach for inferring whether we clicked the correct element is to wait for the appearance of a predefined element, as a button that authorizes the app to access user data on the IdP should appear. However, this is inefficient and expensive as we would need to wait a sufficiently long time after clicking on every element to ensure that the necessary steps (and background server-communication) of the SSO protocol actually completed. We opt for a more elaborate approach that relies on the fact that an HTTP request is issued towards the IdP's SSO endpoint when the correct element is clicked. We setup a modified proxy in passive mode which notifies our framework if such an outgoing request is observed. This allows us to quickly iterate over all candidate elements. The first time our system logs into a website we authorize the app in the IdP by following a few easily-automated steps.

It is worth noting that inferring whether the SSO process was successful is not necessarily equivalent to determining if our system is logged in the web app. For instance, a website might require a few extra steps to be taken (typically pertaining to account setup) after the user clicks on the SSO button and authorizes the app in the IdP; in this case our system will be in an intermediary state where the user is not yet fully logged in. We employ two separate oracles to decide if SSO completed and if we are logged in. The *SSO oracle* first checks if the SSO element we clicked on is still displayed. If not, the SSO was (most likely) successful. However, as some websites keep displaying the elements even after a successful SSO, the SSO oracle utilizes the *SSO login oracle* for further verifying the successful completion of the SSO process. This oracle searches for *displayed* account identifiers, logout buttons, and our IdP test

account's profile photo which is often fetched from the IdP. If any of those checks is positive, the SSO login is deemed successful. This oracle focuses only on displayed elements, because we found cases where a website that was authorized in the IdP loaded identifiers provided by the IdP and displayed them in the page's source (e.g., in an inline JavaScript object) without having logged the user in.

Some websites require a few extra steps pertaining to account setup to be taken in order to complete the SSO. We detect and automate this process as well, using a modified Signup module that has a few minor changes in its workflow and oracle, which address SSO-specific variations in the process. Typically, websites display two options for completing the account setup after a successful SSO, the first being to link the new SSO identity with an existing account and the second about creating a new account. We detect any clickable elements that indicate the latter using regular expressions and iteratively click them. We then collect *all* forms displayed in the page, as we do not have any knowledge of their structure (i.e., it is common that such an account setup form might not even include a password field). Finally, we iterate over the discovered forms, fill and submit them, and consult our modified Signup oracle for each submission. As such, the oracle has been modified so the check for identifiers is done only on displayed elements, for the same reason with the SSO login oracle. In addition, if all other checks fail, we check if any password type fields were submitted in the signup form. If that is the case, we proceed by performing a *generic* login attempt using the discovered login forms.

False Positive/Ambiguous Login Elimination. After creating an account, we perform a final step to eliminate cases where our oracles yield a false positive (i.e., consider a login attempt to be successful despite not actually being logged in) or are not able to disambiguate between being logged in or not for a specific website. We send an HTTP request without appending *any* cookies and consult our login oracle once again; if it claims we are still logged in we mark the website as a false positive and abort the process. This happens when a website does not follow any of the development "conventions" that our oracles anticipate, or other mechanisms interfere with the session's state (e.g., a website displays an identifier that was stored in `localStorage` even when no cookies are submitted). It is worth noting that while it is straightforward to clear such storage mechanisms, we refrain from doing so since this can have unexpected effects on a website's intended functionality and impact the operation of subsequent modules.

Captchas. Protecting account creation through captchas is common practice and, as such, creating a captcha solver can considerably improve the coverage our system obtains. Initially, we implemented a solver based on recent attacks against Google's audio reCaptcha [22, 80]. Unfortunately, reCaptcha's advanced risk analysis system currently detects the use of WebDriver, which results in Google not serving captchas to our framework. Since building a stealthier captcha solver is out of the scope of our work, and funding human captcha-solving services to create accounts presents an ethical dilemma, we opted to not handle such cases. However, due to the popularity of domains that employ captchas, in our evaluation we include a set of popular domains for which we completed the account creation process manually. We stress, however, that the ~25K domains that comprise the bulk of our evaluation *did not* require *any* manual intervention.

3.2 Cookie Auditor

To investigate whether users are exposed to session hijacking attacks due to flawed or vulnerable authentication practices, the next phase of our framework's workflow relies on modules that analyze the cookies set by a specific web app and identify potential hijacking opportunities based on their attributes. As we require a method for deducing with minimal overhead which cookies provide some form of authentication, we design and implement a simple, yet effective, algorithm that we present in Algorithm 1 (see Appendix). The core idea is to inspect whether the discovered cookies are protected with the appropriate security-related attributes and subsequently infer which of those cookies are used for authentication.

Cookie attributes. Our CookieAuditor algorithm begins by identifying which cookies set by the website are protected with the *secure* and *httpOnly* attributes and groups them accordingly (line 2). If a cookie has both attributes enabled, it will be included in both sets. It then iterates over these cookie sets (8) and infers whether the website is vulnerable to a specific attack from our threat model based on the corresponding attribute. Before actually evaluating a cookie set, it first checks if the set is empty. This indicates that the site is vulnerable to the attack, e.g., if none of the cookies has the *secure* flag set, an eavesdropper could successfully perform a cookie hijacking attack (9-10), as described in prior manual studies [77]. On the other hand, if the attribute is present in one or more cookies, the algorithm will either infer the result from the previously tested set or evaluate this cookie set.

Evaluating a set means that we exclude it from the browser's cookie jar (i.e., those cookies will not be sent in the subsequent request), issue a new HTTP request to the website, and consult the login oracle to determine if we are still logged in (30-32). As can be easily deduced, being logged in while excluding all cookies with a specific attribute means that the website is indeed vulnerable to the specific attack. However, if the exact same cookie set has been tested before we can directly conclude whether the website is vulnerable or not (14-15). Finally, in cases where the cookie set is a subset of a previously tested set where our test account remained logged in, we can again safely conclude that the website is vulnerable for this attack as well (16-18). For instance, if we excluded the set *[A, B, C]* and we were still logged in (i.e., vulnerable) then testing the set *[A, C]* would also result in a logged in state, since we would now send even more cookies than before. This is why we prioritize larger cookie sets (we omitted this part of our algorithm for brevity). Finally, after evaluating a cookie set, we send another request containing all the cookies, to make sure our session is still valid. (only if we were logged out after the test). If the session has been invalidated by the server, we login again and update our cookie values with those of the new session. This allows us to efficiently identify if a website is susceptible to cookie hijacking and, if so, via what means. In the worst case scenario, our approach would need 9 requests, i.e., 3 requests per security-related cookie attribute. It is important to note that this technique has the drawback of not revealing which of the cookies are actually authentication cookies.

Authentication Cookies. To further analyze the root causes of authentication flaws, our framework needs to be able to identify the subset of authentication cookies among all the cookies that are set. Mundada et al. [64] proposed an algorithm, however, their

approach overlooks certain cases and can lead to incorrect results. We build upon the core algorithm they proposed and modify it to correctly handle additional cases. Their proposed algorithm starts by considering only the cookies set at login time (*login cookies*) and generating a partially ordered set (*POSET*) of every possible combination. Since the search space is exponential, and in many cases infeasible to test all combinations, the algorithm establishes a series of *rules* based on the outcome of certain tests to reduce the testing time. The core algorithm works as follows:

- Alternate by testing one round from the bottom of the POSET (i.e., disabling cookies from a full cookie set) followed by a round from the top of the POSET (i.e., enabling cookies from an empty cookie set). According to their description, rounds are followed in an incremental manner and all cookie sets for a given round are tested consecutively (e.g., all cookie sets where only 1 cookie is disabled, then all cookie sets where 1 is enabled etc.). This is also the root cause that leads to incorrect results in certain cases, as we detail next.
- If a disabled cookie set causes the test to fail (i.e., the user is logged out), then all subsequent cookie sets that do not contain this set can be skipped.
- If an enabled cookie set is found to cause the test to succeed (i.e., the user remains logged in), then all subsequent cookie sets that contain this set can be skipped.
- If a cookie that was not set at login time is detected to be part of an authentication combination, a similar nested process is executed for the *non-login cookies* and the login cookie array is expanded to include these cookies.

While this approach is generally effective, we have identified scenarios where it yields incorrect results. To illustrate such a case, consider the following example: if a website has two authentication cookie combinations, e.g., *[A,B]* and *[C,D]*, the algorithm will first set a rule when disabling two cookies. Specifically, when disabling *[A,C]* none of the authentication cookie combinations we are looking for will be complete, and the user will be logged out of the web app. This results in establishing the rule *"any cookie set that does not include [A,C] should be skipped"*. Later on, when disabling the set *[B,D]* (which satisfies the first rule), the user will again be logged out, leading to a similar rule for this set as well. At this point the ruleset dictates that *any* set that does not include *[A,C]* or *[B,D]* will be skipped. However, in the very next round (i.e., when enabling two cookies), when checking whether the actual authentication cookie combinations should be tested, the algorithm will skip them as they do not satisfy the above ruleset. As a result, the actual authentication cookie combinations will not be inferred.

Thus, we cannot blindly follow such rules when enabling cookie sets. This, however, introduces the risk of a major performance penalty. Consider a second example of a website that has two authentication combinations, e.g., *[A]* and *[B]*. The first rules the algorithm will set will be when enabling a single cookie. Specifically, when only enabling *[A]* the user will be logged in and a rule will be set, dictating that *"any cookie set that includes [A] should be skipped"*. Likewise, when enabling *[B]* a similar rule will be set. In the next round (i.e., when disabling two cookies) the only set that will be tested will be the one not containing *[A]* and *[B]*, as it is the only one that respects the current ruleset, and the user will be logged out. This results in the rule *"any cookie set that does not include [A] or [B] should be skipped"* being set. Next, when enabling two cookies, and having established that we cannot follow the last rule when enabling cookies, the algorithm will then test *all* sets of length two that do not contain any of the two authentication cookies. The following rounds of the algorithm behave similarly (i.e., disabling/enabling three cookies and so on). However, we can tell that the algorithm has already detected the authentication cookie combinations and should not try any more tests.

To avoid this performance issue, we modify the algorithm to respect such rules when enabling cookies, but in a slightly different manner: cookie sets that result in the user being logged out when disabled are flattened into a vector (e.g., the ruleset [[A,C], [B,D]] from the first example becomes [A, B, C, D]) and we safely skip the cookie sets that do not include **any** of these cookies. In our first example this results in the authentication cookie combinations being detected. In the second example it results in not testing any sets that are redundant after detecting the correct combinations.

We also note that while we label them as *authentication* cookies, since they lead to the exposure of user identifiers, this might be the result of flaws in the web app's authorization policies, and not due to them actually being designed as (or intended for) authentication. Nonetheless, our goal is not to infer the developers' intention but to identify which cookies lead to (full or partial) authentication.

3.3 Privacy Leakage Auditor

Apart from automatically detecting flaws that expose authentication cookies, our goal is to also identify what personal or sensitive user data attackers can obtain. We develop PrivacyAuditor for locating leaked user information following a differential analysis methodology. Our framework first effectively replicates a session hijacking attack; it creates a fresh browser instance and includes all *stolen* cookies, i.e., the ones that are not protected with the corresponding cookie attributes. If our system has labelled a specific web app as susceptible to both eavesdropping and JS cookie stealing attacks we only simulate the eavesdropping attack to demonstrate the privacy threat posed by attackers that are *less sophisticated* due to space constraints. Our system also deploys a logged-out browser alongside the authenticated browser and then proceeds with collecting links of interest. The module focuses on URLs that match account related keywords (e.g. *profile, settings*) and also collects the top 30 links that appear in the main browser but not in the logged-out one (or less if not that many exist). Typically, we expect those links to point to restricted areas of the website where user information, possibly sensitive, will be stored.

We check each page for user information that was supplied during the signup process. If SSO was used, our system also checks for information that the web app might have pulled from the IdP (we have populated our Facebook and Google profiles with additional information). We inspect the rendered page source once JavaScript-generated content has finished loading. Since user data can be leaked in ways that are not directly visible to the attacker, our system also inspects other potential leakage points, including cookies, local and session storage, and the page's URL (we do not look at outgoing connections since we are not interested in what information is shared with third parties, and leaked identifiers will already be present in one of the locations we search). To account for

cases where user information may be "obfuscated", we also check for encoded values of all the identifiers using common encoding (base64, base32, hex, URL encodings) and hashing techniques (MD5, SHA1, SHA256, SHA512). While we are able to capture obfuscated values of all user-specific information, in our experimental evaluation we only discuss obfuscated passwords and emails; this is due to their sensitive nature and because hashed emails can constitute PII and in certain cases are easily reversible [3, 37, 61].

3.4 Browser Automation

At the heart of any web app auditing framework lies the browser and, thus, it is imperative that our framework is orchestrated by a robust browser automation component. In practice, while Selenium is a powerful tool, it is better suited for testing scenarios when the web app's *structure* and *behavior* are known in advance. However, when conducting a complex, large-scale analysis there is no *a priori* knowledge of either. There are also numerous scenarios where unexpected behavior, structure changes, or software crashes impact browser automation functionality. For instance, at any moment during the execution of a module there might be an unexpected popup (e.g., an alert). This can block all other functionality, such as fetching and interacting with elements in the page. Moreover, current error raising and handling support can lead to ambiguous states; e.g., when Selenium's Chromedriver crashes (which is a common issue) a `TimeoutException` might be raised, which is also what happens when a website actually times out. Thus, we need a way to handle such obstacles efficiently whenever they occur without aborting and restarting the whole process. Finally, while other well-designed options exist, e.g., Selenium-based OpenWPM [40], we find that they focus on the browser setup, management and synchronization parts of automation, with little focus on dynamic interaction (e.g., element clicking, form submission) which is a critical aspect of our study. In addition, while Puppeteer [16] does offer interaction functionality, it suffers from the same robustness issues as Selenium, which our system tackles (e.g., element staleness, crash recovery, robust error handling). Moreover, Puppeteer is specifically designed for Chrome/Chromium, while we aim to make our automation component compatible with different browsers.

To address these limitations we develop XDriver, a custom browser automation tool designed for security-oriented tasks that offers improved fault-tolerance during prolonged black-box interactions with web apps. XDriver is built on top of Selenium and the official Chrome and Firefox WebDrivers [11, 13], and will be made open source. We extend Selenium's high level WebDriver class to enhance our system's robustness by addressing the aforementioned challenges in a way that is *transparent* to the caller scripts. Due to space limitations here we present the most prominent exceptions and how our system handles them, as well as a number of useful auxiliary mechanisms we implement. Our extensions amount to approximately 1,500 lines of code.

Invocation. XDriver extends Selenium's WebDriver class and declares a custom `invoke` method which accepts a parent class method as an argument (e.g., `WebDriver.find_element`) and an arbitrary number of named and unnamed arguments. `Invoke` then calls the passed method in a `try-except` block, catches any raised exception and either calls the appropriate exception handler or returns a default value. XDriver then overrides all of WebDriver's methods to call their parent class counterparts via `invoke`.

Element staleness. As our auditing requires prolonged, multi-phase interaction with web apps, page elements frequently become "stale", which creates complications and can lead to crashes. XDriver is designed to handle such cases transparently and robustly. All interactions start by fetching a page element, e.g., based on the `id` attribute, and proceed with processing that element. If in the meantime this element is deleted or, more commonly, an asynchronous page load or redirection occurs, a `StaleElementReferenceException` is raised when interacting with the element, indicating that it is no longer attached to the DOM. However, while from a user's perspective the element might still be present in the page, from Selenium's point of view it is a new element under a new object reference, with no relation to the previously returned element. To handle this, when a `find_element_by` method is invoked, the returned element's object reference is stored as the key in a hash table, with a tuple containing the invoked method and its arguments as the value. Then, whenever such an exception occurs, the given element's reference is retrieved from that hash table and XDriver attempts to re-fetch it by invoking the stored method. If the element is found, the **old** element's object is updated transparently with the newly returned element, and the initial requested operation that raised the exception is retried. Otherwise, the exception is raised since the element truly does not appear in the page.

Handling crashes and timeouts. When Chromedriver or some other component (e.g., intermediate proxy) crashes and a `TimeoutException` is raised, our XDriver module detects the crash, transparently restores the browser instance and state and eventually fulfills any module's request that was interrupted by the crash. Specifically, it launches a new browser instance, reloads the current browser profile to maintain state and updates its own object reference with that of the new one, so as to transparently update all references of the driver held by the framework modules. It also obtains the last known URL and retries the interrupted operation. The `StaleElementReferenceException` handler is extremely useful in this case, since all retrieved web element objects will have become stale due to the browser reboot.

Auxiliary mechanisms. Several other mechanisms have been implemented in XDriver, which further aid our main framework's functionality, such as a *retry mode*, a configurable built-in crawler and our form-filling functionality described previously. Due to space constraints we provide more details in the Appendix. Overall, all of the above enhancements allow for more fault-tolerant interaction with web apps, reduce code complexity, and allow our main framework modules to focus on their specific tasks.

Security mechanisms. Another important feature is the detection and evaluation of security mechanisms pertinent to our study. HTTP Strict Transport Security (HSTS) instructs a user's browser to connect to the HSTS-enabled domain only over HTTPS for a specified amount of time, even if an explicit HTTP URL is followed or typed in the address bar by the user. While this seems fairly straightforward to deploy, domains often do so incorrectly or partially [52, 76, 77]). To evaluate deployment and detect misconfigurations, our module first checks whether the domain is in the Chromium preload list [12] and, if not, uses a passive proxy to capture the target website's redirection flow from its HTTP endpoint

to HTTPS. For each redirection, it stores the HSTS policy (if one is sent) and assesses whether the (sub)domain is indeed protected. Our module detects all the misconfigurations and errors presented in [52]. We note that while we implement mechanisms that are relevant to this work, XDriver's modular design streamlines the addition of other security mechanisms.

4 EXPERIMENTAL EVALUATION

We experimentally evaluate our black-box auditing framework and present our findings from the largest study on cookie-based authentication and authorization flaws in the wild.

Datasets. We use two different versions of the Alexa Top 1 million list. The first dataset was fetched on 09/14/2017; this dataset was useful for guiding the design and implementation of our framework. However, since recent work has revealed that domain ranking lists exhibit significant fluctuation even within short periods of time [74], we also obtained a second up-to-date version on 05/07/2019, when it was time to conduct the final evaluation. All the experiments presented here were conducted between May-October 2019 on a combined dataset that included a total of 1,585,964 unique domains.

Workflow statistics. One of our main goals is the ability to conduct automated black-box auditing of modern web apps without knowledge of their structure, access to the source code, or input from developers. The complexity and often ad-hoc nature of web development render this a challenging task, and various obstacles can prevent the successful completion of a given module. Figure 4 in the Appendix provides statistics on the number of domains for which each phase of our workflow was successful. In general, our auditing modules are highly effective, successfully completing their analysis for 93-98% of the domains they handle. Automated account creation presents the most considerable obstacle; namely, out of the 168,594 domains for which we identified a signup option, we successfully registered and logged into 13.7% of them, while in 2,066 cases our system managed to login via SSO, out of which 346 were a fallback after a failed signup attempt. It is worth noting that for domains where we detected a signup option but were not able to create an account, 19,491 (~13.8%) embedded Google's reCaptcha. Yet our framework is still able to create accounts on *25,242* domains, accounting for almost 12% of the domains for which we have identified a signup option – for comparison, prior related studies analyzed 25 [77] and 149 [64] domains. In studies with a different focus, Zhou and Evans used SSO to audit 1,621 domains for SSO implementation flaws, while DeBlasio et al. [36] explored the risk of password reuse by creating accounts in over 2,300 domains. In other words, our study is several orders of magnitude larger than prior studies with a similar focus, and at least one order of magnitude larger than studies that employed some form of automated account creation. We provide more details on our system's effectiveness and false negative rates in the Appendix.

Cookies. Audited domains set an average of 14.02 cookies, while susceptible domains set 1.21 authentication cookies and have 1.1 authentication combinations on average. In Table 1, we show the number of domains that expose their authentication cookies, i.e., do not protect them with the corresponding cookie attributes.

Eavesdropping. We find that 12,014 unique domains do not protect their authentication cookies with the *secure* flag, even though 1,815 of those set the flag for at least one of their cookies. However, web apps might make use of HTTP-Strict-Transport-Security (HSTS), which can prevent the leakage of those, otherwise exposed cookies. Merely checking for the presence of HSTS headers in the web app's responses is not sufficient, since prior studies have found that developers often deploy HSTS incorrectly [52, 76] or do not adequately protect their entire domain [77]. As such, our framework includes a module for evaluating the correctness and coverage of HSTS deployment for domains that are vulnerable to eavesdropping (the other attacks are not affected by HSTS).

Table 1: Number of unique domains that do not adequately protect their cookies from specific attacks.

Attack	# of Domains (%)
Eavesdropping	12,014 (48.43%)
No HSTS	10,495 (87.36%)
HSTS Preloaded	64 (0.53%)
Full HSTS	188 (1.56%)
Faulty HSTS	
- *Protected*	736 (6.13%)
- *Vulnerable*	426 (3.55%)
Final Vulnerable	10,921 (90.9%)
JS cookie stealing	5,680 (22.9%)
Total	**12,484 (50.33%)**

We find that the situation has not improved much compared to prior studies, as the vast majority of domains do not deploy HSTS. While flawed HSTS deployment remains common, we find that 63.3% of the domains that have a faulty deployment do manage to prevent our cookie hijacking attacks. This is because the set of (sub)domains the auth cookies are sent to are protected by HSTS. For instance, if *example.com* deploys HSTS properly on the *www* subdomain, but leaves the base domain unprotected, and at least one auth cookie has its *domain* attribute set to *www.example.com*, then there is no way for an eavesdropper to retrieve this cookie. The most common misconfiguration is not enabling HSTS on the base domain (696 domains), out of which 143 attempted to set HSTS over HTTP. The remaining domains, while properly setting HSTS on their main domain, did not use the *includeSubdomains* directive, thus potentially leaving certain subdomains exposed. We also find that out of the remaining domains only 99 employ CSP's `upgrade-insecure-requests` directive. While this reduces the attack surface, these domains remain vulnerable since this mechanism does not upgrade top-level navigational requests from third-party sites or the initial request (e.g., when a user opens a new tab and visits a site). Overall, 10,921 domains are vulnerable and expose cookies to eavesdroppers even when accounting for the presence of relevant security mechanisms. We further correlate these domains with the Single Sign On data released by [44] and found that four of these domains are also SSO identity providers (Amazon, Bitly, DeviantArt, GoodReads) and have at least 1,346 unique relying parties, out of which 138 have been audited by our system; 87 were found secure and 51 vulnerable to at least one of our attacks.

JS cookie stealing. We find that users face a considerable threat due to their authentication cookies being accessible via (malicious) JavaScript, as a total of 5,680 domains do not protect them with

Table 2: Number of domains for different values of authentication cookies and combinations of authentication cookies.

	1	2	3	4	5	6	7
Auth combos	10,878	1,110	39	10	3	-	-
Auth cookies	9,912	1,700	364	54	7	2	1

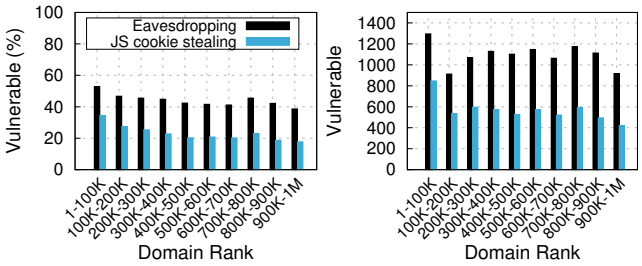

Figure 2: Percentage (left) and absolute number (right) of vulnerable domains per ranking bin.

Table 3: Personal user data that can be obtained by attackers.

Data	Source	Cookies	Storage	URL	Total (%)
Email	6,894	776	174	51	7,130 (61)
Email hash	885	68	10	0	930 (7.98)
Fullname	4,287	198	170	44	4,330 (37)
Firstname	648	58	8	10	686 (5.9)
Lastname	618	86	19	13	665 (5.7)
Username	1,856	339	48	175	1,956 (16.7)
Password	2	20	0	0	22 (0.19)
Pswd hash	12	57	0	0	68 (0.6)
Phone	1,594	8	7	2	1,598 (13.7)
Address	656	0	0	1	656 (5.6)
VAT	17	0	0	0	17 (0.15)
Workplace	540	3	3	1	543 (4.6)
Total (%)	9,122 (78)	1,236 (10.6)	314 (2.7)	290 (2.5)	

the *httpOnly* flag. Our framework's analysis of those domains reveals that 5,099 include at least one embedded 3rd party script (i.e., not isolated in an iframe) that runs in the 1st party's origin and has "permission" to read the user's 1st party cookies. These are fetched from 2,463 unique 3rd party domains. To make matters worse, only 239 of those use the Subresource Integrity (SRI) feature [15] to prevent the manipulation of fetched scripts, and only one domain protects all loaded scripts. Similarly to [31], we find that all SRI-protected scripts are libraries (e.g., jquery). It is important to emphasize that this attack explores the potential threat from compromised or rogue 3rd parties, and that our numbers do not reflect active attacks currently underway in the wild. While our study's focus is not on detecting malicious scripts actually stealing users' cookies, we consider this an interesting future direction.

We emphasize that the 5,680 domains are not necessarily vulnerable to session hijacking through XSS, since other prevention mechanisms might be in place. For instance, Web Application Firewalls (WAFs) [38, 54] or Content Security Policies (CSP) [92] could be deployed to mitigate XSS attacks which could also prevent cookie stealing. Nonetheless, recent work has shown that even such defense mechanisms can be bypassed [57]. As such, our findings constitute an upper bound for web apps that are vulnerable to cookie-stealing via XSS. Nonetheless, while adoption of *httpOnly* is not as limited as in the past [95], it remains an important issue.

Auth combos. Table 2 breaks down the AuthCookies results and reports the number of domains with the corresponding number of authentication cookies and combinations. An interesting observation is that 435 of the domains that have *more than one* combination contain at least one secure combination among them, yet remain susceptible to attacks due to other combination(s) being exposed. This highlights how the ever-increasing complexity in web apps leads to authorization flaws. We also find that 76 domains contain cookie combinations that are correctly detected by our approach for which the algorithm from [64] returns incorrect results.

Popularity. We break down the vulnerable domains based on their Alexa rank in Figure 2. In general, our framework detects more vulnerable domains in the highest ranking bin. This can be partially attributed to popular websites being more likely to support account creation (we find twice as many such domains in the most popular bin compared to the least popular one), while the process succeeds for roughly 11 – 13% of domains across all bins. We also break down the vulnerable websites based on their categories (e.g., online shopping) in the Appendix.

Privacy leakage. In Table 3, we break down the personal or sensitive information that an attacker can acquire upon successfully hijacking a user's cookies, as detected by our PrivacyAuditor module. We also report the total number of domains leaking such information, grouped per sensitive field (e.g., email) and also based on the source of leakage (e.g., page source). While a domain might appear in different columns of the same sensitive field, or different rows of the same source of leakage, it is only counted once in the corresponding totals. In general, we find that the page's source is the most common avenue of exposure, but passwords are typically exposed through cookies. Furthermore, 59 out of the 68 hashed passwords detected by our system are MD5 hashes, which do not offer much protection against offline brute-forcing attacks. In practice, the attacker could potentially recover the password and obtain full control over the victim's account in those services; password reuse [9, 69] can result in attackers accessing accounts in other services as well. Apart from common identifiers like emails and usernames, many domains expose highly sensitive data like home addresses and phone numbers. Overall, an abundance of data is exposed that can be used for doxxing [79], and a plethora of scams including targeted phishing [48] and identity theft [21].

System performance. In Figure 3 we show the total time in seconds required by each module in our framework. Since some modules might fail for certain domains, the different CDFs have been calculated using their corresponding totals. The total time required for auditing websites for attacks (i.e., all modules up to CookieAuditor) is denoted as *Total Attack*. The total time required for the analysis including the execution of AuthCookies and PrivacyAuditor is denoted as *Full Analysis*. We find that our framework's performance is suitable for large-scale studies as half of the domains can be completely audited within 5 minutes and 90% in less than 17

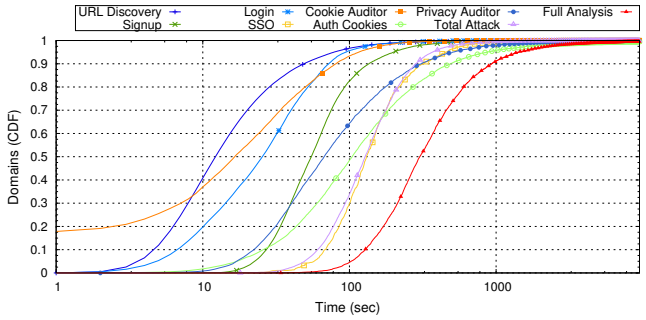

Figure 3: Time required by each module of our system.

minutes. While certain domains in the long tail of the distribution require considerably more time, this is typically due to latency issues with their specific servers. While Webdriver crashes can affect performance, our XDriver optimizations minimize their impact by transparently recovering the browser's state.

Popular domains. While our main goal is to automatically explore the feasibility of cookie hijacking at scale, popular domains are of particular interest because they are used by hundreds of millions of users and, thus, can have a greater impact if vulnerable. Considering that our framework's entire workflow is fully automated and that app-agnostic account setup is extremely challenging, we opt to manually assist with the account setup for a subset of the most popular domains. Specifically, we consider the top 1K domains, where we identified 698 account-based websites. Out of those, 95 were already fully handled by our framework. For the rest, we manage to manually create accounts in 206 domains, which we provided to our framework to complete the automated auditing process. The remaining domains either protected their login forms with reCAPTCHAs, detected the presence of our webdriver, or requested information during signup that we were unable to provide (e.g., phone numbers for SMS verification, valid SSN etc.). Moreover, for 45 websites our Login Oracle could not disambiguate between being logged in and logged out; when sending a HTTP request without any cookies our account would still appear to be logged in. In total, we audited 301 popular websites (the additional 206 domains were not included in our previously reported numbers, thus, pushing our total analysis to over 25K domains).

We find that 149 are vulnerable to eavesdropping, 46 of which were fully handled by our framework. Only 10 domains deploy HSTS effectively, while another 30 (20.13%) use HSTS but remain susceptible due to faulty deployment. For JS cookie stealing, 115 domains were found susceptible and 104 include at least one embedded 3rd party script (from 266 domains) – only five make use of SRI. Overall, 57.81% of the domains do *not* provide adequate defenses, which is alarming considering their massive user base.

Hijacking validation. To manually validate our results and ensure that an attacker can actually access victims' accounts, we conduct an exploratory experiment on domains that were *fully handled* by our framework. We randomly select ten and hand-pick another ten domains out of Alexa's Top-1K, and randomly select another ten from the remaining domains, and simulate cookie hijacking attacks. We setup a browser instance where we log in the website and capture all cookies that are exposed depending on the threat model. Next, we launch a new browser with different characteristics (user agent etc.) on a different machine, in a different network subnet, where we include the stolen cookies and visit the website. We manually interact with the website to detect the extent of access the attacker obtains. We do not set a time limit; instead we opt for an exhaustive approach where we try to identify all user-specific functionality that should be tested. We detail our findings in the Appendix. For the Top-1K random subset, we get full account access for seven domains (i.e., all tested operations succeeded), and partial access for three domains. For the other random subset we get full access in nine out of ten domains. Indicatively we can view and modify account settings, preferences, shopping lists, orders and subscriptions and post comments. In five of all the domains we could also change the user's password *without* knowledge of the current password. For the manually selected popular domains, we get full access in five domains, partial access in four.

This highlights a significant advantage of cookie-based account hijacking over credential-based (e.g., phishing): additional fraud-detection checks employed during login [24] (e.g., IP geo-location [71], comparison of browser fingerprints [50]) are ommitted because the cookies are part of a session that has already been verified as legitimate (i.e, when the victim logged in). While certain attackers can pass geo-location checks (e.g., using an IP address near the user's location [67]), deceiving browser-based security checks is significantly more challenging. While spoofing the victim's fingerprints has been theorized [19] it has not been demonstrated in practice. Surprisingly, throughout all our experiments we identified only one domain (Cloudflare) where we could not access the victim's account from the attacker's machine, indicating additional machine-specific checks that we have not come across in any other domain.

5 DISCUSSION

Automated account creation. Our experimental evaluation revealed that automatically creating accounts is a significant challenge. While our current implementation allowed us to audit orders-of-magnitude more domains than prior manual studies [30, 77], we plan to explore the adoption of more sophisticated heuristics that automatically infer the predicates of account generation in a specific web app and create corresponding inputs. Automatically detecting and parsing error messages returned by the app can be used as feedback for inferring which form fields' format is violated. This, however, is a challenging task as, again, web developers are not constrained to a specific format or structure for returning such messages. Furthermore, each form input variation requires a form submission, which can lead to a significant impact to the overall performance and also trigger anti-bot mechanisms. Certain mandatory resources can also prevent our system from completing the process, e.g., an app may require a valid phone number in a specific country. While attackers can leverage "shady" phone providers [86], this remains an important obstacle for researchers.

Privacy leakage inference. Our system evaluates the leakage of personal or sensitive user information by detecting specific identifiers. In practice, information can be implicitly leaked, e.g., personalized results in search engines or e-commerce systems can reveal sensitive data (typically exposed through site-specific functionality). As part of our future work, we plan to explore the use

of user-action templates that are based on the website's category (e.g., search engine, e-commerce), intended to elicit personalized results. Additionally, it is possible that some user information might already be publicly available on the same or a different website and, thus, the detected identifiers do not constitute actual *leakage*. While leakage can be highly contextual (e.g., a user's email address being publicly available in general versus a local eavesdropper being able to match that person to their email address) we consider this an interesting challenge and plan to explore the feasibility of detection schemes that disambiguate between public and private information.

Countermeasures, disclosure, ethics. Our framework discovered flaws that are exposing millions of users to significant threat. We emphasize that *no user accounts were affected* during our experiments – we only used test accounts. It is also crucial that developers are informed of our findings and address them. While the adoption of cookie security flags is more straightforward, correctly deploying HTTPS and HSTS will likely be more challenging for developers [32, 51–53]. For disclosure we leveraged the insight provided by prior work [58, 73, 84] and sent direct notifications to the affected domains for which we could find a valid contact email address. Specifically, we initially collected `security.txt` files [10], that typically include such contact points. This method proved to be the most ineffective, as such files are not widely adopted, i.e., only 23 domains had them. We then used an off-the-shelf email harvester tool for search engines [8]. Next, we crawled the websites starting from their home page and visiting all contact related URLs, as well as the top 10 first level links. We also collected each domain's WHOIS record and searched for registered abuse addresses. We filtered all collected email addresses to ensure that they belong to the susceptible domain, so as to avoid sending our security-sensitive findings to unrelated parties. Overall, this process yielded 5,373 email addresses which we used for notification. For the remaining domains we sent our notification to standard aliases (`security, abuse, webmaster, info`) [73, 84]. We also manually searched for contact points for all domains we explicitly name in the paper (apart from 2 that did not have a contact email or form). For the notification process we used an institutional email address to increase credibility and provided additional details and remediation advice to all websites that responded. All the responses we received acknowledged our findings, except one case where the developer persistently misunderstood the technical aspects of cookie hijacking. While we followed a best-effort approach to directly notify affected domains, it is infeasible to do so for all of them. Thus, we will also setup a notification service where developers can obtain our reports after proving ownership of a given domain.

HSTS issue. During our experiments we uncovered an unexpected behavior in Chrome with HSTS preloading; we observed that it did not work as expected in slightly older Chrome versions and the initial request to a preloaded domain was, in fact, over HTTP. After communication with the Chromium team they informed us that their policy dictates that *any Chrome version more than 70 days old does not enforce HSTS preloading* because such hardcoded information is considered stale. This has significant implications for users that do not update their software on time, which is common behavior [62, 88, 91]. To the best of our knowledge this issue with HSTS has not been mentioned in prior studies.

Code sharing. Our browser automation tool will be made open source as it can facilitate various research projects, especially those focused on Web security. However, publicly releasing our automated account creation modules poses a significant risk, as they are directly applicable to a plethora of real world attacks and could be misused for malicious purposes; the capabilities of our system far surpass the capabilities of such tools typically found in underground markets [68]. To that end, and to further contribute to the community, we have opted to make these modules available to vetted researchers upon request.

6 RELATED WORK

Cookies and sessions. Several prior studies have explored certain aspects of authentication and authorization flaws in web apps. Sivakorn et al. [77] manually audited 25 popular domains (and their respective mobile apps and browser extensions). Calzavara et al. [30] recently implemented black-box strategies for identifying session integrity flaws using a browser extension, and audited 20 popular websites where they found several vulnerabilities under different threat models. However, the most challenging parts of the process are not automated and app-agnostic (e.g. account creation, status oracles), rendering large-scale deployment and analysis infeasible. Neither of these studies included the JavaScript-based threats that we explore. In another work, Calzavara et al. [27] conducted a large-scale study on TLS vulnerabilities that can enable session hijacking. Kwon et al. [56] exploited the shortcomings of a specific TLS cipher suite and proved that, under certain assumptions, it is possible to disable cookie attributes in HTTPS traffic. Finally, Jonker et al. [46] proposed a system for automated login that can enable post-login studies. However, their system does not handle account creation which is the most challenging process.

While these studies provide useful insights, they are inherently small-scale, require significant manual effort, or are complimentary to our work as they focus on different problems that enable session hijacking (e.g. TLS vulnerabilities). In contrast, our work achieves orders of magnitude larger coverage of audited domains, analyzes the root causes of such attacks and further explores the use of other defense mechanisms, as well as the privacy leakage users face. Orthogonal to our work are prior studies that proposed defenses against session hijacking attacks [17, 23, 28, 29, 34, 66, 87].

Cookies and browsers. Singh et al. [75] built a framework for analyzing the usage of browser features in the wild and detecting browsers' access-control flaws, e.g., *secure* cookies being sent over HTTP. Franken et al. [43] evaluated how different browsers and anti-tracking extensions handle third party requests and showed that cookie-bearing third party requests can be leaked by all browsers, even in the presence of protection mechanisms like *sameSite* cookies. Zheng et al. [94] studied how cookie integrity can be diminished by various adversaries due to specification violations in browser and server-side implementations, and demonstrated practical attacks on popular websites. Cookies are also commonly used for tracking, and Cahn et al. [25] explored their use through empirical large-scale measurements and reported the prevalence of third party cookies. Moreover, Englehardt et al. [41] showed that a passive eavesdropper can exploit third-party cookies to reconstruct up to 74% of a user's browsing history. These studies are orthogonal to

our work since we do not examine browser shortcomings in terms of leaking cookies that can lead to session hijacking; instead, we explore the effects of developer malpractices which, however, can be exacerbated by browsers' inability to properly handle cookies.

Security headers and policies. Chen et al. [32] examined the CORS specification, and browser/server-side implementations, and found security issues in all cases, several previously unknown, which could even lead to data theft and account hijacking. Kranch et al. [52], performed the first in-depth study on HSTS and HPKP, identifying various misconfigurations in preloaded domains as well as Alexa's Top 1M. Mendoza et al. [63] examined HTTP header inconsistencies between websites and their mobile counterparts, and reported cases of mismatches in set cookie flags. Stock et al. [83] presented a longitudinal study on the Web's evolution and, among other things, measured the adoption of security mechanisms. While we leverage certain aspects of these studies [52], our goal is not to evaluate these mechanisms in a generic context; instead, we evaluate the deployment of the relevant mechanisms and how they either enable or prevent session hijacking specifically.

SSO and sessions. Several studies have focused on SSO-related vulnerabilities. Zhou and Evans [96] implemented SSOScan, a tool that detected vulnerabilities in Facebook's SSO scheme and found that of the 1,660 audited websites, 146 leaked credentials and 202 misused them. While SSOScan handles SSO authentication flows, several issues render it unsuitable for our study; however, we do incorporate one of their heuristics in our framework. Mainly, our system needs to handle non-SSO websites, which account for the vast majority of sites we audit (~92%); this necessitates more advanced and robust form-handling capabilities to address the more complex and diverse nature of non-SSO registration. For instance, SSOScan only uses an input element's `id` and `name` attributes to infer its type, while we leverage all of its attributes, dedicated `label` elements, as well as the input's preceding text as possible labels. Also, since SSOScan processes all input elements of a page at once, there is a chance that it uses an unrelated submit button; we avoid this by processing each form separately. Finally, if SSOScan is not able to locate a conventional submit button it will not be able to submit the form, while our system attempts to do so via Selenium's `submit` method. For SSO workflows, we identified several challenges that SSOScan was not able to handle. For instance, SSOScan's oracle relies on the SSO login button not being displayed after logging in, which, as aforementioned, is not always the case. We address this by separating our SSO and SSO Login oracles. In addition, SSOScan operates only on the homepage for locating candidate elements, while we employ a crawling approach to obtain better coverage. Finally, their tool only considers English sites.

Fett et al. [42] proposed and evaluated a formal model of the OAuth 2.0 protocol. Wang et al. [90] employed differential testing to identify logic flaws in SSO implementations and found several popular IdPs and RPs to be vulnerable. Calzavara et al. [26] implemented a lightweight browser-side monitor for web protocols (e.g., OAuth) that uses formalized protocol specifications to enforce confidentiality and integrity checks. Yang et al. [93] used symbolic execution to audit SSO SDK libraries and discovered seven classes of vulnerabilities in 10 SDKs. Zuo et al. [98] proposed a tool to identify vulnerable authorization implementations in mobile apps, which relied on differential traffic analysis for identifying fields of interest in exchanged messages. They used Facebook's SSO to audit ~5K apps (306 were vulnerable). They also explored data leakage in mobile apps [97] that use a cloud-based back-end, stemming from key misuse and authorization flaws. However, their leakage exploration focuses on a very limited set of information and they manually setup an account on only 30 apps. Ghasemisharif et al. [44] demonstrated that SSO magnifies the scale and stealthiness of account hijacking, while rendering remediation impossible in most cases. While we use SSO as an alternative way for registering test accounts, identifying flaws in SSO implementations and specifications is not our objective. Nonetheless, these studies shed light on a different problem that can lead to session hijacking.

7 CONCLUSIONS

We developed a completely automated auditing framework for web apps that detects authentication and authorization flaws that revolve around the handling of cookies and stem from the incorrect, incomplete, or non-existent deployment of appropriate security mechanisms. Our framework is comprised of a series of modules that include novel mechanisms to differentially analyze web apps, assess the deployment of security mechanisms, and detect what user data is exposed. At the heart of our framework lies a custom browser automation tool designed for robust and fault-tolerant black-box interaction with web apps. We used our framework to conduct the largest study on session hijacking to date and audit 25K domains, leading to a series of alarming findings. Despite the increasing adoption of HTTPS, HSTS is rarely deployed (correctly or at all), and ~11K domains are vulnerable to eavesdropping attacks that enable partial or full access to users' accounts. Furthermore, 23% of domains are susceptible to cookie hijacking through JavaScript, the majority of which also include third party scripts that execute in the first party origin. We also demonstrated how hijacked cookies allow access to sensitive and personal user information though various avenues of exposure. Our study reveals that cookie hijacking remains a severe and pressing threat, as adoption of appropriate security mechanisms remains limited and developers continue to struggle with correct deployment. In an effort to shed light on the scale of this threat, guide remediation efforts, and further incentivize the adoption of security mechanisms, we have managed to directly notify ~43% of the affected domains and will also deploy a service for providing reports.

ACKNOWLEDGEMENTS.

We would like to thank the anonymous reviewers, and our shepherd Giancarlo Pellegrino, for their valuable feedback. This work was partially supported by the National Science Foundation under contract CNS-1934597. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government. This work has also received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830927 (CONCORDIA) and under grant agreement No 833456 (GUARD).

REFERENCES

[1] 2017. Open Web Application Security Project - The OWASP Top 10. https://www.cloudflare.com/learning/security/threats/owasp-top-10/.

[2] 2018. Dashlane - World Password Day: How to Improve Your Passwords. https://blog.dashlane.com/world-password-day/.

[3] 2018. Four cents to deanonymize: Companies reverse hashed email addresses. https://freedom-to-tinker.com/2018/04/09/four-cents-to-deanonymize-companies-reverse-hashed-email-addresses/.

[4] 2018. WIRED - a new Google+ blunder exposed data from 52.5 million users. https://www.wired.com/story/google-plus-bug-52-million-users-data-exposed/.

[5] 2018. WIRED - the Facebook hack exposes an Internet-wide failure. https://www.wired.com/story/facebook-hack-single-sign-on-data-exposed/.

[6] 2019. Ars Technica - DHS: Multiple US gov domains hit in serious DNS hijacking wave. https://arstechnica.com/information-technology/2019/01/multiple-us-gov-domains-hit-in-serious-dns-hijacking-wave-dhs-warns/.

[7] 2019. Cisco Talos - DNS Hijacking Abuses Trust In Core Internet Service. https://blog.talosintelligence.com/2019/04/seaturtle.html.

[8] 2019. Email addresses harvester. https://github.com/maldevel/EmailHarvester.

[9] 2019. Google / Harris Poll - Online Security Survey. https://services.google.com/fh/files/blogs/google_security_infographic.pdf.

[10] 2020. https://securitytxt.org/.

[11] 2020. ChromeDriver - WebDriver for Chrome. https://sites.google.com/a/chromium.org/chromedriver/downloads.

[12] 2020. The Chromium Projects - HTTP Strict Transport Security. https://www.chromium.org/hsts.

[13] 2020. Geckodriver. https://github.com/mozilla/geckodriver.

[14] 2020. McAfee - Customer URL Ticketing System. https://trustedsource.org/en/feedback/url.

[15] 2020. MDN Web Docs - Subresource Integrity. https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity.

[16] 2020. Puppeteer. https://developers.google.com/web/tools/puppeteer.

[17] Ben Adida. 2008. Sessionlock: Securing Web Sessions Against Eavesdropping. In *Proceedings of the 17th International Conference on World Wide Web*.

[18] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 1–10.

[19] Furkan Alaca and Paul C Van Oorschot. 2016. Device fingerprinting for augmenting web authentication: classification and analysis of methods. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 289–301.

[20] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *27th USENIX Security Symposium (USENIX Security '18)*. 377–392.

[21] Leyla Bilge, Thorsten Strufe, Davide Balzarotti, and Engin Kirda. 2009. All your contacts are belong to us: automated identity theft attacks on social networks. In *Proceedings of the 18th international conference on World wide web*. ACM, 551–560.

[22] Kevin Bock, Daven Patel, George Hughey, and Dave Levin. 2017. unCaptcha: A Low-Resource Defeat of reCaptcha's Audio Challenge. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*.

[23] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. 2015. CookiExt: Patching the browser against session hijacking attacks. *Journal of Computer Security* (2015).

[24] Elie Bursztein, Borbala Benko, Daniel Margolis, Tadek Pietraszek, Andy Archer, Allan Aquino, Andreas Pitsillidis, and Stefan Savage. 2014. Handcrafted fraud and extortion: Manual account hijacking in the wild. In *Proceedings of the 2014 conference on internet measurement conference*. ACM, 347–358.

[25] Aaron Cahn, Scott Alfeld, Paul Barford, and S. Muthukrishnan. 2016. An Empirical Study of Web Cookies. In *Proceedings of the 25th International Conference on World Wide Web (WWW '16)*.

[26] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. 2018. WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association.

[27] Stefano Calzavara, Riccardo Focardi, MatÃžÅą Nemec, Alvise Rabitti, and Marco Squarcina. 2019. Postcards from the Post-HTTP World: Amplification of HTTPS Vulnerabilities in the Web Ecosystem. In *2019 IEEE Symposium on Security and Privacy*.

[28] Stefano Calzavara, Riccardo Focardi, Marco Squarcina, and Mauro Tempesta. 2017. Surviving the Web: A Journey into Web Session Security. *Comput. Surveys* (2017).

[29] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2018. Sub-session hijacking on the web: Root causes and prevention. In *Journal of Computer Security*.

[30] Stefano Calzavara, Alvise Rabitti, Alessio Ragazzo, and Michele Bugliesi. 2019. Testing for Integrity Flaws in Web Sessions. In *Computer Security - th European Symposium on Research in Computer Security, ESORICS 2019*.

[31] Bertil Chapuis, Olamide Omolola, Mauro Cherubini, Mathias Humbert, and Kévin Huguenin. 2020. An Empirical Study of the Use of Integrity Verification Mechanisms for Web Subresources. In *Proceedings of The Web Conference 2020 (WWW '20)*. Association for Computing Machinery.

[32] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. 2018. We Still Don't Have Secure Cross-Domain Requests: an Empirical Study of CORS. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association.

[33] Sandy Clark, Stefan Frei, Matt Blaze, and Jonathan Smith. 2010. Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities. In *Proceedings of the 26th annual computer security applications conference*. ACM, 251–260.

[34] Italo Dacosta, Saurabh Chakradeo, Mustaque Ahamad, and Patrick Traynor. 2012. One-time Cookies: Preventing Session Hijacking Attacks with Stateless Authentication Tokens. *ACM Trans. Internet Technol.* (2012).

[35] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. 2009. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. In *Proceedings of the 18th Conference on USENIX Security Symposium*. USENIX Association, 267–282.

[36] Joe DeBlasio, Stefan Savage, Geoffrey M Voelker, and Alex C Snoeren. 2017. Tripwire: inferring internet site compromise. In *Proceedings of the 2017 Internet Measurement Conference*. ACM, 341–354.

[37] Levent Demir, Amrit Kumar, Mathieu Cunche, and Cedric Lauradoux. 2017. The pitfalls of hashing for privacy. *IEEE Communications Surveys & Tutorials* 20, 1 (2017), 551–565.

[38] Lieven Desmet, Frank Piessens, Wouter Joosen, and Pierre Verbaeten. 2006. Bridging the gap between web application firewalls and web applications. In *Proceedings of the fourth ACM workshop on Formal methods in security*. ACM, 67–77.

[39] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, Bellevue, WA, 523–538. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe

[40] Steven Englehardt and Arvind Narayanan. 2016. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of ACM CCS 2016*.

[41] Steven Englehardt, Dillon Reisman, Christian Eubank, Peter Zimmerman, Jonathan Mayer, Arvind Narayanan, and Edward W. Felten. 2015. Cookies That Give You Away: The Surveillance Implications of Web Tracking. In *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee.

[42] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2016. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[43] Gertjan Franken, Tom Van Goethem, and Wouter Joosen. 2018. Who Left Open the Cookie Jar? A Comprehensive Evaluation of Third-Party Cookie Policies. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association.

[44] Mohammad Ghasemisharif, Amrutha Ramesh, Stephen Checkoway, Chris Kanich, and Jason Polakis. 2018. O Single Sign-Off, Where Art Thou? An Empirical Analysis of Single Sign-On Account Hijacking and Session Management on the Web. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association.

[45] Shashank Gupta and Brij Bhooshan Gupta. 2017. Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of System Assurance Engineering and Management* 8, 1 (2017), 512–530.

[46] B. Krumnow H. Jonker, S. Karsch and M. Sleegers. 2020. Shepherd: A Generic Approach to Automating Website Login. In *Proceedings of the 2020 Workshop on Measurements, Attacks, and Defenses for the Web*.

[47] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. 2015. Vetting SSL usage in applications with SSLint. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 519–534.

[48] Markus Huber, Martin Mulazzani, Edgar Weippl, Gerhard Kitzler, and Sigrun Goluch. 2010. Exploiting social networking sites for spam. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 693–695.

[49] Muhammad Ikram, Rahat Masood, Gareth Tyson, Mohamed Ali Kaafar, Noha Loizon, and Roya Ensafi. 2019. The chain of implicit trust: An analysis of the web third-party resources loading. In *The World Wide Web Conference*. ACM, 2851–2857.

[50] Hugo Jonker, Benjamin Krumnow, and Gabry Vlot. 2019. Fingerprint Surface-Based Detection of Web Bot Detectors. In *European Symposium on Research in Computer Security*. Springer, 586–605.

[51] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. 2018. Coming of age: A longitudinal study of tls deployment. In *Proceedings of the Internet Measurement Conference 2018*. ACM, 415–428.

[52] Michael Kranch and Joseph Bonneau. 2015. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*.

[53] Katharina Krombholz, Wilfried Mayer, Martin Schmiedecker, and Edgar Weippl. 2017. " I Have No Idea What I'm Doing"-On the Usability of Deploying HTTPS. In *26th USENIX Security Symposium (USENIX Security 17)*. 1339–1356.

[54] Tammo Krueger, Christian Gehl, Konrad Rieck, and Pavel Laskov. 2010. TokDoc: A Self-healing Web Application Firewall. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*.

[55] Deepak Kumar, Zane Ma, Zakir Durumeric, Ariana Mirian, Joshua Mason, J Alex Halderman, and Michael Bailey. 2017. Security challenges in an increasingly tangled web. In *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 677–684.

[56] H. Kwon, H. Nam, S. Lee, C. Hahn, and J. Hur. 2019. (In-)Security of Cookies in HTTPS: Cookie Theft by Removing Cookie Flags. *IEEE Transactions on Information Forensics and Security* (2019).

[57] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A. Vela Nava, and Martin Johns. 2017. Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM.

[58] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. 2016. You've Got Vulnerability: Exploring Effective Vulnerability Notifications. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association.

[59] Zhou Li, Kehuan Zhang, Yinglian Xie, Fang Yu, and XiaoFeng Wang. 2012. Knowing your enemy: understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 674–686.

[60] Moxie Marlinspike. 2009. New Tricks For Defeating SSL In Practice. *BlackHat DC* (Feb. 2009).

[61] Matthias Marx, Ephraim Zimmer, Tobias Mueller, Maximilian Blochberger, and Hannes Federrath. 2018. Hashing of personally identifiable information is not sufficient. *SICHERHEIT 2018* (2018).

[62] Arunesh Mathur, Nathan Malkin, Marian Harbach, Eyal Peer, and Serge Egelman. 2018. Quantifying Users' Beliefs about Software Updates. *CoRR* (2018). http://arxiv.org/abs/1805.04594

[63] Abner Mendoza, Phakpoom Chinprutthiwong, and Guofei Gu. 2018. Uncovering HTTP Header Inconsistencies and the Impact on Desktop/Mobile Websites. In *Proceedings of the 2018 World Wide Web Conference (WWW '18)*. International World Wide Web Conferences Steering Committee.

[64] Yogesh Mundada, Nick Feamster, and Balachander Krishnamurthy. 2016. Half-Baked Cookies: Hardening Cookie-Based Authentication for the Modern Web. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*. ACM.

[65] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 736–747.

[66] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. 2011. SessionShield: Lightweight Protection against Session Hijacking. In *Engineering Secure Software and Systems*, Úlfar Erlingsson, Roel Wieringa, and Nicola Zannone (Eds.). Springer Berlin Heidelberg.

[67] Jeremiah Onaolapo, Enrico Mariconti, and Gianluca Stringhini. 2016. What happens after you are pwnd: Understanding the use of leaked webmail credentials in the wild. In *Proceedings of the 2016 Internet Measurement Conference*. ACM, 65–79.

[68] Avanish Pathak. 2014. An analysis of various tools, methods and systems to generate fake accounts for social media. *Northeastern University Boston, Massachusetts December* (2014).

[69] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. 2017. Let's Go in for a Closer Look: Observing Passwords in Their Natural Habitat. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[70] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, Vol. 00. 615–632. https://doi.org/10.1109/SP.2017.27

[71] Iasonas Polakis, Marco Lancini, Georgios Kontaxis, Federico Maggi, Sotiris Ioannidis, Angelos D. Keromytis, and Stefano Zanero. 2012. All Your Face Are Belong to Us: Breaking Facebook's Social Authentication. In *Proceedings of the 28th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '12)*. ACM, New York, NY, USA, 399–408. https://doi.org/10.1145/2420950.2421008

[72] N. Ramasubbu, M. Cataldo, R. K. Balan, and J. D. Herbsleb. 2011. Configuring global software teams: a multi-company analysis of project productivity, quality, and profits. In *2011 33rd International Conference on Software Engineering (ICSE)*. 261–270.

[73] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. 2020. Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies. In *NDSS*.

[74] Quirin Scheitle, Oliver Hohlfeld, Julien Gamba, Jonas Jelten, Torsten Zimmermann, Stephen D. Strowes, and Narseo Vallina-Rodriguez. 2018. A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists. In *IMC*.

[75] Kapil Singh, Alexander Moshchuk, Helen J Wang, and Wenke Lee. 2010. On the incoherencies in web browser access control policies. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 463–478.

[76] Suphannee Sivakorn, Angelos D. Keromytis, and Jason Polakis. 2016. That's the Way the Cookie Crumbles: Evaluating HTTPS Enforcing Mechanisms. In *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society* (Vienna, Austria) *(WPES '16)*. ACM, 71–81.

[77] Suphannee Sivakorn, Jason Polakis, and Angelos D. Keromytis. 2016. The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information. In *In Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P '16)*.

[78] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. 2019. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *The World Wide Web Conference*. 1735–1746.

[79] Peter Snyder, Periwinkle Doerfler, Chris Kanich, and Damon McCoy. 2017. Fifteen minutes of unwanted fame: Detecting and characterizing doxing. In *Proceedings of the 2017 Internet Measurement Conference*. ACM, 432–444.

[80] Saumya Solanki, Gautam Krishnan, Varshini Sampath, and Jason Polakis. 2017. In (Cyber)Space Bots Can Hear You Speak: Breaking Audio CAPTCHAs Using OTS Speech Recognition. In *Proceedings 10th ACM Workshop on Artificial Intelligence and Security (AISec '17)*.

[81] Sooel Son, Kathryn S. Mckinley, and Vitaly Shmatikov. 2013. Fix Me Up: Repairing access-control bugs in web applications. In *In Network and Distributed System Security Symposium (NDSS)*.

[82] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild.. In *NDSS*.

[83] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. 2017. How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 971–987. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/stock

[84] Ben Stock, Giancarlo Pellegrino, Christian Rossow, Martin Johns, and Michael Backes. 2016. Hey, You Have a Problem: On the Feasibility of Large-Scale Web Vulnerability Notification. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association.

[85] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. 2015. From facepalm to brain bender: Exploring client-side cross-site scripting. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. ACM, 1419–1430.

[86] Kurt Thomas, Dmytro Iatskiv, Elie Bursztein, Tadek Pietraszek, Chris Grier, and Damon McCoy. 2014. Dialing Back Abuse on Phone Verified Accounts. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. 465–476.

[87] T. Unger, M. Mulazzani, D. FrÃijhwirt, M. Huber, S. Schrittwieser, and E. Weippl. 2013. SHPF: Enhancing HTTP(S) Session Security with Browser Fingerprinting. In *2013 International Conference on Availability, Reliability and Security*.

[88] Kami Vaniea and Yasmeen Rashidi. 2016. Tales of Software Updates: The Process of Updating Software. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI âĂŹ16)*. Association for Computing Machinery.

[89] Rui Wang, Shuo Chen, and XiaoFeng Wang. 2012. Signing Me Onto Your Accounts Through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 365–379. https://doi.org/10.1109/SP.2012.30

[90] Rui Wang, Shuo Chen, and XiaoFeng Wang. 2012. Signing Me Onto Your Accounts Through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society.

[91] Rick Wash, Emilee Rader, Kami Vaniea, and Michelle Rizor. 2014. Out of the Loop: How Automated Software Updates Cause Unintended Security Consequences. In *10th Symposium On Usable Privacy and Security (SOUPS 2014)*. USENIX Association.

[92] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1376–1387.

[93] Ronghai Yang, Wing Cheong Lau, Jiongyi Chen, and Kehuan Zhang. 2018. Vetting Single Sign-On SDK Implementations via Symbolic Reasoning. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity18/presentation/yang

[94] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Haixin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. 2015. Cookies Lack Integrity: Real-World Implications. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/zheng

[95] Yuchen Zhou and David Evans. 2010. Why aren't HTTP-only cookies more widely deployed. *Proceedings of 4th Web* 2 (2010).

[96] Yuchen Zhou and David Evans. 2014. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association.

[97] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. 2019. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud From Mobile Apps. In *2019 IEEE Symposium on Security and Privacy*. San Francisco, CA.

[98] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2017. AUTHSCOPE: Towards Automatic Discovery of Vulnerable Authorizations in Online Services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM.

A APPENDIX

A.1 Browser Automation

Unexpected Alerts. If an alert popup appears and an `UnexpectedAlertPresentException` is raised during the invoked method, the execution context is switched temporarily to the alert box, which is then dismissed, and the method is retried. To prevent other alerts from appearing in the current page's context, the `window.alert` method is overridden.

Retry mode. We have developed a *retry mode*, which is used by XDriver whenever it needs to perform an action it can retry in case of failure; this is done without having to return control back to the caller, e.g., when a page's links or login forms are requested. Specifically, if an exception is raised while performing the operation, XDriver will retry the operation for a certain amount of times before raising the exception or returning a default value.

Built-in crawler. Our custom browser automation tool includes a built-in crawler for streamlining crawl-based tasks, a functionality that is especially vital in security-related studies. In our framework's context it is useful for our URLDiscovery and PrivacyAuditor modules for crawling and processing websites. Modules that want to initiate a crawl only need to call the `crawl_init` method with the desired configuration options and then iteratively call the `crawl_next` method, where all logic of the crawl is transparently implemented. The following configuration options are currently supported by our system: (i) Crawl depth, (ii) DFS or BFS mode, (iii) optional support for a set of regular expressions that dictate which URLs and even subdomains to follow or not follow (e.g., focus only on login related URLs or crawl a specific subdomain), and (iv) an optional break function that is applied after every fetched URL to determine whether the crawl should stop (e.g., if a specific type of form is found).

Return values. Additionally, to simplify the checks that the caller modules have to make for determining whether a requested operation was successful, we refrain from raising Selenium-level exceptions and, instead, return default boolean values. Only in cases where our handling mechanisms cannot resolve an issue we consider the exception to be fatal and raise it. For instance, when a module attempts to interact with an element that is not currently interactable (e.g., clicking an invisible element) a `False` value is returned instead of raising the default `ElementNotVisibleException`.

A.2 Attack Workflow Statistics

In Figure 4 we plot the number and percentages of domains processed during each phase of our auditing procedure's workflow. First, our system identifies appropriate account signup or login pages for ~13.4% of all the domains included in our dataset. Next,

Algorithm 1 *CookieAuditor* algorithm

```
1: function AUDIT
2:     critical_cookies ← {
3:       'secure'      ← ['cookieA','cookieB', ...],
4:       'httpOnly' ← ['cookieD','cookieF', ...], }
5:     vulnerable ← { 'secure' ← NULL,
6:       'httpOnly' ← NULL, }
7:     tested ← [ ]
8:     for attr, cookies in critical_cookies do
9:         if cookies.is_empty() then
10:            vulnerable[attr] ← True
11:        else
12:            for tested_attr in tested do
13:                tested_set ← critical_cookies[tested_attr]
14:                if cookies == tested_set then
15:                        vulnerable[attr] ← vulnerable[tested_attr]
16:                else if vulnerable[tested_attr] AND
17:                cookies.is_subset(tested_set) then
18:                        vulnerable[attr] ← True
19:                end if
20:            end for
21:            if vulnerable[attr] == NULL then
22:                vulnerable[attr] = EVAL(cookies)
23:            end if
24:        end if
25:        tested.append(attr)
26:    end for
27:    return vulnerable
28: end function
29: function EVAL(cookie_set)
30:    BROWSER.remove_cookies(cookie_set)
31:    BROWSER.refresh()
32:    return login_oracle()
33: end function
```

the account creation process successfully completes for almost 12% of those domains. As discussed in Section 5, the automated account creation process is the biggest challenge for our framework due to two reasons. First, the registration process may include predicates that significantly complicate the automated input generation due to input format constraints. For instance, the registration may include a mandatory field (e.g., postal address) that requires a valid value for a specific location/country. Iteratively testing different input formats can prohibitively increase the duration of the auditing process at the scale of our analysis. Second, registration might require access to a specific resource (e.g., phone number or credit card) that is not feasible to obtain for a study of our scale. After the account creation, we find that over half of the audited domains fail to correctly protect their cookies and are susceptible to one of the attacks covered by our threat model (as inferred by our Cookie Auditor module presented in Algorithm 1). The remaining modules are highly effective and infer the authentication cookies and detect identifier leakage in the vast majority of the audited domains. The failures in these modules are attributed to websites timing out (or being generally unresponsive) after several auditing tests and network failures. Also, when re-evaluating these domains other factors can affect the execution of our modules, such as our test account being deactivated, expired domains etc.

False negatives. To obtain more insights about our framework's effectiveness we perform an indicative experiment where we investigate the false negative rates (FN) of the different modules in our system. Specifically, we randomly sample 20 websites per module, where the module's execution did not complete successfully, and manually inspect whether these failures were actual true negatives or not. For our URL discovery module, we identified only four FNs, i.e. in four cases there was a login option that our system failed to detect. Our generic account setup component yielded 3 FNs, i.e.

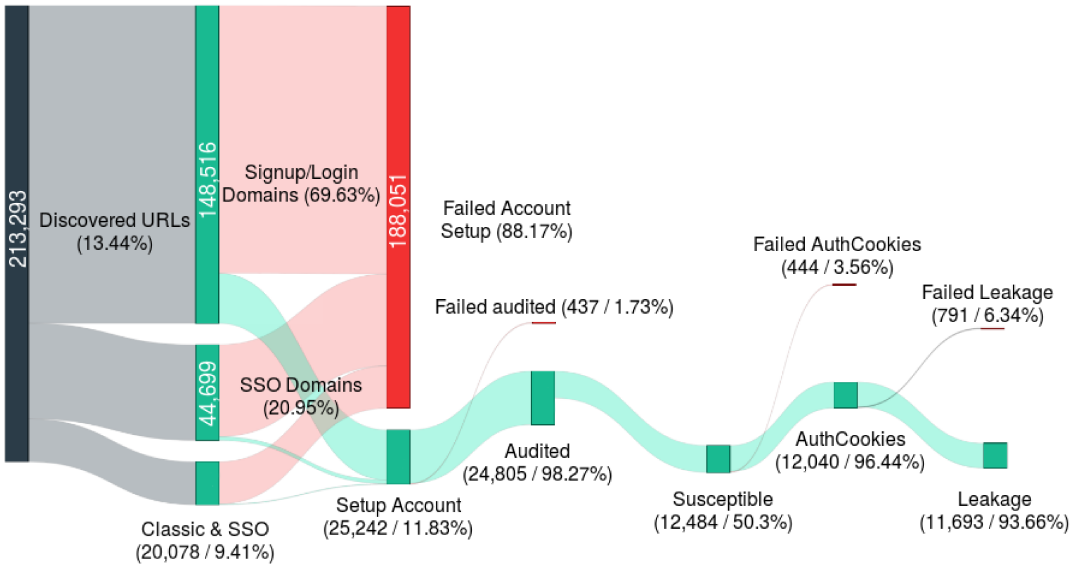

Figure 4: Success rate for different workflow phases.

we successfully signed up and/or logged in the website, but were not able to infer the state. Similarly, the SSO module had 5 FNs. The Cookie Auditor yielded zero FN, meaning that there was not a single case where our system identified a website as secure against an attack, while it really was vulnerable. Finally, the Privacy Auditor had 4 FNs, i.e. there was account information that we provided during the signup process that was not detected as being leaked. We did not measure the Authentication Cookies FN rates, as manually identifying *all* authentication cookies and combinations is prohibitively time consuming or even infeasible in many cases.

URL discovery effectiveness. As mentioned, our URL discovery module initially explores the URLs provided by [44] before falling back to our own crawling approach. As such, it is of interest to quantify how useful this dataset was and, more importantly, how effective our system was in cases where it had to employ our own approach. For all the websites where we identified a signup option, 23.1% were fully discovered using the dataset from [44], while for the remaining 76.9% we had to fall back to crawling the websites (43.1% were included in both datasets, while 33.8% were not included in [44]).

Failed registrations. In an attempt to better understand the reasons behind failed registrations, we manually inspected 50 randomly selected websites. In 22 cases, there was some form of an anti-bot challenge that our system was not able to solve and, thus, could not proceed with registration. In 23 websites one of the fields was rejected due to inappropriate formatting, e.g. mobile phones, addresses, passwords etc. Finally, the remaining 5 websites failed due to unexpected or complex form behavior, e.g. after filling in a specific field, a custom drop down list appeared that also needed to be detected and filled out.

A.3 Manual Session Hijacking Verification.

Table 6 breaks down the results from our manual session hijacking validation experiment. We observe that in all but one cases, the access we obtain through our cookie hijacking attacks leads to the exposure of sensitive information and functionality even if we only obtain partial access. This includes the ability to view and edit personal information, as well as execute site-specific functionality. As expected, in most cases we cannot (fully) change account settings (e.g., password, email). This is due to the fact that such operations typically require the user to retype their password, which is not known to the cookie-hijacking attacker. Nonetheless, we found that multiple domains allow the attacker to change the password even without knowledge of the current password.

Table 4: Most common categories of susceptible domains.

Category	#domains	Category	#domains
Online Shopping	3,725	**Soft/Hardware**	252
Business	1,117	**Sports**	234
Marketing/Merch.	1,100	**Job Search**	229
Internet Services	642	**Pornography**	194
Entertainment	586	**News**	187
Education/Reference	558	**Real Estate**	178
Blogs/Wiki	393	**Public Info**	153
Fashion/Beauty	322	**Health**	148

A.4 Domain Categorization

Domain categorization. Table 4 reports the top domain categories (classified using McAfee's URL Ticketing System [14]) that are vulnerable to at least one attack. We find that online shopping is the most prevalent category of susceptible domains, highlighting the privacy threat of cookie hijacking. These services include a plethora of personal data (e.g., address), while. recommendations and prior purchases can reveal sensitive user traits (e.g., sexual orientation, religion). We also find 148 and 194 domains that provide

Table 5: The 20 most popular vulnerable domains.

Domain	Eavesdropping	JS cookie stealing
amazon.com	✓	✓
reddit.com	✗	✓
twitch.tv	✗	✓
mail.ru	✓	✓
aliexpress.com	✓	✗
alipay.com	✓	✓
bing.com	✗	✓
amazon.co.jp	✓	✓
ebay.com	✓	✓
msn.com	✓	✗
xvideos.com	✓	✓
wordpress.com	✓	✗
amazon.in	✓	✓
xhamster.com	✓	✗
amazon.co.uk	✓	✓
pixnet.net	✓	✓
bongacams.com	✓	✗
roblox.com	✓	✗
nytimes.com	✓	✓
soundcloud.com	✗	✓

health-related functionality and adult content respectively,. which potentially enable access to extremely sensitive user data.

A.5 Popular Domains

Table 5 presents the 20 most popular domains found vulnerable during our study, which span various categories (e.g., e-commerce, blogging, pornography etc.). We manually verified the feasibility of session hijacking attacks in every one of these domains. It is important to note that all of these services have a massive user base, most likely employ professional development teams and may even have dedicated security teams, yet they still expose their users to significant threat. Our PrivacyAuditor module also uncovered several interesting findings. One domain leaked the password hash in a cookie (avgle.com), two leaked the phone number in the page's source (123rf.com, naukri.com) and one in the local storage (southwest.com). One domain leaked the user's postal address in the source (asus.com) and two leaked the user's workplace in the source (alibaba.com, mailchimp.com).

Another interesting observation is that even major services like Amazon struggle with the correct deployment of security mechanisms. Specifically, we found that while amazon.com deploys HSTS, it does so in an incomplete manner. The policy is only set on the "*www*" subdomain and thus the authentication cookies we have identified are leaked over unencrypted connections to the base domain, since their domain attribute is set to ".amazon.com".

Table 6: Manually validated domains and hijacking capabilities.

Domain	Read	Write	Settings	Exposed information & functionality
Top-1K (hand-picked)				
amazon.com	◑	◑	✗	View/edit cart, ad preferences, vouchers/coupons, shopping list, email subscriptions, deals & notifications, browsing history and recommendations
aliexpress.com	●	●	✗	View/edit favorite stores, wish list, cart, profile photo, full name, follow sellers. View messages, order history, coupons
ebay.com	●	◑	✗	View/edit cart, watchlist, saved searches/sellers, messages, address, profile photo. View recently viewed items, active bids/offers, purchase history, own items for sale
alibaba.com	●	●	✗	View/edit cart, full name, phone number, gender, address, job information, favorites, profile photo. View messages, orders, transactions, contacts, recommendations
reddit.com	●	●	✗	View/edit posts, comments, saved, display name, about section, profile photo, inbox, email notifications, block users
bing.com	◑	◑	✗	View/edit search history, interests. View first name, profile photo
bestbuy.com	◑	◑	✗	View/edit cart, saved items. View shopping history, orders
banggood.com	●	●	✗	View/edit cart, wishlist, address, full name, gender, phone number, messages, reviews, comments, download full activity record. View orders, coupons, gifcards, search history
wish.com	●	●	✗	View/edit cart, wishlist, full name, birthdate, email, notification settings. View orders, recently viewed items
cloudflare.com	○	○	✗	None. The attack only succeeds when performed from the same PC
Top-1K (randomly selected)				
indeed.com	◑	◑	✗	View/edit saved job offers, job applications, scheduled interviews, visited jobs
hotels.com	◑	◑	✗	View/edit favorties, searches
vidio.com	●	●	✓	View/edit phone number, comments, followed channels, password. View transaction history, watch history
nature.com	●	◑	✗	View/edit full name, professional information, subscriptions. View email
sciencedirect.com	●	●	✗	View/edit full name, email, job information, phone number, address. View recommendations, history
1fichier.com	●	●	N/A	View/edit files, folders, full name, address, phone number.
bitly.com	●	●	✗	View/edit bitlinks, link statistics, email address, delete account. View API key, session history (and disconnect all sessions)
cdiscount.com	●	●	✗	View/edit subscriptions, wish/favorites list, address, phone number. View email, birthdate, orders, messages, vouchers, credit card info
elsevier.com	●	●	✗	View/edit cart, full name, email, address, phone number, partial payment information, add new credit card
espncricinfo.com	●	●	✗	View/edit full name, email, phone number, gender, address, delete account
Any-rank (randomly selected)				
sendatext.co	●	●	N/A	View/edit SMS texts (sent and replies), calls, address book
metzlerviolins.com	●	●	✓	View/edit address, cart, wish list, password. View orders
swotanalysis.com	●	●	✗	View/edit teams and members, billing history, projects
kokpit.aero	●	●	✓	View/edit full name, email, phone number, password, comments
brauchekondome.com	●	●	✗	View/edit full name, address. View email, birth date, orders
soccergarage.com	●	●	✗	View/edit username, email, company name, address, cart, wish list, delete profile
packlane.com	●	○	N/A	View orders, saved designs
doggiesolutions.co.uk	●	●	✗	View/edit full name, email, address, cart, delete profile. View order history
jellyfields.com	●	●	✓	View/edit email, username, website, favorites, password
helmetstickers.com	●	●	✓	View/edit full name, address, cart, password, delete profile. View order history

Access: full ●, partial ◑, none ○