To Auth or Not To Auth? A Comparative Analysis of the Pre- and Post-Login Security Landscape

Jannis Rautenstrauch, Metodi Mitkov, Thomas Helbrecht, Lorenz Hetterich, Ben Stock
CISPA Helmholtz Center for Information Security
⟨first⟩.⟨last⟩@cispa.de

Abstract—**The web has evolved from a way to serve static content into a full-fledged application platform. Given its pervasive presence in our daily lives, it is therefore imperative to conduct studies that accurately reflect the state of security on the web. Many research works have focussed on detecting vulnerabilities, measuring security header deployment, or identifying roadblocks to a more secure web. To conduct these studies at a large scale, they all have a common denominator: they operate in automated fashions without human interaction, i.e., visit applications in an unauthenticated manner.**

To understand whether this unauthenticated view of the web accurately reflects its security as observed by regular users, we conduct a comparative analysis of 200 websites. By relying on a semi-automated framework to log into applications and crawl them, we analyze the differences between unauthenticated and authenticated states w.r.t. client-side XSS flaws, usage of security headers, postMessage handlers, and JavaScript inclusions. In doing so, we discover that the unauthenticated web could provide a significantly skewed picture of security depending on the type of research question.

1. Introduction

The web has won. Users spend more time on the web than ever before and perform all kinds of actions, from searching for information, over entertainment, to banking and business. However, attacks on websites and users are a constant threat, and news reports about hijacked accounts and breached websites release almost daily. Over the years, much research has focussed on detecting vulnerabilities and deployment of countermeasures at scale using automated crawlers [22, 23, 26, 36, 43, 44]. However, such crawlers usually use fresh browser instances, and all rely on the tacit assumption that what they measure is a good enough approximation of what real users would encounter.

There is an ever-growing body of research, particularly in the privacy domain, examining differences between various crawling configurations and standard user browsers, which shows that geolocation, IP address, and crawling technology may impact the results. Varying results can be due to differing behavior based on user-agent switching, legislation that governs the web (e.g., the GDPR in Europe), or merely bot detection mechanisms deployed by major CDN providers [1, 2, 11, 21, 50]. The fact that users do not visit websites in fresh browser profiles every time but instead have rich information saved in their browsers (such as visited subpages, being authenticated, and having accepted cookies) has yet to be examined in detail in the area of security. One recent study by Klein et al. showed that consenting to cookie banners increases the included third-party scripts by 45% and, in turn, resulted in 55% more verified XSS exploits [25]. Other works have shown that websites deliver different security headers to authenticated and non-authenticated requests enabling XS-Leaks [32, 46].

In essence, most prior work looking at understanding security threats on the web has been performed without establishing meaningful user state. This paper systematically analyzes the post-login security landscape and compares it with its pre-login counterpart to close this research gap. We explicitly answer the following overarching research question: *Does the security landscape differ significantly between logged-in users and their non-logged-in counterparts?* Are the results strictly better, worse, or comparable if there are differences? Can we identify any specific patterns in the observed data? And, of utmost importance for both past studies and future research: is a non-logged-in view of the web an accurate reflection of the general state of security on the web?

To answer these questions, we created a semi-automatic framework to create accounts and login on various websites and use it to perform the largest-to-date study on the post-login security landscape on 200 sites. We study security header usage and misconfigurations, JavaScript usage and dependencies, reflected and persistent client-side XSS, and postMessage issues. The results of our experiments show that depending on the research question and subject of study, the differences between authenticated and non-authenticated crawlers can be essential and should be considered by future research in the area of web security.

To sum up, our paper makes the following contributions:

- We open-source a semi-automatic framework to ease future post-login studies (Section 4).
- We create a systematic methodology to study the security landscape of websites in different states (Section 5).
- We perform a comprehensive study on security differences between pre-login and post-login websites by analyzing 200 sites across four experiments (Section 7).
- We discuss the impact of our results and propose recommendations for future studies (Section 8).

2. Background

Given the web's success, attackers have also found their way to it. Modern websites are plagued with a plethora of security issues, and their attack surface keeps increasing due to applications and browsers getting more complex. Apart from individual well-publicized attacks and reports to singular websites by penetration testers or bug bounty programs, most of the knowledge of the state of the security on the web is constructed by web crawlers testing for security issues on a large scale.

Over the years, various attacks have been found, most of which are mitigated after the initial discovery through various security mechanisms. For example, the long-running reign of Cross-Site Scripting (XSS) can be mitigated by deploying a Content Security Policy (CSP). Similarly, attacks such as Clickjacking have their own countermeasures by controlling framing through CSP's frame-ancestors or the (deprecated, yet still widely used [6]) X-Frame-Options header. In the following, we outline the issues that have been covered by prior work, which sets the scene for the experiments we conduct throughout our paper.

2.1. Client-Side XSS

Cross-Site-Scripting (XSS) is the most prevalent attack on the web. It allows attackers to execute code on a website in the context of the attacked user. XSS is commonly classified by two dimensions: where the vulnerable code is located, server-side or client-side, and whether the payload is permanently stored, persistent, or reflected.

While prior work has developed techniques to detect server-side XSS [9, 19], scanning for these at scale is ethically questionable. Specifically, finding persistent XSS implies that a payload would also be visible to others, which either interrupts regular browsing or, worse, allows others to understand the susceptibility of the site to an XSS and develop an actual exploit. To that end, our work focuses on client-side XSS. This type of Cross-Site Scripting was first described by Klein [24] as *XSS of the third kind* or *DOM-based XSS*. More recent work has instead used the term *client-side* XSS to accurately reflect the usage of non-DOM APIs for both sources and sinks. Contrary to server-side XSS, where the actual vulnerable code is unknown to the browser, client-side XSS implies that all vulnerabilities are contained in client-side code and, thus, executed in the browser. Prior work [28, 30, 43] has used a taint tracking approach to find XSS. Since XSS boils down to data flows from attacker-controlled sources (e.g., the URL) to dangerous sinks (e.g., `eval`), taint tracking is well-suited. As was done in prior work, we leverage the taint data with an exploit generator to confirm the presence of a vulnerability. For this, we consider both reflected client-side XSS (through URLs or the referrer [28, 30]), and persistent client-side XSS (from Local Storage or cookies [43]). Most importantly, since client-side XSS occurs exclusively on our client, no other visitors of the tested site will be affected.

2.2. Security Headers

The web platform initially focussed exclusively on functionality rather than security. This is still evidenced today by the fact that authentication is enabled through cookies, a mechanism never intended to be related to security. Over the years, various mechanisms have been added to address security issues in the platform retroactively. Here, we outline the three most prevalent and impactful security headers and the attacks they mitigate.

2.2.1. Content-Security-Policy. Cross-Site Scripting (XSS) boils down to a simple fact: attacker-controlled code is executed within the confines of a victim application. Naturally, this code is not desired to be present by the site's developer. The original idea of the Content Security Policy (CSP [41]) was to enable an allowlist of scripting resources that should be executed. This would enable the developer to specify the scripts they wanted to execute, leaving it to the browser to ensure that any script not explicitly allowed would be blocked. However, CSP's first version did not support a meaningful way to enable developer-controlled inline scripts. This is because browsers either blocked all inline scripts if some CSP was specified, or allowed *all* inline scripts in case the dreaded `unsafe-inline` keyword was contained in the script-src directive. To overcome this, CSP level 2 introduced the concept of nonces and hashes, allowing developers to attach a unique value to inline scripts (nonce case) or allow inline scripts by their hash. In either case, an attacker would be unable to guess the nonce or find a hash collision, thereby allowing fine-grained scripting control. CSP's specification is currently at level 3 [8], which includes additional directives that go well beyond scripting control.

The first major other use case is the enforcement of TLS. The introduction of the `upgrade-insecure-requests` directive allowed site developers to ensure that resources included in their page could not be (accidentally) loaded through insecure HTTP connections. This feature also comes in handy when migrating from HTTP to HTTPS [35], as any remaining HTTP links are seamlessly migrated to their HTTPS counterparts.

Finally, the third major use case of CSP is framing control. The web stack provides ample ways for an attacker to abuse it. One attack enabled by the freedom to position HTML elements anywhere on a page and even control their opacity is Clickjacking. Here, the attacker finds some page on the vulnerable site which causes a state-changing action based on a single button click, e.g., a one-click buy button in a web store. They then load the vulnerable page in a frame, position the frame on top of some element the user can be lured to click on (e.g., a button to load more content or one that is part of a game), and make it transparent. Hence, the unknowing victim trying to click on the attacker page's button instead clicks into the invisible iframe and onto the one-click buy button. To overcome this, site operators can use CSP's `frame-ancestors` directive. This specifies which URLs are allowed to frame a particular page. As

CSP is enforced by the browser, the attacker may still try to force the victim to load the target page in an iframe. However, unless the attacker's URLs are allowed within `frame-ancestors`, the browser refuses to load the page, thereby stopping the attack.

2.2.2. X-Frame-Options. The predecessor to CSP's `frame-ancestors` is the `X-Frame-Options` (XFO) header. While the syntax of the header and its flexibility (e.g., to allow multiple sources to frame a page) are limited, the functionality roughly aligns with its CSP counterpart. XFO instructs the browser to block the loading of a page, either always or for cross-origin embeddings. Notably, XFO was only formally specified [17] well after it had been implemented by various browsers, which led to inconsistent support across browsers. As a side effect that is still observable today, modern browsers do not support the `ALLOW-FROM` directive, which was meant to enable a single origin to frame the page. However, site operators may still rely on this old configuration, which effectively nullifies the protection given by XFO. Importantly, though, since CSP's `frame-ancestors` was meant to replace XFO entirely, if CSP is configured to control framing, modern browsers ignore the XFO header altogether.

2.2.3. Strict-Transport-Security. The final security header we consider in our work is HTTP `Strict-Transport-Security` (HSTS) [16]. Given that all of us are using the web on a daily basis, including for sensitive tasks like banking, it is imperative that the connection between a client and server is secured with TLS. Otherwise, a man-in-the-middle attacker could easily eavesdrop on credentials being transmitted or steal the authentication cookies sent by the browser the browser to impersonate the victim. To overcome this, browsers have long since implemented HSTS. This header, when sent by the server through a TLS-enabled HTTPS connection, instructs the browser *not* to visit the given host through an insecure HTTP connection until the timeout of `max-age` has been reached. HSTS can also be set with the `includesubdomains` keyword, which ensures that all subdomains of the given domain cannot be loaded through an insecure connection.

2.3. JavaScript Inclusions

A modern site on the web no longer consists only of code that is developed by the site's operator. Instead, sites rely on the inclusion of external scripts, be it for advertising, map services, or general-purpose libraries such as jQuery. The fact that any site can include scripts from anywhere else, yet the code executes within the confines of the including site, also means that a vulnerability within a third-party script becomes the problem of the first party if they include said script. Therefore, generally speaking, the inclusion of additional code implies a larger attack surface. Prior work has focused on various aspects of third-party script inclusions. These range from using scripts to track users [3] to how the inclusions interfere with CSP deployment to the inclusion of known-vulnerable and sometimes duplicate libraries [26, 42]. Moreover, in the context of online ads, particularly, included scripts often include more scripts dynamically, which means a delegation of trust occurs. This leaves first-party developers with little control over what code is executed within their security boundaries.

2.4. PostMessage Issues

The web's most fundamental security mechanism is the Same-Origin Policy (SOP). It ensures that only documents which share an origin (i.e., the protocol, host(name), and port) can access each other through JavaScript. However, in the context of a modern application, sites may load external content, e.g., ads, in cross-origin frames. Notably, these frames might need to exchange some data between them and their parent page. Under the confines of the SOP, this is impossible. To overcome this, the postMessage API was introduced. It allows two documents loaded in the same browser to interact with each other through a well-defined message API. This way, they can exchange data without allowing full access to each others' content.

Browsers enable the confidentiality, integrity, and authenticity of messages. Specifically, a sending document can specify the target origin of another window to which it posts data. If this window (e.g., an iframe) has been navigated away from the target origin, the message will not be delivered, thereby ensuring confidentiality. Similarly, postMessages always go through the browser core, which means that they cannot be manipulated by an attacker, ensuring the integrity of the message. Finally, when handling an incoming postMessage, the receiving JavaScript code can check the `origin` property of the message event. It contains the origin of the original sender and cannot be forged by an adversary, which ensures authenticity. However, specifying the target origin and verifying the origin of the incoming messages is optional and only ensures their properties *if* implemented correctly.

Prior work has shown that while the overall state of postMessage security has improved since the early 2010s [40], some sites still implement postMessage origin checks insecurely [44]. If sites then incorporate incoming messages as part of their business logic or, worse, pass the content to XSS sinks such as `eval` or `innerHTML`, the security provided by the SOP is effectively undermined. Moreover, as shown by Steffens and Stock [44], other attacks, such as using a vulnerable handler to relay a message, thereby laundering the original message sender, or plain data leakage by not setting the target origin, are possible.

3. Related Work

The following section reviews other studies concerning meta questions of web studies, client-side vulnerability measurements, login automation and security of login.

3.1. Meta Questions on Web Security Measurements

In the past years it became more and more clear that reliable knowledge of the state of the security in the web can only be achieved by large scale crawls and that the results obtained by such crawls might be inaccurate. A variety of studies studied how various features such a IP address (cloud vs residential), geolocation, browser-configuration, choice of crawling technology, or URL collection method (landing page only vs internal pages) affect web measurement results [1, 2, 11, 21] and how crawlers compare to the human browsing experience [50]. Most similar to our work Klein et al. studied how giving consent to cookie banners changes the number of third-party scripts and XSS exploits [25].

We instead compare how being logged-in influences the results of web security measurements which is another feature that was not yet studied by previous works. We left all other parameters constant according to best practices.

3.2. Client-Side Vulnerability Measurements

Many researchers crawled the web to assess the real world risks for client-side vulnerabilities such as CXSS [28, 30, 43], inconsistent security headers [6, 36], postMessage issues [40, 44], vulnerable JavaScript libraries [26], DOM-Clobbering [23], or prototype pollution [22].

All the above works used anonymous sessions that opened websites under test in a fresh browser. We measure a subset of the same vulnerabilities pre- and post-login to be able to compare the security landscape of logged-in users and how it is different from non-authenticated sessions.

3.3. Login Automation and Security of Login

Various researchers automated parts of the process for studying logged-in users. From automatically finding login forms [47], over automatically logging in using given credentials [20] up to full automation of registration and login [10, 13]. In addition, there exist several works on the topic of SSO automation [18, 51]. Using these automated tools or using a full manual process several researchers studied the security of the login process itself or similar (e.g., whether the cookies are set securely or whether the sessions are correctly invalidated) [4, 13, 47].

Most of these works did not open-source their tools and cannot be used by the community. We took inspiration from their described procedures for our new open-source semi-automatic login and registration tool chain *account framework*. Instead of focusing on the security of the login process itself, we ask whether there are systematic differences between the security landscape of authenticated sessions and non-authenticated sessions and whether crawling the web without being logged-in gives reasonably accurate results.

4. The Account Framework

Studying post-login issues is a challenging and labor-intensive task. It requires accounts on various websites and the ability to login to each account on each website. Performing all these steps by hand for various experiments is time-consuming and does not scale. Thus, we created a semi-automatic framework that eases account registration and manages login, validation, and session distribution. We open-source the account framework to assist other researchers studying problems on the web that require being logged-in on websites[1].

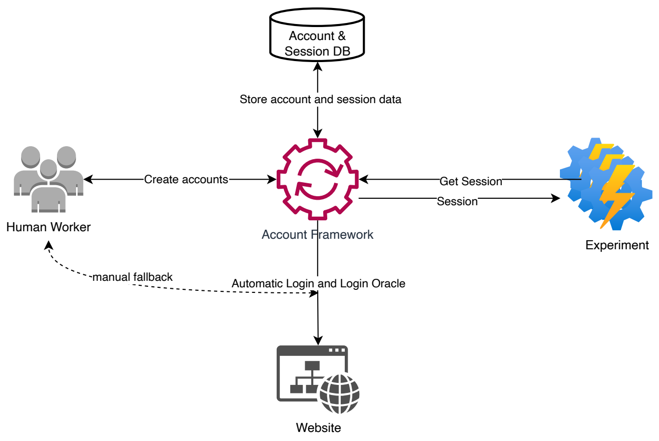


Figure 1: Overview of the Account Framework.

4.1. Semi-Automatic Framework

Figure 1 presents a general overview of the account framework. Each deployment contains one central account framework instance that manages all accounts and sessions. Each consuming client, i.e., experiment, can request a session, and if there is a valid session in the database, this session (currently localstorage and cookies; the alternative would be the complete browser profile) is returned and can be used by the client until it is given back to the framework or a configurable timeout is reached. The default time an experiment can use a session is 24 hours, and sessions are revalidated after being returned or after 12 hours. The backend ensures that no two clients receive a session belonging to the same account simultaneously and that no experiment receives a session for the same account (or website) twice. The framework can handle arbitrary many identities, and each identity can have accounts on many websites. For this study, we only used one identity.

The account framework is responsible for managing the account and session data. Experiments consume sessions, and producers (either automated scripts or human workers) create accounts and perform logins. For the scope of this study, we implemented fully-automated login and validation workers and password-manager-assisted scripts for registration, login, and validation that require a human in the loop. Both the automated and human-in-the-loop workers communicate with the account framework via an API and can be replaced by others in the future.

1. Available at https://github.com/cispa/login-security-landscape

The framework accepts any list of websites and any list of identities as input. For each identity, the framework then creates account creation tasks. For each successfully created account, the framework creates login tasks; for each successful login, it creates a session validation task. New validation tasks are also scheduled after a session was used by an experiment or after 12 hours (timeouts can be changed). If a task fails, either the task is recreated for a different worker type (e.g., if automated login fails, we fallback to a manual login), a different task is created (if validation fails, we create a new login task), or the account is marked as broken (e.g., the account got blocked or the website is not working anymore).

We also provide an optional helper script that uses an automated browser to find login and registration forms on a list of websites. It only creates account creation tasks on websites where it found such forms and attaches the discovered login and registration URLs to these tasks.

For our experiments, we run the framework on the same machine as the experiments to minimize issues related to risk-based authentication, e.g., if websites check the IP address or the operating system to ensure the sessions have not been stolen. For human-in-the-loop workers, we use Xvfb and VNC to work on the same machine. However, in general, the only requirement is that the workers can reach the account framework's API.

4.2. Assisted-Manual Registration

Our framework assists manual workers in making their work smooth and fast and minimizing the time for the human-in-the-loop. Initially, we start the assisted-manual script once to set everything up correctly. We login into Gmail (opened automatically and used for Email verification and email-based login) and the Bitwarden browser extension (password manager). In addition, we configure the extension to autofill and never logout. Our script also adds all identities of the account framework to Bitwarden.

To perform tasks, a human connects to the manual worker script via VNC. A command-line program coordinates the general process. If the user starts a task, the program first adds the account information to use (such as password and username) to Bitwarden. Then, it opens one browser with the registration form of a website open (if our optional script was not used and the framework does not know the registration URL, it will open the landing page). In addition, it opens another browser with our Gmail account open. Ideally, Bitwarden auto-filled all required information, and the human worker must only confirm. If not, the worker has to fill in the remaining required information and perform other steps to succeed in the registration, such as solving captchas. After the worker has successfully finished the task or failed to register an account, e.g., because account creation requires payment, the worker closes the browser. Following the close of the browser, the program asks the user several questions about the outcome before the user has to decide whether to continue with the next task or stop. The program sends the registration results to the account framework and schedules either an automatic login task or saves the reason why no account could be created.

4.3. Automated Login and Validation Tasks

The automated workers that fetch login tasks and the automatic helper script prepares account creation tasks use a module to find login and registration forms. The module is also responsible for filling out the login fields and logging in. We use heuristics similar to prior work [13, 20] to achieve this and use keyword lists with English and German words. For example, HTML forms with at most one password field or one or two active input fields are marked as potential login forms. If these forms contain clickable elements with login keywords, we mark them as login forms. These heuristics allow us to detect even involved multi-step login forms. Automated workers solving login tasks fill the login forms' input fields with the appropriate credentials. They use the input types, placeholders, ids, and labels of input elements to derive the correct submission. Then, they submit the form by hovering and clicking submit-like buttons. New tasks are then scheduled for automatic verification or manual login based on the outcome. Finally, the resulting session is saved in the account framework (i.e., LocalStorage and cookies).

The automated workers that fetch verification tasks use a module to verify if an account is logged-in. Again we use heuristics similar to Drakonakis et al. [13] and Jonker et al. [20]. Specifically, we check if account indicators, such as username or email, are present on the landing page of the login state but missing from the landing page of an unauthenticated fresh browser. If that fails, we check if the URL with the login form is accessible or that the login form is missing in the login state.

The automated login and verification can fail due to various reasons: captchas, complex custom login procedures, blocked accounts, and similar. Thus, we have a manual fallback option where failed tasks get rescheduled for manual workers. However, we did not use this option for the scope of this study. The automatic verification oracle could produce false positives. However, false positives are unlikely as the oracle checks for account credentials, which we hand-picked during the registration process to be sufficiently unique to ensure they do not randomly occur on a site.

5. Measuring Security Relevant Differences

In the following, we describe how we measure various security indicators pre- and post-login and compare them, including security headers, JavaScript usage (e.g., vulnerable libraries and amount of third-party inclusions), client-side XSS, and postMessage usage.

5.1. Client-Side XSS

For client-side XSS, we compared the prevalence of potentially vulnerable sink invocations and actual vulnerable client-side XSS present in logged-in and non-login states.

To collect taint reports, we used the Foxhound browser engine [39], a fork of Firefox with taint tracking capabilities. We used the exploit generator by Steffens et al. [43] to produce reflected and persistent XSS exploits for detected tainted sink invocations in either state. We then validated each generated exploit during crawling. For each verified exploit in one state, we afterward verified it in the respective other state to understand whether or not those vulnerabilities are present in both states.

We collected the detected taint flows for each visited page and fed them to the exploit generator. We chose the Foxhound browser [39] as the taint tracking method since the browser engine provides support for recent JavaScript language features not present in the Chromium taint tracker used by Steffens et al. [43]. In our setup, we used Playwright version 1.33 with the Foxhound browser commit `2916e01`, which is a fork of Firefox 109. As the exploit generator was built for the Chromium taint tracker, we converted the taint reports from Foxhound to match the format of the exploit generator. We captured the exploit generator output and subsequently verified each generated exploit by visiting the respective page using the exploit data. We note that Foxhound does not support `iframe.srcdoc` detection, and due to an oversight, no exploits were generated for `script.text`. However, since we leverage the same engine in both states, the impact of the issue is limited, given our *comparative* analysis.

To verify the generated exploits, we proceeded in the following way:

- For reflected exploits, we visited each generated URL containing the payload and checked whether it was executed. This method aligns with prior works [28, 30].
- For persistent exploits, we first visited the vulnerable URL and waited until it was loaded. We then waited 3 seconds and subsequently modified the storage with the output of the exploit generator. Here, we leverage the same logic as Steffens et al. [43]. Afterward, we refreshed the page and checked whether our payload code was executed. We distinguished the storage types `localStorage`, `sessionStorage`, and `document.cookie`, in which we replaced the relevant key accordingly. For exploits that used the `script.src` sink, we included an externally hosted script containing our payload. The URL that pointed towards that script was included in the exploit generation and injected into the exploit data by the generator.

For both verification tasks, we waited for 10 seconds after page load to determine whether the payload executed.

5.2. Security Headers

Many websites use security headers to protect themselves against various security issues. However, these headers are only effective if used securely and consistently. For the scope of this paper, we focus on the following headers on main page responses: `Strict-Transport -Security` (HSTS), `X-Frame-Options` (XFO), and `Content-Security-Policy` (CSP). To evaluate and compare the security of sites and responses, we apply the definitions of consistency and equivalence relations for the semantics of security headers outlined by Roth et al. [36]. However, we modified the definition of equivalence relation for CSP TLS to ignore *'block-all-mixed-content'* as modern browsers ignore it, and mixed content is blocked by default. In addition, we modified the X-Frame-Options parsing code to reflect its recent specification in the HTML standard [17], which results in blocking all framing if several distinct values are specified.

Specifically, we consider a header to be secure if it is present, does not contain syntax errors, and:

- for **X-Frame-Options** either framing is denied, or only same-origin framing is allowed.
- **HSTS** has a positive max-age value.
- **Content Security Policy (XSS)** employs a *safe* CSP, i.e., it is not trivially bypassable, e.g., by including * or *'unsafe-inline'* without nonces or hashes.
- **Content Security Policy (framing)** either only the own origin is allowed, no framing is allowed, or framing is constrained (not containing * or *https:*).
- **Content Security Policy (TLS)** the *upgrade-insecure-requests* directive has to be present.

To collect headers, our crawler uses Playwright to register a listener on each page to capture all request and response pairs. We analyze the responses to top-level documents as all considered headers have a defined semantics for such responses and it is inline with Roth et al. [36]. If a page redirects, we only consider the final document response and only if it is same-site in respect to the originally visited site. Thus, if `domain.com` redirects to `domain.us` because of a geo-specific version, we do not consider the results.

As Roth et al. [36] discussed, sites may exhibit non-deterministic inconsistencies in the headers. The authors noted that this may be due to different origin servers or other types of misconfigurations. We visit each URL five times to avoid drawing incorrect conclusions from these potential intra-test inconsistencies. In line with Roth et al. [36], we compare the semantic equivalence of the received headers to ignore syntactic changes in headers that do not result in different security for users, such as different casing in XFO headers. If all five repetitions of a URL visit have the same security level, we count it as consistent. Otherwise, we count it as inconsistent. If a URL shows inconsistent security headers, we discard it from the comparative analysis to avoid comparing partially random values with each other.

We also compare the security of responses between the two states, logged-in and non-logged-in, to evaluate whether these user groups receive headers that differ in their security. For this analysis, we could only compare URLs collected in both the logged-in and logged-out crawl. In addition, we only consider URLs that were intra-test consistent in both states. We consider a URL intra-test consistent if it had the same *semantic* value for all *five* considered properties for all *five* repetitions. We then compare the semantic values of each property to see which state has better security for this URL or if both states are the same.

5.3. JavaScript Inclusions

For JavaScript inclusions, we inject a modified version of the script used by Steffens et al. [42] into every visited URL to hook APIs related to script execution. Additionally, we use Playwright and the Chrome DevTools Protocol (CDP) to register handlers for events like *Debugger.scriptParsed*. Whenever a hooked API which may lead to parsing and executing new JavaScript code snippets is called, all those snippets are collected and reported alongside a stack trace.

Every time JavaScript code is parsed, the *Debugger.scriptParsed* handler is executed, and the parsed script can be attributed to the corresponding API call. We divide script parser invocations into 5 categories. First, there are two ways of statically adding JavaScript: **inclusion**: the initially served HTML contained the script with a *src* attribute, and **inline**: The initially served HTML contained the script without a *src* attribute. Moreover, code can be dynamically added at runtime through: (1) **dynamic inclusion**, i.e., a script tag with *src* attribute was programmatically created and appended to the DOM, (2) **dynamic inline**, where the DOM was programmatically modified, and a new script was parsed. Last, there is **eval** if the script was parsed due to an invocation of *eval*. As *Eval* cannot be hooked, we indirectly classify scripts as eval if they do not match any other criteria.

To identify third parties, we collect the source URL for each script, if available, and compare it with the site of the originally visited URL. Additionally, we use retirejs [33] to identify used libraries and disconnect [12] to identify known tracking entities..

5.4. PostMessages

We integrated the open-sourced code from PMForce published by Steffens and Stock [44] and updated their code to work with Playwright version 1.33 instead of Puppeteer, while preserving functionality. For each page we visited, we injected their automated analysis framework that leverages force execution and taint tracking for analyzing detected handlers. The SAT timeout for queries to the constraint solver was set to 30 seconds. Each visited URL ran for at least 10 seconds, and afterward, the crawler awaited the termination of the verification of each generated exploit. PMForce captured all parsed handlers and generated exploit candidates, which were immediately verified during the same crawl.

6. Experimental Setting

The aim of this study is to evaluate differences in security indicators between authenticated and non-authenticated browsers and to estimate how accurate of an approximation large-scale web crawls with non-authenticated browsers are for the browsing realities of everyday web users. In the rest of the paper, we refer to the authenticated, i.e., logged-in, state as S_{auth} and to the unauthenticated browser state as S_{noauth}. In theory, one would like to compare as many sites as possible, however, due to the manual nature of account registration, we had to limit the scope of the experiments. To still achieve the goal of best approximating real web browsing behavior, we decided to test popular websites as they are likely to have an account system, and users spent a disproportionally large amount of time on them [37]. To select popular sites, we started with Tranco [27] (up to rank 445) and switched to the CrUX dataset [15] (Top 5K) which better represents web popularity [38]. Due to an overlap between these lists and CrUX using origins instead of sites, our initial dataset consists of 4,485 unique sites. On these we automatically extracted more than 900 sites where we detected a login and a registration form. On these sites, we manually created over 400 accounts during a time period of several months in 2022 and 2023. On the other sites we encountered the following issues:

- Infinite captcha loops, timeouts, bot or country block pages, and broken registration processes: 177
- Payment, phone number, or similar required: 170
- False positives in registration form detection or only SSO login (mainly in an early iteration): 88
- Beyond those, we encountered duplicate domains (.com and .country versions use the same accounts), language issues, or failed delivery for activation emails.

For this study, we used around 200 sites where our automated login continuously succeeded during the span of the experiments. To evaluate the reasons why the automatic tool failed on the other sites and to improve it, we manually spot checked them. The three main reasons were: login requires solving a captcha, login requires clicking a link in a mail, and the account was blocked.

The four experiments ran independently on one of three identical crawling machines. Each of the four experiments requested sessions from the account framework and, after receiving a session, crawled the site twice independently in parallel, once with the received session (S_{auth}) and once with a fresh browsing profile (S_{noauth}), to be able to compare results one would receive if one would either crawl with being logged-in or not. As real users interact with websites and not only visit the landing page, each crawl was configured to crawl up to 1,000 same-site URLs with a timeout of 24 hours. Our crawlers use Playwright v1.33 in its default configuration. The crawlers visit the current URL, wait until the `load` event fires (max 30 seconds), wait for another five seconds, and execute all modules. The collect links module uses `page.locator('a[href]')` and we add all same-site links up to a maximum depth of two. In the case of the authenticated crawls, we excluded common logout URLs to avoid accidental logout during the experiment. We note that the collected same-site URLs might redirect cross-site and such responses are later excluded from the analysis.

7. Results

In this section, we present the results of our study. We start with general insights from our dataset applicable to all experiments. Then, we present the results of each of the four experiments separately.

7.1. General Crawling Statistics

Each experiment ran in the same timespan of two weeks in July 2023. The number of sites each experiment processed was between 203 and 212. The experiments processed different amounts of sites because some accounts stopped working or the automated login started to fail during these two weeks. We decided to analyze the 200 sites processed by all four experiments. Two sites had no additional URLs discovered for all experiments, and further investigation on these sites showed that they only rely on buttons instead of links. On the other sites, no same-site URLs could be collected in at least one state and experiment, and we decided to recrawl these once in the corresponding experiments. This might have been caused by a temporal issue, such as a failed loading of the landing page. On average, 12 sites per experiment were recrawled as they had not collected any URLs in at least one state, and after recrawling, an average of 7.5 had at least one state with no collected URLs left.

The average and median Tranco (ID N76GW from July 4, 2023) rank of the 200 considered sites is 15,137 and 3,038 respectively; only one site is not in the Tranco list. 51 sites are in the highest bucket (top 1,000) of CrUX (April 2023), with the majority (110) in the second bucket (top 5,000) and only 24 and 15 in lower buckets (10,000 and 50,000).

Each experiment ran in pairs, i.e., each site was visited by one experiment in parallel in both states. On average, the time spent to crawl one site ranged from 3h19m (Javascript inclusion experiment) to 16h27m (Security header experiment). The average time of the two states over all experiments was 8h44m for S_{noauth} and 8h31m for S_{auth}. A total of 52 site-state pairs ran into the 24-hour limit and got terminated early. 25 of these belong to S_{auth} and 27 to S_{noauth}.

On average, 840 URLs were collected in S_{noauth} and 820 URLs in S_{auth}. An average of 809 and 764 were crawled successfully. The numbers for collected/crawled URLs range from (771/614) for the CXSS experiment to (890/876) for the Javascript inclusion experiment. We do not exclude sites that did not collect any additional same-site URLs to visit after the landing page for at least one state as this might be a valid result, e.g., authenticated users are automatically redirected to a user portal with only buttons and no links. For a small number of sites, we did not collect any same-site data for at least one state in some experiments as they redirected to a non-same-site URL, or the initial page did not load successfully. Thus, we exclude such sites from the experiment analysis, and each experiment has between 195 to 198 sites to analyze.

7.2. Client-Side XSS

The key question to ask for the detection of client-side XSS is to what extent lacking an authenticated state reduces the number of sites found to be vulnerable. This may stem from some URLs not being accessible without login or the crawler simply not being able to find a particular URL that is found in the authenticated state. Additionally, prior work has investigated the overall number [28, 30] and complexity [45] of dataflows as an indicator of potential for XSS flaws. A dataflow is information coming from one source ending up in one sink invocation. Sink invocations such as `document.write(location.href + document.cookie + location.href)` can use several sources and even the same source several times, thus we decided to report the number of sink invocations. Therefore, we also investigate if the number of sink invocations or the distribution across sinks changes between the two states.

TABLE 1: Sites with at least one direct CXSS sink.

Sink	S_{noauth}	S_{auth}	Total
innerHTML	145	153	164
script.src	152	149	163
eval	68	79	81
script.text	9	12	13
document.write	9	6	9
outerHTML	3	3	4
setTimeout	1	1	1

We collected a total of 38,105,442 sink invocations, 22,294,095 for S_{auth} and 15,811,347 for S_{noauth}. Out of all sink invocations 8,781,897 filter at least part of the source (any of hasEscaping, hasEncodigURI, hasEncodingURIComponent is true for at least one entry in the sink invocation source list), out of these 6,358,454 filter all source entries, the remaining 29,323,545 sink invocations are *Plain* and do not use any encoding or filtering. Table 1 shows the number of sites with at least one sink invocation that *could* result in XSS for both states. Table 2 shows the sink invocations categorized by sink invocation group. The table shows both the number of sites with at least one such sink invocation and the total number of sink invocations.

For most sinks both the number of unique sites and the number of total sink invocations is higher for S_{auth}. *Potential CXSS* invocations are all invocations to the sinks shown in Table 1 and for this group more invocations exist for S_{noauth}. The *generator* row shows all sink invocations that use a supported sink and are tainted by a source that is supported by the exploit generator. Here more invocations exist for S_{auth} again, the reason is that nearly half of the potential CXSS flows for S_{noauth} used the `XHR.response` source that is not supported by the exploit generator. The last row *exploitable* shows all sites and sink invocations that resulted in successful exploits.

Table 3 presents all generated exploits. In total we discovered seven unique vulnerable sites. Six are vulnerable

TABLE 2: Sites with at least one sink invocation and absolute count of sink invocation.

	Sites			Sink Invocations	
	S_{noauth}	S_{auth}	Total	S_{noauth}	S_{auth}
Any	193	191	195	15,811,347	22,294,095
Plain	192	190	194	12,475,288	16,848,257
Potential CXSS	158	162	172	774,457	667,804
Generator	140	145	159	336,492	492,504
Exploitable	4	6	7	418	2,330

TABLE 3: Sites with CXSS exploits.

	Exploits			Sites		
	S_{noauth}	S_{auth}	Total	S_{noauth}	S_{auth}	Total
PCXSS	420	2,465	2,885	3	5	6
RCXSS	2	18	20	1	1	1
Total	422	2,483	2,905	4	6	7

in S_{auth}, and four in S_{noauth}, three sites are affected in both states. Six of the vulnerable sites are affected by a persistent flaw and one site is affected by a reflected client-side XSS. The vulnerable sink is always *innerHTML*.

One of the main questions is whether the discovered vulnerabilities only were not found in the respective other state or whether the vulnerabilities do not exist in the other state. To answer this question, we ran every confirmed exploit in the respective other state. 66% of the exploits found in S_{auth} could be exploited in the S_{noauth} crawler, whereas 91% of the exploits found in S_{noauth} could be exploited in the S_{auth} crawler. Both states could confirm exploits for 3 sites. Out of these, 2 also had working exploits in their respective other state in the original crawl. One site that also had exploits in both states could not confirm these in the respective other states. Taking all information together, 2 sites were only ever exploitable in S_{auth} whereas 0 sites were only ever exploitable in S_{noauth}.

Although CXSS exploits were rare on the studied sites, which shows that simple attacks are getting harder on the web, crawling from a logged-in perspective provides us with vulnerable URLs which could not be reached otherwise. This means that studies from an unauthenticated perspective [28, 30, 43, 45] likely underreport the prevalence of XSS vulnerabilities. On the other hand, only looking at a logged-in context also misses sites that have potential dangerous sink invocations.

7.3. Security Headers

We now outline the results of our security header analysis regarding observed usage, consistency within each state, and the comparative analysis for URLs visited in both states.

7.3.1. Usage and Security. In total, we collected 1.7 million top-level responses. To ensure the reliability of the results, we filtered out the responses with rendering or network errors. Additionally, we removed the responses that redirect to a different site. Ultimately, we analyzed over 1.5 million top-level responses, including 797k Strict-Transport-Security headers, 871k X-Frame-Options headers, and 525k Content-Security-Policy headers.

We analyzed all responses, assessing each considered header's presence, security, and absence. Then, we grouped the responses by site and state, as shown in Table 4. This approach gives us first impressions of the potential differences between the headers in both states.

TABLE 4: Sites that used a header securely, insecurely, or did not specify it at least once.

Header	Using	Secure	Insecure	Missing
CSP XSS mitigation	57	28	44	190
- only S_{noauth}	4	4	3	5
- only S_{auth}	2	5	1	1
CSP framing control	84	83	2	192
- only S_{noauth}	9	9	0	4
- only S_{auth}	2	2	0	0
CSP TLS enforcement	48	48	0	194
- only S_{noauth}	2	2	0	4
- only S_{auth}	0	0	0	0
Strict-Transport-Security	149	147	8	174
- only S_{noauth}	7	6	2	11
- only S_{auth}	0	0	0	3
X-Frame-Options	167	164	9	178
- only S_{noauth}	5	5	1	10
- only S_{auth}	2	2	2	2

Table 4 shows the site-specific usage of response headers. For the examined headers, the X-Frame-Options header appears on most pages, with 167 sites using it at least once. It is also the most frequently used header, appearing on average in 56% of responses. The Content-Security-Policy header is the least used. On average, less than 12% of the responses had a specified CSP for mitigating cross-site scripting attacks. While CSP for XSS mitigation is the least used, it is also the most frequently misused header, with 44 sites specifying a non-safe policy.

Our analysis reveals that the header usage and security are generally similar between S_{auth} and S_{noauth}. However, some sites employ security headers differently. For instance, we observed seven sites that specified the Strict-Transport-Security in S_{noauth} exclusively, while others deployed the X-Frame-Options header in S_{auth} only. Despite these differences, we observed no significant changes in the average frequency of deploying a header on the pages of a site between the two states. While there are some differences between S_{auth} and S_{noauth}, they largely depend on the implementation of the specific server, and most sites show minimal disparities in the usage and security of headers between the two states. Notably, all headers are frequently absent on at least one page per site, implying inconsistent deployment. We note that more sites with missing headers are contained in the unauthenticated dataset, implying that to study the absence of headers, this is a better choice.

7.3.2. Intra-State Consistency. Roth et al. studied header inconsistencies, which can be attributed to deterministic and non-deterministic factors [36]. To validate the consistency of our data and eliminate potential influence from non-deterministic factors, we examine the intra-consistency of the headers. We group the responses by site, visited URL, and state and compare the headers' semantics using equivalence relations. Each equivalence class is assigned a numeric value, such that higher values indicate better security. We consider a header intra-consistent for a given URL if all five

TABLE 5: Sites with intra-inconsistent security headers.

Header	Inconsistent	Inconsistent (w/o missing)
CSP XSS mitigation	6	0
- only S_{noauth}	2	0
- only S_{auth}	1	0
CSP framing control	23	0
- only S_{noauth}	2	0
- only S_{auth}	9	0
CSP TLS enforcement	11	0
- only S_{noauth}	1	0
- only S_{auth}	4	0
Strict-Transport-Security	61	4
- only S_{noauth}	16	0
- only S_{auth}	11	2
X-Frame-Options	55	1
- only S_{noauth}	8	0
- only S_{auth}	15	0

visits result in the same equivalence class for the header. We present an overview of our findings in Table 5.

Many sites exhibited an intra-inconsistent header on at least one URL. For instance, 61 sites had an inconsistent HSTS header. In Table 5, the last column examines the header inconsistencies, ignoring differences in equivalence classes caused by missing headers. The results highlight that the majority of inconsistencies are attributed to missing headers and aligns with the results of Roth et al. [36]. Notably, the frequency of inconsistent headers is low - our findings suggest that over 99% of URLs had consistent headers. The high level of consistency indicates that our results remain unaffected by non-deterministic factors.

Similarly to the usage and security of headers, our analysis indicates no significant differences in the intra-consistency between authenticated and unauthenticated browsers. For instance, while the HSTS had more inconsistent headers in S_{noauth}, the XFO and CSP headers had more inconsistencies in S_{auth}. However, these inconsistencies are attributed to missing headers, and there is no discernible pattern of one state having more inconsistencies.

7.3.3. Inter-State Comparison.

We compare the security of the two states by following the semantic relations of Roth et al. We exclude the intra-inconsistent responses for both states, group the remaining responses by the visited URL, and compare the security for each header. We subtract the headers' equivalence class to get a numeric value that indicates which state has a more secure header.

Table 6 presents an overview of the number of sites per state with a header with better security on at least one of their URLs. We observe that only a handful of sites exhibit different levels of security on the same URL. In total, 24 sites have at least one page with strictly better security for S_{auth} than S_{noauth}, and 24 have it vice-versa. The biggest contributors are XFO and HSTS for both classes. Notably, six sites are better in terms of security in both states at least once. This suggests a hidden intra-test inconsistency rather than an intentional difference between the states on different URLs. Again, missing headers are the most prevalent cause, as highlighted by the last two columns. Moreover, only 422 out of 46,992 (0.9%) URL pairs are affected.

TABLE 6: Sites with a better header security in either state.

Header	Better security			w/o missing	
	S_{noauth}	S_{auth}	Both	S_{noauth}	S_{auth}
CSP XSS mitigation	3	2	0	2	0
CSP framing control	8	4	2	1	0
Strict-Transport-Security	14	13	5	4	2
X-Frame-Options	13	16	2	1	2
Any	24	24	6	4	6

Overall, our findings for security headers suggest no substantial differences in the adoption and potential misconfigurations between the two states. As prior work [36] noted around 8% of start pages do not deliver intra-test consistent results, the impact of randomness on serve seems much higher than any lack of authentication might have on the experiment results.

7.4. JavaScript Inclusions

In total, our crawlers parsed 42,437,161 scripts. Of these, 22,607,693 were observed in S_{auth} and 19,829,468 in S_{noauth}, totalling 1,719,724 hash unique scripts. In the following, we dive into the different types of scripts being parsed, the number of libraries we found, and how the findings impact a site's ability to deploy CSP.

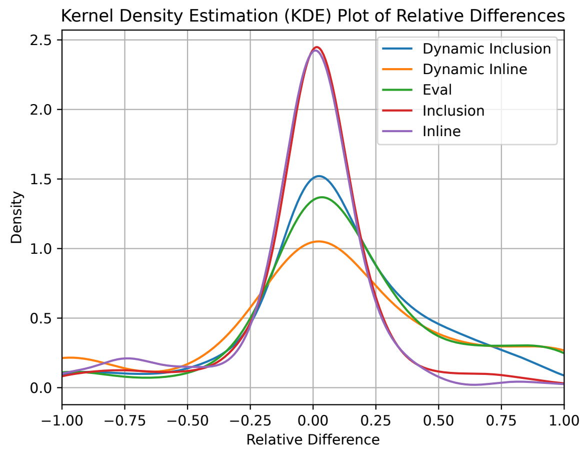
7.4.1. Parsed JavaScript and Library Usage.

To analyze the impact of login state on the observed JavaScript and potential attack surface, we look at the amount of individual scripts and parser invocations across S_{auth} and S_{noauth}. In the following, we refer to JavaScript parser invocations as *script parsings*. Additionally, we compare the usage of known vulnerable libraries.

Table 7 shows all script parsings and unique scripts grouped by the two states and by the different script parsing types. Overall, the number of unique scripts in S_{auth} is 4.9% greater than in S_{noauth}. Note that the total unique scripts are not the sum of the two states since some scripts occurred in both states. It is worth noting, though, that approximately 10% of all unique scripts were only observed in one or the other state. All 198 sites contained more unique scripts in S_{auth} compared to S_{noauth}.

Figure 2 shows the density for the relative differences between the two states over the sites for each of the *five* script parsing types. A value of -1 means only the S_{noauth} had script parsings, a value of 0 means both states had the same number of script parsings, and a value of +1 means only the S_{auth} state had script parsings. The highest density is around 0, showing that most sites have a similar amount of script invocations for both states. However, we can see that the three types *Dynamic Inclusion*, *Dynamic Inline*, and

Figure 2: KDE of relative differences for script parsings.



eval are shifted to the right. For these types 66, 53 and 60 sites respectively have more than 20% more invocations for S_{auth}. The other two types are more closely centered around zero.

The number of unique scripts identified as libraries in S_{auth} (1,864) is 51.05% greater than in S_{noauth} (1,234). In our experiment, 1,535 of the scripts containing libraries in S_{auth} had at least one known vulnerability while only 553 had a known vulnerability in S_{noauth}. However, this difference is mainly caused by one outlier, `cnbc.com`, which has 986 unique scripts containing vulnerable libraries in S_{auth} and only 4 for S_{noauth}. Even though these scripts only differ slightly, they are classified as unique scripts since they have different hashes. The number of websites that include at least one vulnerable library on at least one crawled URL is similar (143 S_{auth}, 145 S_{noauth}).

TABLE 7: Scripts parsings grouped by type and state.

Type	Script Parsings	Unique scripts
Total	42,437,161	1,719,724
- S_{noauth}	19,829,468	934,545
- S_{auth}	22,607,693	980,569
Inline	4,074,246	647,133
- S_{noauth}	2,116,945	413,988
- S_{auth}	1,957,301	301,572
Dynamic Inline	1,140,202	73,077
- S_{noauth}	484,319	34,203
- S_{auth}	655,883	44,057
Inclusion	6,216,870	133,060
- S_{noauth}	2,624,347	73,912
- S_{auth}	3,592,523	79,543
Dynamic Inclusion	5,753,260	455,301
- S_{noauth}	2,408,434	189,677
- S_{auth}	3,344,826	292,294
Eval	25,252,583	414,011
- S_{noauth}	12,195,423	224,317
- S_{auth}	13,057,160	265,112

TABLE 8: Sites that include at least one script of the given type on at least on crawled URL.

	Inline	Dynamic Inline	Eval	Inclusion	Dynamic Inclusion
Total	198	158	179	198	198
- S_{noauth}	197	147	175	197	197
- S_{auth}	197	150	175	198	197

TABLE 9: Number of unique third-party scripts, third parties, and known trackers.

	Unique Third-Party Scripts	Unique Third Parties	Unique Trackers	Average Third Parties per Site	Average Trackers per Site
Total	507,948	1,231	219	25.93	9.54
- S_{noauth}	216,952	1,053	181	19.90	6.52
- S_{auth}	322,525	1,146	209	23.68	8.88

7.4.2. Impact on CSP and Privacy Studies. To analyze the impact of login state on the ability to deploy a secure and concise CSP, we compare the JavaScript usage in S_{auth} and S_{noauth}. As described in Section 5.3, JavaScript parser invocations are grouped into categories. Websites with *Inline* or *Dynamic Inline* have to rely on *unsafe-inline* or use nonces, which are non-trivial for third parties to use. Notably, they could use `strict-dynamic`, yet as shown by Steffens et al. [42], many third parties added event handlers which cannot be enabled through nonces, which in turn is a prerequisite for `strict-dynamic`. A website containing scripts of the category *eval* must deploy CSP with the *unsafe-eval* directive. As depicted by Table 8, there are no significant differences between S_{auth} and S_{noauth} regarding this unsafe practices. However, it is worth noting that crawling in one or the other state would miss up to 11 sites (for dynamic inline), i.e., potentially have results that are off by 5%.

Similarly, websites that use *Inclusion* or *Dynamic Inclusion* scripts must allow all third parties in the *script-src*. Table 9 shows the amount of unique third-party scripts and the number of third-parties grouped by the two states. In addition, it shows the number of known tracking entities defined by the Disconnect Tracker Protection List [12]. Overall, the number of third parties in S_{auth} is 8.83% greater than in S_{noauth}. Similarly, the number of unique third-party scripts in S_{auth} is 48.66% greater than in S_{noauth}. A total of 102 sites have more unique third parties in S_{auth}, whereas only 46 have more in S_{noauth}. On average, sites have 23.68 third parties in S_{auth} and only 19.90 in S_{noauth}. Note that the combined average is higher, as each site may have third-party inclusions which happen only in one state, therefore the union of included third parties is higher than the two averages. Notably, our findings imply that works that automatically curate CSPs or study roadblocks [14, 31, 42] might be significantly off for allowlist requirements. Moreover, works that aim to understand third-party tracking [7, 29, 34] may suffer from similar limitations as we discovered 15.47% more unique tracking entities in S_{auth}.

TABLE 10: Number of postMessage handlers.

State	Handlers	Hash-Unique	AST-Unique	Unique Sites
S_{noauth}	23,616	5,736	493	138
S_{auth}	49,795	6,835	899	174
Total	73,411	12,280	1,133	183

The vast majority of sites have more scripts, third parties, and trackers when crawled in S_{auth} compared to S_{noauth}. Thus, information gathered about JavaScript usage while crawling unauthenticated does not automatically and fully translate to authenticated users. The effect is limited when judging insecure practices like `unsafe-inline`. However, both CSP allowlists [5, 42, 49] and tracking research [7, 29, 34] may be off.

7.5. PostMessages

For the postMessage analysis we first focus on the general usage in both states. Then, we look at potentially vulnerable and verified exploitable postMessage handlers.

Table 10 provides an overview of the collected PostMessage handlers. Our crawlers collected a total of 73,411 handlers. Notably, more than twice as many (49,795 vs. 23,616) were detected in S_{auth}. However, sites may re-use the same handler repeatedly, which is shown by the fact that considering hash-unique handlers, 5,736 handlers were detected in S_{noauth} and 6,835 occurred in S_{auth}. Moreover, even hash uniqueness is not necessarily a criterion for unique handlers. This is because they might merely contain a timestamp or variable which points to the current URL, i.e., although their code is effectively the same, the hash differs. We, therefore, computed the AST hash of each handler. To do this, we parse the handler code with esprima and only hash the resulting tree *without* values. Considering these AST-unique handlers, the difference is even larger with 493 to 899 handlers, respectively. For 83 hash-unique handlers, we could not determine an AST-hash as esprima failed to parse them. We note that in total, 183 sites registered a postMessage handler, and merely visiting sites in one or the other state would not have allowed us to capture all the handlers we did.

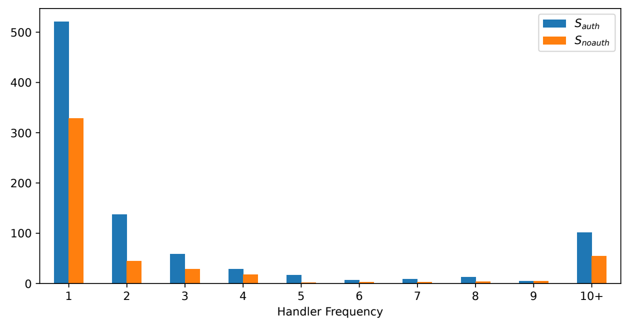


Figure 3: Usage of hash-equivalent postMessage handlers.

TABLE 11: Sites with postMessage exploit candidates.

Sink	Exploit Candidates S_{noauth}	Exploit Candidates S_{auth}	Candidate Sites S_{noauth}	Candidate Sites S_{auth}	Vulnerable Sites S_{noauth}	Vulnerable Sites S_{auth}
document.cookie	1	2	1	2	0	0
document.write	0	2	0	1	0	0
innerHTML	2	4	1	1	0	0
Storage	0	6	0	3	0	1
Total	3	14	2	6	0	1

In terms of finding potential vulnerabilities (e.g., XSS or state manipulation), it is important to understand how many handlers are *only* included within one state. Figure 3 shows a histogram of how many AST-unique handlers were included how often for both states. Most AST-unique handlers only occurred in *one* state on *one* site. However, 332 handlers occurred on multiple sites and 259 handlers occurred in both states. 118 handlers occurred on two sites and one handler was found on a total of 117 distinct sites. Out of the handlers that occurred multiple times, 172 occurred exactly twice, others however appeared up to 20,000 times. These mostly relate to third-party code, e.g., Google Syndication.

Most of the discovered handlers are harmless and do not possess any flow to a dangerous sink. Table 11 shows all exploit candidates generated by our exploit generator. In the end, only two exploits worked, both of which affected a single site in the authenticated state.

For postMessages, our analysis highlights that the number of hash- and AST-unique handlers increases in the authenticated state. That is, any study on postMessages usage should attempt to perform a login to cover as many handlers as possible. However, nine sites were only discovered in S_{noauth}. Moreover, given the findings of both Steffens and Stock [44] and us, postMessage vulnerabilities are fortunately very rare on the modern web. Therefore, the implications of the authentication state are limited when it comes to finding exploitable handlers.

8. Discussion

Here we discuss our main insights, list limitations, and ethical considerations, and recommend actions future studies using web crawling in the area of security should take.

8.1. Main Insights

Crawling in an authenticated state is generally similar to crawling with a fresh unauthenticated browser. However, some sites redirect or serve special *user portals* to logged-in users with no or almost no links and instead rely on buttons and other HTML elements to provide their functionality. Thus, we observed fewer average collected URLs per site in S_{auth}. In addition, one has to be careful to avoid logging out accidentally, and the challenging task of obtaining a valid user session to begin the crawl has to be solved.

In general, the differences between authenticated and non-authenticated users can matter but do not necessarily have to. In short, how well the results of non-authenticated users generalize to authenticated users highly depends on the subject of study. For most of our experiments, we saw moderate to significant increased values for authenticated users. For example, 14.01% higher number of script parsings, 35.05% higher number of plain sink invocations, and 82.35% higher number of observed AST-unique postMessage handlers for S_{auth}. At the same time, the general usage and security of security headers were mostly the same. Similarly, the intra-test consistency was nearly identical for both states, and the inter-test comparison could also not identify one state to be more secure in their usage of security headers. We hypothesize that one of the reasons for the almost nonexistent differences in security headers is that most sites set these headers application-independent and origin (or site)-wide at a reverse proxy instead of setting these headers within the application.

In addition, while we have seen some clear trends for certain security indicators, it is not the case that the authenticated state is always better or worse in terms of security of the observed entity. Instead, the security indicators can differ for each site, and outliers exist in both directions. Thus, to report clear trends and not only single outlying sites, one must study a large enough set of sites, investigate outliers, and correctly accommodate them in the used metrics. In general, however, we observed that the authenticated state has a larger observable attack surface and more vulnerabilities. We explain this by the fact that authenticated users have access to more functionality on a site while usually not losing access to the parts of a site accessible to non-authenticated users.

8.2. Limitations

As with any other study, ours comes with limitations. Foremost, the selection of sites might not be suitable to generalize to the whole web. Due to the challenging and labor-intensive task of creating accounts, we could only study 200 sites. We choose popular websites from both Tranco and CrUX as the basis of site selection. In addition, we only included sites where our automated tool could find a login and registration form and where automatic login succeeded. Thus, our study is affected by selection and survivorship bias. Still, we think the set of sites is meaningful as it portrays a wide range of popular sites on which users spend much time (median Tranco Rank of 3,038).

Secondly, the crawling of each site is limited (e.g., we only test a site for a maximum of 1000 URLs or 24 hours and did not attempt to visit intentionally invalid URLs which may have different security configurations [48]), and random effects might affect both states differently. We tried to minimize all such errors by starting the crawl in both states for each site simultaneously and on the same machine. However, as each state crawled individually, they might have diverged from one another during the crawl. In addition, while we avoided logout URLs, we cannot guarantee that all crawls that started with an authenticated session finished the crawl authenticated.

Thirdly, the number of security-relevant issues and indicators we could study in this work is limited, and our results might not transfer to other subjects of study. For example, we could not study any server-side security issues.

8.3. Ethical Considerations

For this study, we created accounts on various sites and open-source our toolchain to ease the process of performing web security studies with logged-in accounts. The first question is about the harm done by our experiment, and the second is about the potential harm of releasing our toolchain. We only created one account per site, and all tested sites were popular. In all our attacks and experiments, we only investigated our own accounts and only studied client-side security issues. Thus no other users of the sites were influenced by our study. For the second question, we think the potential benefit for researchers is higher than the potential for abuse. The account creation process still requires a human-in-the-loop, and similar tools already exist for attackers.

Our analysis uncovered vulnerabilities in real-world sites that may affect many users. Even though the flaws were discovered with publicly available tools, we notified the affected site operators to ensure these cannot be exploited.

8.4. Recommendations for Future Studies

We recommend each web security study to reflect on whether a difference between authenticated and non-authenticated sessions could be important for their outcome and their interpretation thereof. While we have seen a trend of a larger attack surface and more vulnerabilities in three of the four experiments performed for this study, we cannot know whether this generalizes to other study subjects. In addition, we discovered unique observations in the non-authenticated state for all our measurements, meaning a pure authenticated crawl would also miss out on affected sites.

In a perfect world, each crawl would test thousands of websites in both states. However, studying security issues with authenticated sessions has many challenges and is a time-intensive task that will never scale to the same dimensions as crawling with fresh browser sessions. We thus suggest for new security issues to perform both a large-scale crawl with fresh browser sessions and a smaller-scale comparison of authenticated and non-authenticated sessions. This way, the potential error introduced through unauthenticated crawls can be estimated. To ease such studies, we open-source our account framework pipeline.

9. Conclusion

Given the importance of the web in our daily lives, it is imperative to accurately measure its state of security. We set out to investigate the impact that (lacking) authentication can have on web security measurements. Our comparative analysis of about 200 sites revealed that unauthenticated results can be significantly skewed depending on the type of measurement and research question.

In particular, while overall trends in security header adoption can be measured in fresh browsing sessions, vulnerable libraries and exploitable XSS flaws are less prevalent before authentication. Moreover, our results for third-party inclusions show that in order to obtain a comprehensive picture for studies that require precise numbers (such as CSP allowlists or tracking analyses), *both* states should be considered where possible. Moreover, generally speaking, more functionality is exposed post-login, which allows for more thorough analyses. To enable other researchers to conduct their studies with login, we will make our semi-automated account framework available.

Availability

The account framework and the crawling code for the four experiments is available at: https://github.com/cispa/login-security-landscape.

Acknowledgements

We thank our anonymous shepherd and the reviewers for their valuable feedback.

This work was conducted in the scope of a dissertation at the Saarbrücken Graduate School of Computer Science.

References

[1] Syed Suleman Ahmad, Muhammad Daniyal Dar, Muhammad Fareed Zaffar, Narseo Vallina-Rodriguez, and Rishab Nithyanand. "Apophanies or Epiphanies? How Crawlers Impact Our Understanding of the Web". In: *The Web Conference*. 2020. DOI: 10.1145/3366423.3380113.

[2] Waqar Aqeel, Balakrishnan Chandrasekaran, Anja Feldmann, and Bruce M. Maggs. "On Landing and Internal Web Pages: The Strange Case of Jekyll and Hyde in Web Performance Measurement". In: *ACM Internet Measurement Conference*. 2020. DOI: 10.1145/3419394.3423626.

[3] Richard Atterer, Monika Wnuk, and Albrecht Schmidt. "Knowing the User's Every Move: User Activity Tracking for Website Usability Evaluation and Implicit Interaction". In: *World Wide Web Conference*. 2006. DOI: 10.1145/1135777.1135811.

[4] Stefano Calzavara, Hugo Jonker, Benjamin Krumnow, and Alvise Rabitti. "Measuring Web Session Security at Scale". In: *Computers & Security* 111 (2021). DOI: 10.1016/j.cose.2021.102472.

[5] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. "Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild". In: *ACM SIGSAC Conference on Computer and Communications Security*. 2016. DOI: 10.1145/2976749.2978338.

[6] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. "A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web". In: *USENIX Security Symposium*. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/calzavara.

[7] Darion Cassel, Su-Chin Lin, Alessio Buraggina, William Wang, Andrew Zhang, Lujo Bauer, Hsu-Chun Hsiao, Limin Jia, and Timothy Libert. "Omni-Crawl: Comprehensive Measurement of Web Tracking With Real Desktop and Mobile Browsers". In: *Proceedings on Privacy Enhancing Technologies Symposium* (2022). DOI: 10.2478/popets-2022-0012.

[8] *Content Security Policy Level 3*. W3C Working Draft. 2023. URL: https://www.w3.org/TR/CSP3/.

[9] Johannes Dahse and Thorsten Holz. "Simulation of Built-in PHP Features for Precise Static Code Analysis". In: *Network and Distributed System Security Symposium*. 2014. DOI: 10.14722/ndss.2014.23262.

[10] Joe DeBlasio, Stefan Savage, Geoffrey M. Voelker, and Alex C. Snoeren. "Tripwire: Inferring Internet Site Compromise". In: *ACM Internet Measurement Conference*. 2017. DOI: 10.1145/3131365.3131391.

[11] Nurullah Demir, Matteo Große-Kampmann, Tobias Urban, Christian Wressnegger, Thorsten Holz, and Norbert Pohlmann. "Reproducibility and Replicability of Web Measurement Studies". In: *The Web Conference*. 2022. DOI: 10.1145/3485447.3512214.

[12] Disconnect. *Disconnectme/Disconnect-Tracking-Protection*. 2023. URL: https://github.com/disconnectme/disconnect-tracking-protection.

[13] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. "The Cookie Hunter: Automated Black-box Auditing for Web Authentication and Authorization Flaws". In: *ACM SIGSAC Conference on Computer and Communications Security*. 2020. DOI: 10.1145/3372297.3417869.

[14] Mattia Fazzini, Prateek Saxena, and Alessandro Orso. "AutoCSP: Automatically Retrofitting CSP to Web Applications". In: *IEEE International Conference on Software Engineering*. Vol. 1. IEEE, 2015. DOI: 10.1109/ICSE.2015.53.

[15] Google. *Chrome UX Report*. Chrome Developers. 2023. URL: https://developer.chrome.com/docs/crux/.

[16] Jeff Hodges, Collin Jackson, and Adam Barth. *HTTP Strict Transport Security (HSTS)*. Request for Comments RFC 6797. Internet Engineering Task Force, 2012. 46 pp. DOI: 10.17487/RFC6797.

[17] *HTML Standard XFO*. 2023. URL: https://html.spec.whatwg.org/multipage/document-lifecycle.html#the-x-frame-options-header.

[18] Louis Jannett, Vladislav Mladenov, Christian Mainka, and Jörg Schwenk. “DISTINCT: Identity Theft Using In-Browser Communications in Dual-Window Single Sign-On”. In: *ACM SIGSAC Conference on Computer and Communications Security*. 2022. DOI: 10.1145/3548606.3560692.

[19] Martin Johns, Björn Engelmann, and Joachim Posegga. “XSSDS: Server-Side Detection of Cross-Site Scripting Attacks”. In: *Annual Computer Security Applications Conference*. 2008. DOI: 10.1109/ACSAC.2008.36.

[20] Hugo Jonker, Stefan Karsch, Benjamin Krumnow, and Marc Sleegers. “Shepherd: A Generic Approach to Automating Website Login”. In: *Workshop on Measurements, Attacks, and Defenses for the Web*. 2020. DOI: 10.14722/madweb.2020.23008.

[21] Jordan Jueckstock, Shaown Sarker, Peter Snyder, Aidan Beggs, Panagiotis Papadopoulos, Matteo Varvello, Benjamin Livshits, and Alexandros Kapravelos. “Towards Realistic and Reproducible Web Crawl Measurements”. In: *The Web Conference*. 2021. DOI: 10.1145/3442381.3450050.

[22] Zifeng Kang, Song Li, and Yinzhi Cao. “Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites”. In: *Network and Distributed System Security Symposium*. 2022. DOI: 10.14722/ndss.2022.24308.

[23] Soheil Khodayari and Giancarlo Pellegrino. “It’s (DOM) Clobbering Time: Attack Techniques, Prevalence, and Defenses”. In: *IEEE Symposium on Security and Privacy*. 2023. DOI: 10.1109/SP46215.2023.10179403.

[24] Amit Klein. “DOM Based Cross Site Scripting or XSS of the Third Kind”. In: *Web Application Security Consortium, Articles* 4 (2005). URL: http://www.webappsec.org/projects/articles/071105.shtml.

[25] David Klein, Marius Musch, Thomas Barber, Moritz Kopmann, and Martin Johns. “Accept All Exploits: Exploring the Security Impact of Cookie Banners”. In: *Annual Computer Security Applications Conference*. 2022. DOI: 10.1145/3564625.3564647.

[26] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. “Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web”. In: *Network and Distributed System Security Symposium*. 2017. DOI: 10.14722/ndss.2017.23414.

[27] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. “Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation”. In: *Network and Distributed System Security Symposium*. 2019. DOI: 10.14722/ndss.2019.23386.

[28] Sebastian Lekies, Ben Stock, and Martin Johns. “25 Million Flows Later: Large-Scale Detection of DOM-based XSS”. In: *ACM SIGSAC Conference on Computer and Communications Security*. 2013. DOI: 10.1145/2508859.2516703.

[29] Timothy Libert. “An Automated Approach to Auditing Disclosure of Third-Party Data Collection in Website Privacy Policies”. In: *World Wide Web Conference*. 2018. DOI: 10.1145/3178876.3186087.

[30] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. “Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting”. In: *Network and Distributed System Security Symposium*. 2018. DOI: 10.14722/ndss.2018.23309.

[31] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. “CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites”. In: *ACM SIGSAC Conference on Computer and Communications Security*. 2016. DOI: 10.1145/2976749.2978384.

[32] Jannis Rautenstrauch, Giancarlo Pellegrino, and Ben Stock. “The Leaky Web: Automated Discovery of Cross-Site Information Leaks in Browsers and the Web”. In: *IEEE Symposium on Security and Privacy*. 2023. DOI: 10.1109/SP46215.2023.00067.

[33] *Retire.Js*. URL: https://retirejs.github.io/retire.js/.

[34] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. “Detecting and Defending Against Third-Party Tracking on the Web”. In: *Symposium on Networked Systems Design and Implementation*. 2012. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/roesner.

[35] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. “Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies”. In: *Network and Distributed System Security Symposium*. 2020. DOI: 10.14722/ndss.2020.23046.

[36] Sebastian Roth, Stefano Calzavara, Moritz Wilhelm, Alvise Rabitti, and Ben Stock. “The Security Lottery: Measuring Client-Side Web Security Inconsistencies”. In: *USENIX Security Symposium*. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/roth.

[37] Kimberly Ruth, Aurore Fass, Jonathan Azose, Mark Pearson, Emma Thomas, Caitlin Sadowski, and Zakir Durumeric. “A World Wide View of Browsing the World Wide Web”. In: *ACM Internet Measurement Conference*. 2022. DOI: 10.1145/3517745.3561418.

[38] Kimberly Ruth, Deepak Kumar, Brandon Wang, Luke Valenta, and Zakir Durumeric. “Toppling Top Lists: Evaluating the Accuracy of Popular Website Lists”. In: *ACM Internet Measurement Conference*. 2022. DOI: 10.1145/3517745.3561444.

[39] SAP. *Project Foxhound*. 2023. URL: https://github.com/SAP/project-foxhound.

[40] Sooel Son and Vitaly Shmatikov. “The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites.” In: *Network and Distributed System Security Symposium*. 2013. URL: https://www.ndss-symposium.org/ndss2013/postman-

always - rings - twice - attacking - and - defending - postmessage-html5-websites.

[41] Sid Stamm, Brandon Sterne, and Gervase Markham. “Reining in the Web with Content Security Policy”. In: *World Wide Web Conference*. 2010. DOI: 10.1145/1772690.1772784.

[42] Marius Steffens, Marius Musch, Martin Johns, and Ben Stock. “Who’s Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI”. In: *Network and Distributed System Security Symposium*. 2021. DOI: 10.14722/ndss.2021.24028.

[43] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. “Don’t Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild”. In: *Network and Distributed System Security Symposium*. 2019. DOI: 10.14722/ndss.2019.23009.

[44] Marius Steffens and Ben Stock. “PMForce: Systematically Analyzing postMessage Handlers at Scale”. In: *ACM SIGSAC Conference on Computer and Communications Security*. 2020. DOI: 10.1145/3372297.3417267.

[45] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. “From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting”. In: *ACM SIGSAC Conference on Computer and Communications Security*. 2015. DOI: 10.1145/2810103.2813625.

[46] Avinash Sudhodanan, Soheil Khodayari, and Juan Caballero. “Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks”. In: *Network and Distributed System Security Symposium*. 2020. DOI: 10.14722/ndss.2020.24278.

[47] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. “Measuring Login Webpage Security”. In: *Symposium on Applied Computing*. 2017. DOI: 10.1145/3019612.3019798.

[48] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. “Raising the Bar: Evaluating Origin-wide Security Manifests”. In: *Annual Computer Security Applications Conference*. 2018. DOI: 10.1145/3274694.3274701.

[49] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. “CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy”. In: *ACM SIGSAC Conference on Computer and Communications Security*. 2016. DOI: 10.1145/2976749.2978363.

[50] David Zeber, Sarah Bird, Camila Oliveira, Walter Rudametkin, Ilana Segall, Fredrik Wollsén, and Martin Lopatka. “The Representativeness of Automated Web Crawls as a Surrogate for Human Browsing”. In: *The Web Conference*. 2020. DOI: 10.1145/3366423.3380104.

[51] Yuchen Zhou and David Evans. “SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities”. In: *USENIX Security Symposium*. 2014. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/zhou.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

Measurement studies are common in web security. Typically, such studies are performed using browser-automation software with fresh browser profiles. In contrast, real usage of the web includes many sessions in which the user is logged into their account. This paper conducts a measurement study comparing the prevalence of web security flaws on $\tilde{2}$00 websites both when logged into an account and when not logged into an account. The paper finds that the ability to accurately generalize scans of the non-authenticated state to the authenticated state depends on the class of security flaw. For example, for XSS, the unauthenticated state will likely underreport the vulnerability. Furthermore, authenticated sessions are often exposed to JavaScript files not seen in unauthenticated sessions. However, such differences were not notable for some other classes of web security flaws tested.

A.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Provides a Valuable Step Forward in an Established Field
- Creates a New Tool to Enable Future Science

A.3. Reasons for Acceptance

1) It has been long expected, but not verified, that only crawling the unauthenticated state in large-scale measurement studies might result in missing important security problems. This paper confirms this hypothesis, and it also quantifies how big of a problem this issue seems to be for different categories of security problems. The lessons can make future web measurement studies more rigorous.
2) The paper creates a tool to help with semi-automatic registration and handling of user credentials and sessions. This will greatly help researchers in performing authenticated web scanning.

A.4. Noteworthy Concerns

1) Only around 200 sites were tested, limiting the generalizability of the conclusions.
2) More critically, the sampling strategy for selecting these sites likely exhibits biases towards very popular sites, those in the English language, and those for which it was easy to create an account. These skews again limit the study's generalizability.
3) Initial registration for the sites tested required a human in the loop, though subsequent logins were automated, again biasing the measurement away from sites whose subsequent logins could not be automated.