Towards Automated Auditing for Account and Session Management Flaws in Single Sign-On Deployments

Mohammad Ghasemisharif, Chris Kanich, and Jason Polakis
University of Illinois at Chicago,{*mghas2, ckanich, polakis*}*@uic.edu*

Abstract*—Single Sign-On (SSO) is both a core and critical component of user authentication and authorization on the modern web, as it is often offered by web and mobile applications along side credential-based authentication to facilitate the account creation and login process. However, the interplay between local account management and SSO functionality in the backend leads to flaws that enable or magnify account hijacking attacks. These flaws are not baked into the actual SSO protocols, but manifest due to the complexity of supporting separate but intermingling authentication paths. As a result, these types of flaws cannot be detected by the SSO protocol or implementation verification tools proposed in prior work. In this paper we introduce SAAT, a fully automated modular framework that assesses whether relying parties (RPs) that use Facebook as the IdP comply with secure practices and guidelines, and uncovers flaws in account and session management that stem from or are affected by the interplay of SSO and local functionality. We conduct a large-scale exploration of authentication and session practices in Facebook's RPs, revealing a volatile ecosystem where SSO support can be suddenly dropped and 17.6% of the tested RPs exhibit non-functional SSO implementations. This highlights the need for the continuous and systematic testing of the SSO ecosystem made possible by SAAT. More critically, we find that security measures are often missing and official guidelines are routinely overlooked or misconfigured, with only 0.8% of the RPs fully enabling re-authentication which can prevent compromise from hijacked identity provider (IdP) cookies. Our study also shows that less than 2% of RPs correctly react to SSO revocation and 67% continue to allow account access even 10 days after revocation. Overall, we envision our framework as a tool for enabling and guiding widespread remediation efforts by major SSO identity providers, which were previously infeasible due to the sheer scale and inherent mutability of this ecosystem.

I. Introduction

Account creation and authentication are essential aspects of the modern web ecosystem. Creating individual accounts for each web service is tedious both for the user who needs to manage multiple passwords, and the service owner who needs to develop and maintain a complex component of their overall system where any flaw can have severe security ramifications. Single Sign-On (SSO) mechanisms offer an attractive alternative that allows users to avoid tedious account creation processes by leveraging their existing accounts on popular services (referred to as *Identity Providers* or *IdPs*). Online services (referred to as *Relying Parties* or *RPs*) can then outsource some or all of their authentication infrastructure to these IdPs, enabling a more integrated and uniform browsing experience across different web services and applications while also streamlining account and session management for both the RPs and the end users.

Although centralizing authentication with major services like Facebook and Google can improve security at relying parties by leveraging their substantial security resources and expertise, SSO introduces a complementary set of security risks to users. While the security of the underlying protocols has been studied in depth [1] and are currently understood to be free of substantial flaws, the implementation thereof may itself be incorrect [2]. Importantly, most uses of SSO augment, rather than fully replace, a site's native authentication mechanisms and there is substantial flexibility in the integration of the RP's and the IdP's authentication mechanisms. This flexibility has led to a variety of specific implementations, which unsurprisingly leads to both a challenge for the RPs to implement said schemes correctly (leading to various vulnerabilities [3], [4], [5]), and difficulty in longitudinally evaluating the security of these implementations [6], [7].

Although many of the aforementioned vulnerabilities are predicated on being successfully authenticated to the IdP, the security of these services can still create a well-fortified but imperfect single point of failure in online authentication. These major services are still not impervious to flaws that enable account hijacking, as shown by prior research [6], [8]. A recent real-world attack campaign resulted in the largest hack in Facebook's history [9], where the authentication tokens (i.e., cookies) of 50 million users were stolen. Alarmingly, as had been previously demonstrated by Ghasemisharif et al. [6], a compromised IdP account allows attackers to obtain persistent and stealthy long-term access to users' RP accounts with little to no option for remediation [6]. In essence, the issues they identified can be traced back to the complexities that arise from the co-existence and interplay of two separate account authentication pathways, that of traditional credentials and that of Single Sign-On, and the ensuing session management processes.

While numerous studies have conducted extensive evaluations of the design and implementations of SSO protocols, no prior work has conducted a systematic, large-scale and in-depth exploration of account and session management in the SSO ecosystem. We introduce SAAT, an automated black-box framework for auditing systems, which use Facebook as the IdP, in the wild. First we compile a set of best-practice guidelines and recommendations for core building blocks of the SSO ecosystem: integrated registration, authentication, and session management. Next, we define a series of auditing tasks, modelled as finite-state machines, that identify violations and insecure practices in the implementation of these processes in Relying Parties. This builds upon our ability to orchestrate actions and infer state changes in the Identity Provider and Relying Parties. More importantly, our fully automated testing pipeline handles every aspect of the SSO protocol; from detection of SSO support and account registration to access revocation and session termination.

We use SAAT to obtain a large-scale longitudinal view of SSO support, uncovering a brittle and volatile ecosystem, with 17.6% of the RPs we tested having non-functional SSO implementations and almost 8% suddenly dropping support for SSO within a 50-day

period. We also identify a significant lack of security mechanisms being deployed, with only 0.8% of the RPs adequately protecting users accounts from IdP cookie hijackers. Moreover, only 1.7% of the RPs log users out in response to IdP access revocation and 67% continue to allow access to the account even 10 days past the revocation. Our auditing also reveals that 10% of the tested RPs violate account merging guidelines. Overall our findings highlight that SSO-deployment insecurity is not limited to implementation flaws in the protocols themselves, but instead should be viewed as an error-prone multicomponent integration process complicated by local account and session management mechanisms. While our study focuses on Facebook as the IdP, we also perform a manual comparative analysis to Google and Apple SSO implementations in a subset of the analyzed RPs. In 99% of the cases we do *not* find any differences across IdPs, indicating that the flaws detected by our system are not limited to a given IdP implementation but instead are intrinsic and affect multiple IdPs supported by vulnerable RPs.

It is important to note that some of the flaws that we identify in our study only become operational after a user's IdP account is compromised. Unfortunately, this may result in a lack of incentive for RPs to fix these flaws as blame can be shifted to the IdP. As such, IdPs can leverage our framework to identify RPs that do not conform to best practices or violate security guidelines, and enforce stricter onboarding requirements or mandate the use of official SDKs. In summary, our research contributions are the following:

- We provide a modeling of SSO account and session management, which guide our development of an automated black-box auditing framework for testing RPs that support Facebook as the IdP.
- We conduct a large-scale study of SSO deployments in the wild, uncovering the prevalence of flaws, non-compliance, and insecure practices across the SSO ecosystem.
- We have disclosed our findings to affected vendors, and also developed a Chrome extension for providing users with additional information about guideline-compliance in RPs. To foster additional research, we will share our code and data with researchers and IdP vendors upon request.

II. Background and Motivation

We provide pertinent background information on SSO, and an overview of policies and best practice guidelines for account and session management when SSO is supported. We also highlight the flaws that are the focus of our study, and end with our threat model.

A. Preliminaries

Identity Providers (IdPs) are entities that provide an authentication service to other entities. We use "SSO support" when a third-party entity allows their users to authenticate via an IdP. We focus on Facebook due to it's prevalence as an IdP [6], as that will allow us to gain a broader view of how RPs incorporate SSO in their websites.

Relying Parties (RPs) are third-party services that delegate their authentication process to an Identity Provider. RPs can be websites or IdP-side applications that other websites use.

Account Access. After the SSO login process, the RP receives *access tokens* that have certain permission scopes. The RP can use these tokens to talk to the IdP (within the permission scope) on behalf of the user. Moreover, the user receives RP cookies which they can use to communicate with the RP directly. It is important to note that access tokens obtained from web logins (as opposed to mobile app logins) are typically short-lived [10] whereas the RP cookies' validity can remain valid way beyond the lifetime of the access tokens. Additionally, the access tokens are either accepted or rejected by the IdP, while RP cookies can provide different levels of access. For instance, an RP may allow accessing non-sensitive parts with cookies while the sensitive parts may require re-authentication via password. In this paper, we do not differentiate between these levels of access and we strictly define RP access based on whether the presence of RP cookies in a request changes the RP's state.

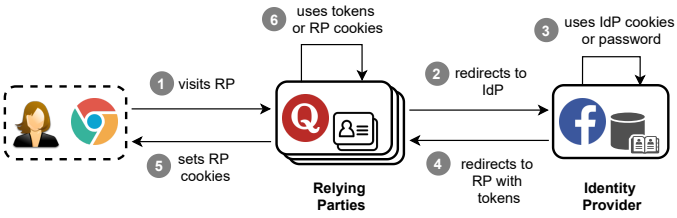

Fig. 1: Single Sign-On workflow.

B. Single Sign-On Workflow

Figure 1 shows a typical SSO authentication process: (1) upon visiting the RP's website and initiating the SSO process, (2) the user's browser gets redirected to the IdP's website. Depending on the RP's configuration, (3) the IdP will attempt to authenticate the user via cookies or ask them to re-authenticate using their password. Once authentication is complete, (4) the IdP redirects the browser back to the RP while appending a code. The RP can then optionally (5) send cookies to the user that will get stored in the browser's cookie jar. Once cookies are set, the RP can (6) continue authenticating the user in the future via those cookies, until they expire. The RP can also use the code to retrieve the necessary tokens for communicating with the IdP on behalf of the user, until those also expire. In this study, we use three primary terms for the SSO workflow and the corresponding test cases that are part of our auditing process.

Authentication paths or channels. RPs may offer users one or more authentication options, namely SSO-based and traditional credential-based authentication. An authentication path is created when one of those options is used to register or log into an account.

Account merging is required for different authentication paths to lead to a single account. For instance, this will occur if a user initially creates an RP account using credentials and later uses SSO to log into the RP. If the SSO process ends in the same account created via credentials, an account merge has occurred, which is the expected behavior per Facebook's guidelines on using SSO with existing login systems [11]. While the lack of account merging may appear as a functionality issue, creating two separate accounts can also lead to more serious security problems such as ransom-type account hijacking [6] where attackers take control of victims' data due to misconfigurations in how the accounts are keyed internally by RPs.

Revocation is offered by IdPs and allows users to request the revocation of the access tokens issued to an RP. While the RP will not be able to use the revoked access tokens to communicate with the IdP, the RP can still continue authenticating the user using the cookies that were set during the SSO workflow (step 5 in Figure 1).

C. Account and Session Management

We consider several policies as part of our RP auditing process for testing RPs' compliance with account management and

authentication systems. The account management guidelines that we incorporate in our assessments originate from IdPs (e.g., Facebook) and outline best practices on how RPs should handle various scenarios, e.g, for conflict-resolution when multiple authentication pathways exist. We also select a set of authentication-related recommendations from the security community (i.e., OWASP) regarding session management in websites that support authentication. Here, we provide an overview of these recommendations.

Offering multiple authentication pathways may cause conflicts which can generally occur in two different ways: (1) a user first creates an account with an RP using a username and password and later decides to login over SSO or link their IdP account to the RP account to be able to perform IdP-related actions, and (2) a user first logs in with an IdP and later decides to add a username and password to the RP. In the first scenario, Facebook's guidelines suggest merging the account information based on the email address being the same, and adding the IdP account information in a separate database table [11]. When the email addresses are different (e.g., the RP account was created using `foo@example.com` and the IdP account with `bar@example.com`) Facebook suggests offering the user an explicit *account merging* option. In the second scenario, Facebook suggests verifying the supplied email before requesting the user to set a password. We note that both resolutions point to an explicit merging policy: *assuming an account belongs to the same user, taking different authentication paths should not result in separate accounts.*

Among the several recommendations and mandates provided by OWASP for securing session management [12], we focus on session expiration as it is directly applicable to our study's focus, i.e, the additional complexities introduced by SSO support. OWASP categorizes automatic session expiration into *idle* and *absolute* timeouts. The idle timeout refers to the amount of time that a session remains active for, when there is no activity. However, since our threat model focuses on session hijacking, this does not limit the attacker's access to a user account as they can thwart such a defense by keeping the session active. For the absolute timeout, while OWASP does not specify an exact time range as it depends on the web app's functionality, they make it clear that the server side's expiration time must precede the client side's (cookie) expiration. In addition to the automatic session expiration, OWASP also advocates for offering a manual session expiration option via the logout button. While there is no specific time defined for cookie expiration, we argue that the cookie expiration in RPs should at least resemble token expiration in IdPs. Facebook's short-lived tokens have a lifetime of an hour and long-lived tokens of 60 days [10]. As such, we consider three primary policies for evaluating session management:

1) Cookies that allow access to RP accounts for more than 60 days after the initial login are non-compliant.
2) Session expiration must be done server-side. Assuming that the server-side timer is in sync with the client-side's cookie expiration date, prolonging the client-side cookie expiration should not affect the server-side timer.
3) If a logout button is offered, it must invalidate session cookies and prevent access unless re-authentication occurs.

Threat Model. We focus on RPs that offer account registration through SSO as well as via local credentials. Our main objective is to design a tool that audits RPs regarding their adoption and correctness of defensive account and session management mechanisms, as well as policies for mitigating the impact and coverage of an IdP-account compromise, such as offering short-term sessions, frequent access-token validation, and re-authentication enforcement. For our analysis, our main assumption is that the attacker has compromised IdP accounts at the session level and subsequently targets the users' RP accounts. This encompasses several attack vectors [13], [14], [12] of varying complexity and scalability, all of which are captured by our threat model because our attack is agnostic to the method through which the IdP session was compromised. For instance, prior academic work on cookie hijacking has demonstrated that incomplete cookie protections are common across popular websites [8], [15], [16]. While various cookie-hijacking protections exist, such as encrypting traffic and using `Secure` and `httpOnly` flags, a recent large-scale study on almost 25K websites found that 50% do not sufficiently protect authentication cookies [17]. Cookie hijacking attacks that enable complete IdP account takeover may require attackers to sniff the users' mobile traffic [6] (which is inherently less scalable), or can be the result of software bugs that allow large-scale cookie harvesting (e.g., as was the case for the stolen Facebook cookies of 50 million users [9]). More recently, Google has detected a surge in malware-driven pass-the-cookie attacks, which enabled the compromise of high-value YouTube accounts; in fact, Google's Threat Analysis Group [18] reported the "resurgence as a top security risk" of these attacks, signifying the prevalence and threat of cookie-stealing attacks. However, it is important to note that session-hijacking attacks are typically more complex and smaller scale compared to more widespread account compromising attacks (e.g., stealing credentials through phishing). Nonetheless, while we focus on session-hijacking attacks, a subset of our experimental findings (e.g., account merging errors, cookie expiration issues, access revocation and session invalidation flaws) also apply to phishing attackers.

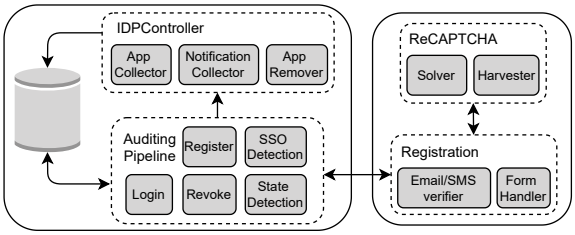

Fig. 2: Components of our SAAT framework.

III. System Overview

Here we present an overview of our framework for auditing RPs. Our framework consists of two main components; an account creation engine and an auditing pipeline, as shown in Figure 2. While our implementation focuses on Facebook as the IdP, we manually verify our approach for Google and Apple in §IV-D.

A. Automated Account Registration

The automated account creation component of our framework is built upon Puppeteer [19] for orchestrating our browser automation.

Registration pages. We combine two strategies for finding login pages. First, we search for the RP's registration page using multiple search engines (`startpage`, `bing`, `duckduckgo`) and select the top three results based on a majority vote. Second, we test common paths (i.e., `/register`, `/signup`, `/login`, `/signin`, `/account`) that were not in the search results. Finally, we filter out unreachable pages (e.g., that return a `404` status).

Registration forms. Our crawler visits the registration pages and locates the sign up section by identifying all `<form>` elements looking for sign up forms using keyword matching. If our

system finds a form element but fails to match any keywords, it selects that form element if and only if that is the only form element in the page and it does not have any search or login related keywords. While this approach may not detect form-less sign up pages as opposed to searching for all relevant inputs, we opt for this approach as it minimizes false positives for pages where the login and sign up sections are included on one page. If the crawler fails to detect forms, it looks for potential links that point to an account creation page and follows them. This is particularly useful in websites whose login and registration URLs are indistinguishable, and single-page applications where sign up forms appear upon interaction. After locating the forms, we select all non-hidden `<input>` fields within the forms and fill them out with random information and check all checkboxes. This initial process allows us to detect dynamically created input elements that will only appear depending on whether other inputs have been filled (e.g., a password confirmation field). We then record all visible inputs, identify the type of personal information required and add them in the corresponding input fields. While we apply a similar method to `<select>` and `<input type="radio">` fields, if we cannot identify the type of information needed we select an option randomly. For each type of input, we have a set of possible values in case some of them are not accepted by the website. We also take the `pattern` attribute into account and select the values that match the pattern. For instance, for password inputs, if our password does not match the pattern, we generate a new one using RandExp [20]. Finally, we check for potential invalid inputs by searching for `aria-invalid` attributes, the `:invalid` CSS pseudo-class, and "error" keywords in the input elements; we try variations of the information until one is accepted. To avoid getting stuck at this phase of our workflow, we need to set an appropriate threshold for this process: we first check for the invalid inputs and if none are found we try submitting the form; if it fails, we check for invalid inputs again. This approach covers both cases of invalid inputs that appear immediately after typing and ones that only get flagged during the submission. We limit the number of trials and the navigation to two and three attempts respectively. The navigation threshold allows us to also handle multi page/step registration forms.

Email/SMS activation. Some websites require activation via email or SMS after submitting the forms, to complete account registration. We use Gmail's API [21] for retrieving the latest emails and filter them based on the website's domain and the existence of verification-related keywords. We also look for verification/activation keywords and numbers that can be used as activation codes. To support SMS verification, we use Twilio's [22] SMS API and follow the same code extraction process for locating potential verification codes in SMS texts. We listen for incoming emails and SMS messages for 15 seconds after the form is submitted. Once we receive a code, we submit it to complete the registration. If neither an email or an SMS is received, we assume that the website does not require additional verification and the account has been successfully created.

State changes and detection. Our account creation and auditing flows primarily rely on correctly distinguishing between an account's *logged in* and *logged out* states. For our auditing process, we consider an event as a state-changing transition if sending two equal HTTP requests, where only one of them carries authentication cookies, results in two different responses. Since web pages can contain dynamic content (e.g., advertisements), we use unique identifiers that belong to the user for detecting differences in the responses. Additionally, while many web pages could be used for detecting state, we have found that visiting the login page is a reliable indicator for detecting states. Every time we need to determine a page's state, our system visits the login page in a separate tab (all storage is shared between the two pages) and checks whether the login page contains indicators such as a login form or unique identifiable information that points to the user.

CAPTCHAs. Websites often rely on CAPTCHA challenges as a means of preventing automated account creation [17]. We draw inspiration from prior studies [23], [24] and use Wit.ai's [25] speech-to-text API to implement a solver for the audio challenges presented by Google ReCAPTCHAs v2. Our solver resides on a remote web server for bypassing rate limiting restrictions by funnelling requests through multiple proxies and different user agents. Before submitting filled-out registration forms, we look for instances of ReCAPTCHA v2; if it exists we extract its `site-key` and send it to our solver. The solver then completes an audio challenge and sends the corresponding token back to the crawler. The token is then submitted along with the registration form. In our initial implementation, we adopted common anti-bot-detection practices such as overwriting `navigator.webdriver` or spoofing `navigator.plugins` [26]. However, due to the cat-and-mouse nature of these evasion techniques, eventually we resorted to using the third-party package `puppeteer-extra-plugin-stealth` [27], which frequently gets updated with the latest evasion techniques. While this could potentially violate RPs' terms of service, automation is a widely established common practice in web security research, and incorporating anti-bot-detection features is becoming increasingly necessary for realistic experimentation [28]. During our experiments, we also noticed that even then some websites were able to detect automation and displayed blank pages. We traced the problem back to code executed from `doubleclick.com`. As such, we included a rule in our main crawler for blocking `doubleclick.com` requests.

B. Single Sign-On Workflow

Single Sign-On detection. We leverage the browser's Web Accessibility API for identifying SSO support. The main goal of this API is to expose an interface that can be used for assistive technologies, as it exposes a semantic version of the user interface and facilitates conveying important information across different platforms, particularly for users with impairments. It is also often used in automated testing and for UI automation in applications like password managers [29]. We provide a code sample of a page with HTML tags and its corresponding accessibility tree in Listings 1, 2 (Appendix). We use Chromium's Accessibility API, which returns a web page's representation as a tree of objects, and traverse the accessibility tree to look for nodes that contain SSO-related information. Since Puppeteer's accessibility tree does not directly expose DOM nodes, we modify the Accessibility class to expose each node's unique identifier (`BackendNodeId`), which we use for resolving the node that contains SSO information.

After detecting SSO support, we inject the IdP cookies into the page and proceed with initiating the login process for the RP. We then collect the following information about the deployment of SSO. First, we log if the IdP requires the user to enter their credentials and re-authenticate despite the presence of the session cookies. This only occurs if the relying party explicitly asks the IdP to re-authenticate users; this can be done through an optional parameter in the SSO workflow. Second, we log if the IdP asks the user for their permission. For instance, Facebook displays a "Continue as" button and upon clicking, the authentication process succeeds and the browser gets redirected back to the RP website. In addition

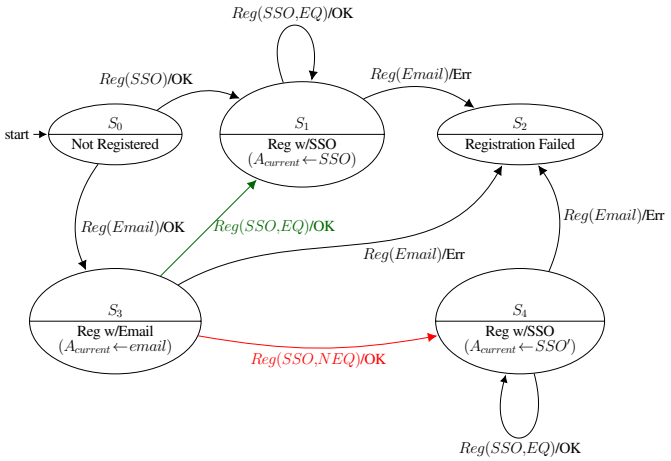

Fig. 3: State machine model of RP account registration.

to the authentication method, we also collect the relying party's cookies, and its unique `app_id` which is assigned by the IdP.

IDPController. A critical dimension of our auditing workflow is interacting with the IdP and observing the impact of IdP actions on the RPs. The controller requires programmatic ability to authenticate with the IdP, access to a list of logged-in RPs, and revoke RP access. These functionalities are ubiquitously supported by popular IdPs, and are handled by our Login and IDPController modules. While we focus on Facebook, these actions can be generalized to other IdPs by modifying the aforementioned modules. Specifically, the loginIDP function should be modified to support the new IdP's authentication flow and the functions in IDPController should be tailored to the specifics of the new IdP for obtaining a list of RPs and removing RPs from the IdP. For instance, Facebook does not currently expose a public API for interacting with RP apps. Instead of interacting with Facebook through an orchestrated browser, we have reverse engineered the communication between client and server and extracted the required data for successfully querying Facebook servers via direct HTTP POST requests. This allows us to speed up the auditing process and is less dependent on Facebook's UI and any changes that would require a modification of our automation actions. If Facebook changes its behavior or a new IdP has a different implementation, we can obtain the data via UI interaction. Facebook assigns RPs to three categories: `active`, `removed`, and `inactive`. Using our approach, we can collect the apps in all categories and also remove them from the active tab. Among the app-related information, we obtain the `install_time`, `inactivation_time`, `removed_time`, `app_user_id`, `permissions`, and `deletion_url` attributes.

C. Auditing Workflow

To enable our collection of auditing tasks and testing procedures, we model our framework's actions and the ensuing state changes as finite-state machines, which allows us to identify non-compliance and violations of security guidelines and best practices.

Authentication paths and account merging. Our testing flow identifies the different registration paths supported by the RP and explores whether taking each path ends in a similar state in the same account. Specifically, our system assesses whether signing up with SSO and creating an account using credentials (i.e., username and password) gets linked to the same account. This workflow verifies whether the RP correctly merges accounts. Figure 3 depicts how we

TABLE I: Mapping the combinations of input symbol and guards to abstract input symbols; m is the registration method, $A_{current}$ points to the registered account in the current state, and A_m is the account created. We use EQ/NEQ to represent the equality/inequality of accounts (i.e., if they are merged). For instance, $Reg(SSO,EQ)$ represents a move where registering an account using SSO results in an account equal to the current state's account ($A_{current}$).

Input	Guards	Abstract Symbols
$Reg(m)$	$m = Email$	$Reg(Email)$
	$m = SSO \wedge A_{current} = undefined$	$Reg(SSO)$
	$m = SSO \wedge A_{current} = A_m$	$Reg(SSO,EQ)$
	$m = SSO \wedge A_{current} \neq A_m$	$Reg(SSO,NEQ)$

model and formalize the relying party's account registration behavior. This test is motivated by the expectation that account registration in a relying party that supports separate authentication paths must result in creating a single account per user (i.e., email address) regardless of the path taken by the user. We now define a set of actions and states we incorporate into our modelling of the registration process.

Registration Model. We use a Mealy machine to model the registration behavior of a relying party. A Mealy machine is a finite-state machine where the current state and current inputs determine the output and the next state. This model can effectively represent the registration process in a relying party since the action taken by our system (mimicking a user action) as well as the current state dictate the subsequent registration state. A Mealy machine M is a six-tuple $(S, S_0, \Sigma_I, \Sigma_O, \delta, \lambda)$ where S is a finite set of states, $S_0 \in S$ is the initial state, Σ_I is a finite set of input symbols, Σ_O is a finite set of output symbols, $\delta : S \times \Sigma_I \rightarrow S$ is the transition function, and $\lambda : S \times \Sigma_I \rightarrow \Sigma_O$ is the output function. In our model the starting state is that of a user being *Not Registered*. We define a state-dependent variable $A_{current}$ that is initially undefined and points to the created account. We also create abstract input symbols by using combinations of input symbol Reg and a set of guards (shown in Table I). We define the set of output symbols $\{OK, Err\}$ describing the generated output as we move to another state. Note that in this model we only focus on registration actions and not linking accounts or adding a password after creation, as those actions occur within a created account.

Merging. We say that a relying party R *merges* accounts created via SSO and credentials if (i) R supports another registration option (i.e., email) in addition to SSO, (ii) when an account is already registered over SSO, attempting to create an account through a credential-based method using the email associated with the IdP account will fail, and (iii) if an account was already created using the credential-based method, signing up with SSO will access the same account as if it was created using SSO (the green transition in Figure 3). This definition is compatible with Facebook's guidelines on account merging [11]. We consider the following cases:

1) $\delta(\delta(S_0, Reg(SSO)), Reg(Email)) = S_2$: Register an account by signing up via SSO, and then check whether creating an account using credentials (namely, the IdP email address) will generate an error stating that the account already exits.
2) $\delta(\delta(S_0, Reg(Email)), Reg(SSO,EQ)) = S_1$: Register an RP account using credentials. Then check whether registering over SSO using an IdP account with the same email address will end up navigating to the same RP account.

Note that we classify the transition $\delta(S_3, Reg(SSO,NEQ)) = S_4$

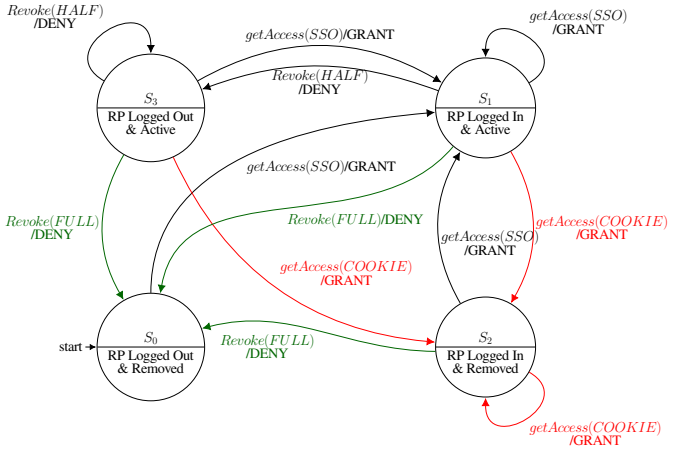

Fig. 4: State machine model of RP's access revocation.

(the red transition in Figure 3) as counter-intuitive behavior. The assumption for case (1) is that accounts registered via SSO are keyed with the IdP email address, which is also backed by our observation that accounts created through the SSO process cannot be re-registered using the IdP's email address. While case (1) is detectable using relatively simple heuristics, case (2) is more challenging. We consider the following strategy for determining account similarity ($A_{current} = A_m$): we use different unique identifiers for account registration and SSO sign up, and check whether we observe the same identifiers in both pages.

Credential test. This procedure investigates whether the RP or IdP allow logins using hijacked cookies. The process begins with visiting the authentication page and verifying that the RP supports SSO. We then attempt to log into the RP using the IdP cookies to complete the SSO process. During the IdP authentication, if the IdP does not ask for credentials, we mark the RP as one that does not explicitly ask for re-authentication. This procedure also examines the validity of RP cookies that were collected in §IV-A. Moreover, we also examine whether explicitly asking for re-authentication using a username and password can be disabled from the client side; this would allow an attacker with hijacked IdP cookies to bypass this security check that requires knowledge of the IdP credentials (and also avoid additional security checks that typically occur at login [17]).

Revocation test. This test explores the efficacy and effectiveness of SSO access-revocation. Specifically, we assess whether revoking access from the IdP affects the user's access to an already-connected relying party. Again, we model the revocation process using a Mealy machine. We define a set of input symbols $getAccess$ and $Revoke$ with two arguments RP_{cookie} and RP_{id}, which represent the relying party's cookies and its unique identification number that is assigned by the IdP. The combinations of these input symbols with a set of guards (shown in Table II) will determine the next state. We also define a set of output symbols{*GRANT, DENY*} representing the output of the taken action as we transition into another state. Figure 4 depicts the revocation process in relying parties. The $Revoke(FULL)$ (green) transition allows a user to completely and permanently revoke access to the RP account, whereas the $getAccess(COOKIE)$ (red) illustrates incomplete revocation where the IdP revokes the RP's access but the RP's authentication cookies remain valid.

The testing workflow starts from state S_0 and logs into an RP using SSO. This creates a transition from S_0 to S_1 ($\delta(S_0,getAccess(SSO)) = S_1$). To test if the RP correctly responds to access revocation, the workflow tests RP's compliance in the logged in (S_1) and logged out (S_3) states. Logging out creates a transition to state S_3 ($\delta(S_1,Revoke(HALF)) = S_3$). When access is revoked, the workflow checks whether the RP's cookies grant access to the RP account. If the RP is not compliant, it will create transitions to state S_2 ($\delta(s, getAccess(COOKIE)) = S_2$ for $s \in \{S_1,S_3\}$), otherwise it will move to state S_0. The state detection method (§III) identifies the state after each transition. Finally, the workflow continues testing RP's compliance overtime while in state S_2, and if it fails it will create a transition from S_2 to itself ($\delta(S_2,getAccess(COOKIE)) = S_2$).

TABLE II: Mapping the combinations of input symbols and guards to abstract input symbols. We use RP_{id} to represent the unique identifier assigned to the RP by the IdP and $Active$ as a set of unique identifiers whose access has not been revoked yet by the IdP. $hasAccess$ returns true or false depending on whether the previously collected RP_{cookie} provides access to the RP. For instance, the abstract input symbol $getAccess(COOKIE)$ represents a move in which the RP's cookies grant access to the account even though RP's access was revoked by the IdP.

Input	Guards	Abstract Symbols
$getAccess(RP_{id},RP_{cookie})$	$RP_{id} \in Active \wedge hasAccess(RP_{cookie})$	$getAccess(SSO)$
	$RP_{id} \notin Active \wedge hasAccess(RP_{cookie})$	$getAccess(COOKIE)$
$Revoke(RP_{id},RP_{cookie})$	$RP_{id} \in Active \wedge \neg hasAccess(RP_{cookie})$	$Revoke(HALF)$
	$RP_{id} \notin Active \wedge \neg hasAccess(RP_{cookie})$	$Revoke(FULL)$

Algorithm 1 Merge test

```
1:  procedure MERGETEST(u,τ,A)
2:      result := undefined
3:      let {A_sso,B_info} ∈ A
4:      let {A_email,A_info} ∈ A_sso
5:      if τ = type1 then
6:          loginWithSSO(u, A_sso); register
7:          result := HasErr(createAccount(u,{A_email,A_info}))
8:      else if τ = type2 then
9:          createAccount(u, {A_email, B_info})
10:         page1 := loginWithCredentials(u, A_email)
11:         (β_id,β_state) := IsLoggedIn(u, page1, {A_info,B_info})
12:         page2 := loginWithSSO(u, A_sso); register
13:         (α_id,α_state) := IsLoggedIn(u, page2, A_info)
14:         if β_state ∧ α_state then
15:             result := IsEqual(α_id,β_id); unique identifiers
16:         end if
17:     end if
18:     return result
19: end procedure
```

D. From Theory to Practice: Auditing Process Implementation

Here we provide an overview of how we implement the auditing workflow in our framework, based on the modeling detailed previously, and clarify how inputs and outputs are mapped to SAAT's components and how it performs the compliance tests.

Merge test workflow. Given the account merge definition in §III-C, we implement the merge test process using SAAT's *Register*, *Login*, and *Detection* modules. Each module contains functions representing the appropriate abstract symbols (see Table I). For instance, the *Register* module defines `createAccount` representing $Reg(Email)$ which takes RP's registration URL

and a set of email and registration information and creates an account. In an SSO workflow, the registration and login processes are the same even though they may be presented as different, e.g., *Register with SSO* and *Login with SSO* follow the same Single Sign-On procedure. Due to this similarity, we define the `loginWithSSO` function within the *Login* module to represent both $Reg(SSO)$ and $getAccess(SSO)$. We also define the function `loginWithCredentials` which receives a URL and a set of credentials (i.e., email and password) to facilitate the login process for the account that was created with `createAccount`.

Algorithm 1 describes the implementation of the merge test, where the MERGETEST function receives a login page URL u, test type τ, and account information $\mathcal{A}$ as inputs, and returns a boolean representing whether the RP merges an account with two authentication paths. The type input τ accepts two inputs $type1$ and $type2$ representing the two merge cases discussed previously. Account information $\mathcal{A}$ contains a set of information pertaining to the SSO account A_{sso} (i.e., IdP account) including the email address A_{email} and personal information A_{info}. It also contains personal information B_{info}, which is used for the $type2$ account creation as well as being a unique identifier when we compare accounts created using different authentication paths (line 15). We define the function `IsLoggedIn` as part of the *Detection* module that receives a URL (i.e., login URL), a web page and a set of identifiers and returns a tuple with the detected identifiers and a boolean value representing the web page's state (i.e., logged in). Note that the $type1$ and $type2$ tests are done separately with fresh accounts, such that lines 6 and 12 are the first time the IdP is connected to the RP.

Revocation test workflow. We implement the revocation test workflow using the *Login* and *Detection* modules, where the *Login* module defines the `loginWithSSO` and `logoutFromRP` functions to implement $getAccess(SSO)$ and $Revoke(HALF)$. Algorithm 2 shows the implemented procedure for the revocation test, where the REVOCATIONTEST function receives a login URL u, test type τ, and account information $\mathcal{A}$ and returns a boolean representing the RP's revocation compliance. This process first logs into the RP using the A_{sso} account and collects the RP cookies (line 5). Before proceeding with the test, we examine whether including the cookies in a new page is sufficient for obtaining access to the account (lines 6 and 7). Next, depending on the test type τ, REVOCATIONTEST performs session termination (logout) or access revocation actions. The former is to examine whether the RP correctly invalidates cookies after logging out, whereas the latter tests the RP's response once the IdP revokes the RP's access. In both scenarios, we ultimately test whether after each action, the collected cookies will provide access to the RP account. Lastly, we leverage the *Detection* module's `IsLoggedIn` function to retrieve the login status after adding the RP's cookies to a newly created page (line 13). If the state (α'_{stat}) is true, indicating that session termination and/or revocation actions do not invalidate cookies, the revocation test result will be false (i.e., the RP is not compliant).

Detection. Our detection functionalities, such as the state detection and SSO detections, are implemented in the *Detection* module, which also defines the `locateAuthPage` function that implements the technique for finding login or registration pages (see §III-A). Each test workflow begins with locating the login or registration web pages that support Single Sign-On. Next, each retrieved URL is used in the MERGETEST and REVOCATIONTEST as parameter u (Algorithms 1 & 2) until the test is complete (i.e., error free), at which point the remaining URLs are ignored.

Algorithm 2 Revocation test

```
1:  procedure REVOCATIONTEST(u,τ,A)
2:      result := undefined
3:      let A_sso ∈ A
4:      let {A_email, A_info} ∈ A_sso
5:      A_cookie := collectCookies(loginWithSSO(u, A_sso))
6:      page := AddCookie(A_cookie)
7:      (α_id, α_state) := IsLoggedIn(u, page, A_info); state detection
8:      if α_state then
9:          if τ = logout then
10:             logoutFromRP(u)
11:         else
12:             d := getDomain(u)
13:             App ID := getAppID(d)
14:             removeApp(App ID); revoke permission
15:         end if
16:         page' := AddCookie(A_cookie)
17:         (α'_id, α'_state) := IsLoggedIn(u, page', A_info)
18:         result := ¬α'_state
19:     end if
20:     return result
21: end procedure
```

IV. EXPERIMENTS & RESULTS

In this section we detail our experimental evaluation and findings.

Experimental setup. We use our framework for two main objectives: 1) quantifying SSO support and obtaining insight into the relationship of RPs and IdPs, and 2) performing compliance tests on RPs. Initially, SAAT takes a hostname and a rank number (to create a unique id) as input and finds the potential login URLs which will be used by the various modules and testing workflows. We use SAAT's *SSO Detection* and *Login* modules for quantifying SSO support and provide the results and detailed examination in §IV-A. For auditing workflows and compliance tests, we leverage SAAT's components to independently identify non-compliant RPs. For instance, for the revocation test (Algorithm 2), we use the *Login* module to log into *all RPs* using SSO (line 5) then remove the RPs from Facebook using *Revoke* module (line 11), and finally use *Detection* module (line 14) to identify the current state of the accounts when authentication cookies are present (i.e., logged in vs logged out). Separating each step and running them in parallel allow us to 1) find and repeat incomplete steps due to errors and 2) prevent our system from getting banned for sending too many requests particularly to the IdP in a short time.

Experimental analysis. We note that Single Sign-On is a volatile ecosystem where RPs may drop SSO support or completely change their authentication workflow over short periods of time. Given our extensive set of experiments conducted at a large-scale, and to account for these changes, the results and statistics for each experiment will only include the websites that were available and supported SSO at the time of each given experiment. Finally, we have tuned our process to optimize for precision (i.e., minimize false positives) which may impact recall (i.e., increase false negatives).

Manual verification. In §IV-B and §IV-C, we manually verify the state-detection results described in §III-A to ensure the accuracy of SAAT's state-detection and measure its performance. When we perform an action that can change the state of an account (e.g., log in/out), we take screenshot images of the page before and after taking the state-change action and save them along with the result we receive from our *Detection* module. Then, we manually go through the images and assess the state-detection results.

A. Single Sign-On Support

RPs often change in a short time period, which can render prior data on SSO support stale. Therefore, continuous observation is necessary for building an accurate picture. Here we present our study of SSO support and provide insight into this ever-changing ecosystem.

Methodology. We select the top 100K sites from Majestic [30] and identify their login pages. Majestic ranks websites based on the number of unique IP subnets that refer to the website and has been used in several recent studies [31], [32], [33], [34]. Nonetheless, our approach is agnostic to the top list used. After identifying the login page, we detect support for Facebook SSO and initiate the login process using our injected Facebook cookies. During the login process, we record instances of RPs that do not accept Facebook cookies and explicitly ask for Facebook credentials. We use two separate Facebook accounts with names that are distinguishable from common names and English words. This facilitates differential analysis during the state detection process. We also collect IdP-generated errors that may appear during login. Such errors may occur if the RP is not configured correctly or is currently in development mode. This allows us to filter out *non-functional* RPs that would pollute our measurements. After logging into each RP, we collect the cookies. We repeat this process to ensure that unexpected errors, due to network disruptions, are minimal. One of the practical challenges that we faced during our crawling process was sending too many and frequent requests, which trigger Facebook's bot detection system. To avoid overwhelming the servers (and potentially getting banned), we limited our crawl to 10k websites per day. However, this created a gap between when we logged into each RP and later performed the revocation tests which could affect the results (e.g., RP cookies expire prior to our revocation test). Therefore, after the second crawl, we repeat the login process for all detected RPs with a 40-60 second sleep time in between. To speed up the process, we use GNU Parallel [35] to run 5-6 processes at a time, thus requiring approximately 1 day to complete. All large-scale data collections were done on an Ubuntu 18.04 server with an Intel(R) Xeon(R) Silver 4110 CPU and 32GB RAM, and manual inspections were done on a personal computer.

Results. We identified and initiated the login process for 2,689 websites that supported SSO with Facebook. Of those, 669 either had a *null App ID* or returned an error, and 120 did not complete the login process due to either freezing from SSO misconfigurations or not loading correctly in headless Chrome. In total, we successfully logged into 1,900 websites through Facebook SSO. To further ensure the reliability of our results, we filtered websites based on whether the login URL's domain (via search engines) matched their corresponding Majestic record, which left us with 1,622 websites with matching domains. While this filtering process can also remove legitimate websites such as `shelfari.com` (merged with `goodreads.com`), we believe that it provides a more accurate dataset for our experiments and eliminates false positives.

To become a Facebook RP a site must first create an application in Facebook, where it will be assigned a unique *App ID*. We extract the *App IDs* during the SSO login process (from the `app_id` or `client_id` URL parameters) and match them to the data collected from Facebook's "Apps and Websites" portal. After completing the login process, we collected applications from both Facebook accounts and selected the matching *App IDs* associated with those 1,622 websites. In total, we collected 1,494 unique apps from Facebook's portal. We note that the relationship between *App IDs* and websites can be one-to-one or one-to-many; in the case of one-to-many the app owner must explicitly whitelist those websites otherwise Facebook throws errors. An example is shown in Table III (Appendix) where the "JotFrom Login" *App ID* is shared between multiple websites with some of them not being whitelisted. We found 36 *App IDs* that had one-to-many relationships, with 19 having similar second-level domains (e.g. `yelp.com` and `yelp.ca`) and 17 cases with different second-level domains. For instance, `cancer.gov`, `interiordesign.net` and `submittable.com` use Submittable, an online platform for collecting and reviewing submissions and applications. Figure 9 (Appendix) visualizes the one-to-many relationships between *App IDs* and the websites in our data. Generally, having the same second-level domain can be an indicator that the RP is managed by the same organization as the websites. However, the opposite is not necessarily true. This has three implications: (1) if the entity behind the shared *App ID* gets compromised then all RPs that outsourced their account management will be affected, (2) the entity in charge of the *App ID* can track users between the different RPs, and (3) the RP becomes a "front", obscuring the actual entity users have to trust.

> **Takeaway 1:** The relationship between IdP-side applications and the websites can be one-to-one or one-to-many. Outsourcing account management to a third-party application creates a single point of failure and an environment where users can be tracked across disjoint RPs without their knowledge.

SSO permissions. In addition to the *App IDs*, we collected the permissions requested by RPs from Facebook's portal. Facebook relies on an app review process for applications that request more than the `public_profile` and `email` permissions. As can be seen in Table IV (Appendix), `public_profile` and `email` are the most prevalent permission combinations we observed throughout the apps collected from the `applications` and `business_tools` sections. `Business_tools` apps request a different set of permissions and are used for managing business assets like pages, events, and groups. However, both use SSO as a login method. Figures 10, 11 (Appendix) illustrate the distribution of requested permissions in the `business_tools` and `applications` sections respectively. Our manual investigation of 54 apps that did not request the `email` permission revealed that for the majority of the apps (33) SSO cannot be used for account creation and can only be added to existing accounts. The rest of the apps only need profile info (e.g., to enforce age restriction) or allow for an email address to be added after SSO.

Longitudinal Single Sign-On support. We tracked changes in Single Sign-On support across two rounds of data collection that were 50 days apart. We found that Single Sign-On support was dropped in 119 websites. We also tracked IdP-side apps through Facebook's portal and noted that after 50 days, 41 apps switched the development mode flag (on/off) at least once. Interestingly, we observed that if an app goes into development mode, it disappears from Facebook's portal, thus preventing users from modifying its access permissions. However, while apps in development mode are not shown in the app portal, their access can still be revoked through Facebook's recovery process. This requires the user to go through the recovery process that is different from visiting the applications page and removing the apps, which may not be obvious to average users. Our findings also show that many RPs had non-functional SSO implementations. Out of 669 unsuccessful SSO logins, 407 were due to errors; these were narrowed down to 348 through our aforementioned domain-filtering process. We manually categorized the errors based on the received descriptions and found that the majority

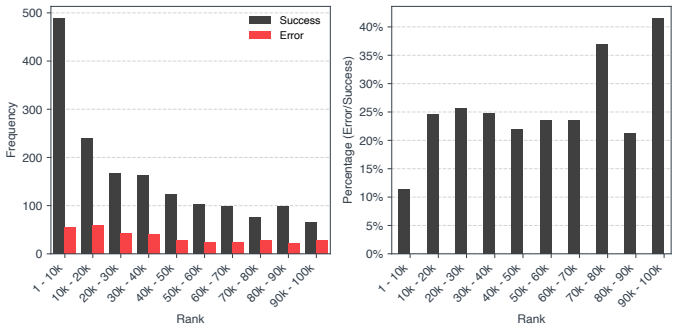

Fig. 5: Number of Relying Party errors and success cases (left) and error to success ratio (right) per website rank.

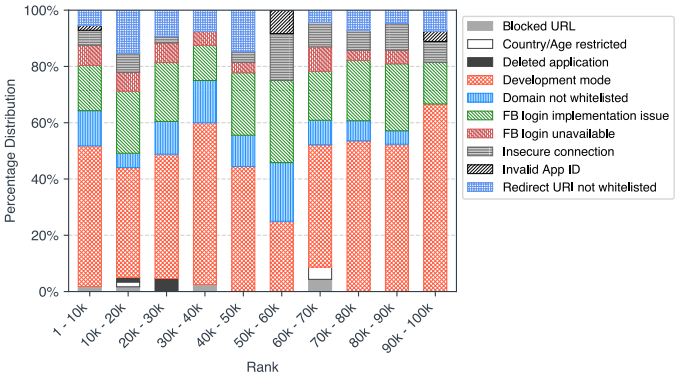

Fig. 6: Error type distribution by website rank.

of errors are caused by apps being in development mode. Figure 5 illustrates the counts of successful and erroneous SSO logins as well as their ratio per website rank. Figure 6 depicts the percentage of error categories correlated with the website rank. As can be observed from the two figures, while the absolute number of erroneous apps is comparable across all bins, the error-to-success ratio is disproportionately lower for the most popular websites (i.e., top 10K).

Takeaway 2: SSO support is dynamic and often changes in a short time window. Such changes can be temporary (development mode) or permanent. The login process for 17.6% of Relying Parties resulted in errors, demonstrating that an accurate assessment of the ecosystem requires interactive measurements.

Re-authentication enforcement. During the Single Sign-On support test, we collected the list of RPs that explicitly ask the Identity Provider to re-authenticate the user even if the user's IdP identity is verified via existing authentication cookies. This additional step is particularly useful for ensuring that having access to the hijacked cookies without knowledge of the password is not sufficient for completing authentication (i.e., it can prevent IdP cookie hijackers from obtaining access to the user's RP accounts [6]). Facebook supports two ways for mandating re-authentication. The first method is performed on the client side, by sending a query string containing `auth_type=reauthenticate` [36]. The second method requires enabling the *Force Web OAuth Reauthentication* option in Facebook's client OAuth settings. We observed a total of 24 RPs requesting re-authentication enforcement. Alarmingly, however, 11 RPs only request re-authentication on the client side; we found that by simply removing the `auth_type` parameter, we were able to bypass re-authentication enforcement and authenticate using the IdP cookies. In other words, only 13 (0.8%) of the 1,622 RPs adequately protect user accounts from IdP cookie hijackers. Interestingly, while Google (the second most prevalent IdP) also follows the RFC and allows the RP to decide [37] [38], Apple follows a more secure approach and always requires reauthentication. Obviously, if the attacker knows the user's IdP password (e.g., through phishing) this mechanism will not prevent the attacker from gaining access.

Facebook SDK. RPs should regularly check the SSO access token's validity, as highlighted by Facebook after their 2018 breach, when they stated that leveraging the official SDK would protect RPs [39]. Facebook also recommends using its JavaScript SDK for protecting against traffic redirection [40]. While not using the SDK is not inherently bad practice, we are interested in understanding how many RPs use it and are considered protected per Facebook's guidelines. To use the SDK, the RP needs to include JavaScript code that loads and initializes the SDK. The SDK's URL is a variation of https://connect.facebook.net/en_US/sdk.js, and once the SDK is fetched it creates a JavaScript object called `FB`. The initialization is completed by passing RP-specific configuration options to `FB.init()`, which is generally wrapped in the `window.fbAsyncInit()` function. Due to the asynchronous nature of this call, we observed that `fbAsyncInit()` is a more accurate proxy for measuring SDK-initialization across RPs. To measure SDK usage, we visited each RP and inspected outgoing requests for the SDK's URL as well as checking for the existence of the `FB` object. We also record calls to the `fbAsyncInit()` and `FB.getLoginStatus()` functions by overriding the functions prior to `page_load`, which makes our approach resilient to obfuscation.

Additionally, `FB.getLoginStatus()` allows the RP to query Facebook whether the user is currently logged into Facebook and whether they have logged into the RP website in the past. Each of these attributes indicates a different level of SDK usage. For instance, websites can include the SDK URL which also creates the `FB` JavaScript object, but never use any functions from the SDK. Our findings show that 49.5% (of 1360) RPs included the SDK in their website, of which 81% initialized the SDK using `fbAsyncInit()` and 4.3% called `FB.getLoginStatus()`. According to Facebook, any SDK functions *must* be called after SDK initialization. In other words, an absence of SDK initialization can be a strong indication that RP does not actually use the SDK's functions. For comparison, we also measured SDK usage for two other popular IdPs, Apple and Google. Both offer official SDKs and provide easy-to-follow documentations for RPs. Our findings show that Google and Apple SDKs have comparable SDK usage, with 45% (of 727) and 40% (217) respectively.

Takeaway 3: Re-authentication enforcement is extremely low among RPs (1.4%), and can be bypassed in 45.8% of the RPs that enable it. Despite best practice guidelines and three years having passed from the Facebook data breach, only 40% of the observed RPs include and properly initialize the official SDK.

B. Access Revocation

Methodology. We investigate the impact of revoking RP permission from within Facebook and explore the functionality of RPs' cookies in the aftermath of access revocation. We visit the RP's login page in two separate flows, with only one of them including the cookies in its requests. Using our state-detection method (see §III-A), we compare the two pages and detect the RP's state via observable side-effects. Note that the state detection's goal is to detect whether

the cookies affect the website's state and not to infer the level of access that the cookies provide. We also take screenshots of the login pages to manually verify the results. During our manual analysis, we inspected all the screenshots and compared the visual differences with what was detected by our system (i.e., personal info, logout buttons, images, etc.) to verify the results of our state-detection method.

We conduct two experiments: (i) an *initial* experiment where we log into RPs, revoke their access and wait for 10 days, and (ii) an *extended* experiment where we log into RPs and wait for a month. The additional time for the control group is to obtain a more extensive cookie-expiration timeline. To prevent being flagged as automated bots, we randomly wait 20-60 seconds between each login attempt, increasing the duration of the entire login process to two days. For the first experiment, we collect the state-change results and take screenshots of the login pages before revocation to ensure that the cookies still work. We continue collecting daily state-change data and the screenshots for the next 10 days. Apart from the revocation step, we follow the same steps for the second experiment and continue to obtain daily screenshots for a month. In both experiments, we collect two sets of state-change data: one with RP cookies *as-is*, and another where we extend the expiration dates to study the impact of client-side and server-side cookie expiration checks. Finally, we investigate whether RPs correctly terminate sessions after logging out. We use the following order of actions for this experiment: first we log into an RP using SSO and collect the RP's cookies. Next, we inject the collected (valid) cookies in a fresh browser and log into the RP again and look for the logout button in the main page and the login pages. The heuristics we use are very similar to our SSO button detection. If the logout is successful, we use the (invalidated) cookies in a fresh browser and visit the RP and invoke our state detection process. If it detects that we are successfully logged in, we flag the RP as non-compliant. Similar to previous experiment, we collect screenshots at each step to later verify our state-detection results.

Results. We consider 1,107 RPs that were successfully processed and found in both the initial and extended groups, and their domain matched the login URL. Initially, we identified 470 RPs where the presence of RP cookies resulted in detectable changes to the RP's state after the SSO login. We manually checked the screenshots and noted that our state detection method had a 3% false positive rate. After the revocation process, only 68 RPs showed a logout behavior. By comparing these RPs with our extended group, we found that in 60 RPs the logout behavior was caused by cookie expiration and only 8 RPs actually exhibited the logged out behavior due to access revocation. We also note that 318 RPs continued accepting cookies 10 days after revocation. As discussed in §IV-A, RPs can use the official SDK to regularly check the access token's validity and, thus, get notified of invalidated tokens. By comparing SDK results, we note that 196 (41%) RPs initialized the SDK (i.e., called `fbAsyncInit()`), but *none of them correctly logged out after access revocation*. Three of the RPs that correctly logged out included the SDK and two of them initialized it. This shows that while RPs that use the SDK are in a better position for getting notified of invalid access tokens, taking the appropriate actions in response to access revocation is still their responsibility, yet is mostly ignored.

Takeaway 4: Only 1.7% of RPs logged out the user in response to IdP access revocation and 67% of the RPs continued to allow access to the accounts even 10 days past the revocation. Use of Facebook's SDK does not correlate with correct logout behavior.

Cookie expiration. We note that short-lived cookies can have a mitigating effect despite a lack of other defensive actions when access revocation occurs (although, this is not a complete and robust solution). Our goal here is to quantify how RP cookies expiration can impact account access over time. We use the collected data from the second experiment to analyze the impact of cookie expiration over a period of 40 days. We identified 1,092 RPs that set cookies after authentication. For each cookie, we calculate the time difference between the expiration timestamp and the login timestamp (which was recorded when we logged in). In cases where the cookies had already expired (e.g., they have negative values or already expired in the past) we replace the expiration date with a value of 0. To better represent the data, we calculate the minimum and median expiration dates of all the cookies for each RP and use the median as the main reference value. Based on our observation, the median value offers a less skewed representation than the average, particularly due to cookies that may expire many years in the future. Since we aim to study the impact of cookies on account access, we only consider the 424 RPs in which cookies made a detectable state change after login completed.

We acknowledge that not all cookies are required for authentication; however, by comparing cookie expiration with the state-change results, we can study the correlation between the time of expiration and account access. To better understand the impact of cookie expiration, we first identify RP's actual behavior using our state-detection method and verify through manual inspection. Our goal is to study whether there is a correlation between cookie expiration values and how RPs handle users' sessions. Next, we separate our dataset into RPs that prematurely rejected cookies that are yet to expire (based on the median value) and RPs whose cookie acceptance or rejection behaviors are aligned with their *median expected expiration*. We identified 127 RPs (57% of the overall RPs that rejected cookies in under 40 days) that prematurely reject cookies. Figure 7 illustrates the CDF for the cookie expiration time in days. The left figure provides a comparison between the *median expected expiration* and actual expiration after which cookies do not provide access to the accounts. The median expiration timestamps are calculated for each RP whereas the actual expiration is measured and verified using our state-detection method over 40 days. The left diagram represents RPs that rejected their cookies before reaching the median expiration timestamp. The right diagram displays the median expected expiration for RPs that accepted cookies before reaching median expiration date, or rejected cookies after reaching their median expiration date. This figure shows that most RPs set cookies that expire instantly or in a short amount of time. The left diagram in Figure 8 shows the actual expiration over 40 days. The RPs that rejected the cookies are noticeably higher two weeks and one month after login. The right diagram in Figure 8 shows a comparison of rejection due to cookies expiration (black) and rejection after revocation (red) for the same RPs; the number of RPs that reject after revocation is higher on the first day and 2 days after. As discussed in §II-C, RPs that accept cookies 60 days after the initial login are non-compliant. Our state-change data shows that 179 RPs accepted cookies even after 70 days. We emphasize that while our cookie expiration examination measured the correlation between cookie expiration and actual expiration to understand how RPs set the expiration date, we use the *actual* expiration data collected from SAAT's state-detection module to identify non-compliant RPs.

Takeaway 5: 48% of RPs accepted cookies 40 days after the initial login and 86% of those RPs were non-compliant and continued to accept cookies even after 70 days.

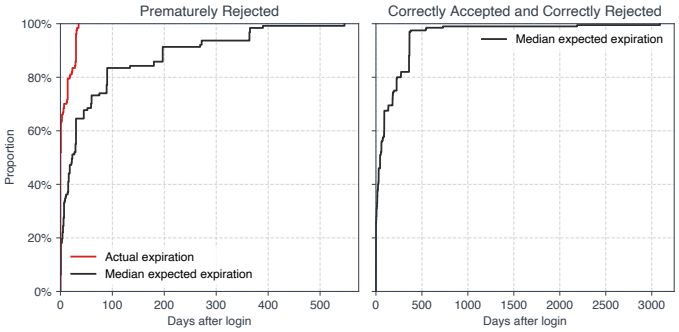

Fig. 7: Actual vs. expected cookie expiration.

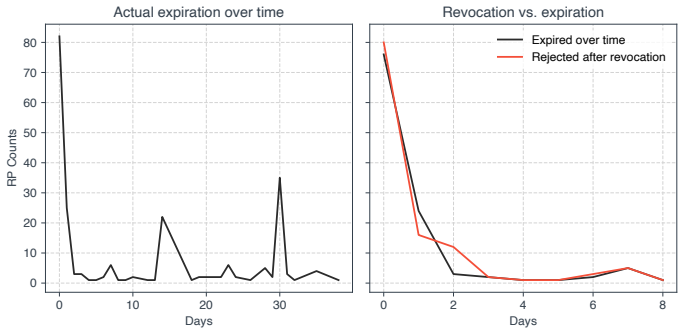

Fig. 8: Actual cookie expiration over 40 days (left). Comparison of rejection due to cookie expiration and after revocation (right).

Server-side vs. client-side expiration. Enforcing the lifetime of cookies falls into two categories: *persistent* and *session* cookies. The lifetime of persistent cookies are specified by the `Expires` or `Max-Age` attributes. Unlike persistent cookies, session cookies don't have explicit expiration dates and are expired and removed at the end of the session. While the definition of a *session's end* may vary between browsers, they treat expired cookies as stale and won't include them in future requests. In this experiment, we investigate whether websites correctly enforce a crucial session expiration rule: regardless of the category, session expiration must be enforced on the server-side [12]. To investigate this, we collect cookies after authentication and send them to the RPs in two scenarios: (i) with their original expiration dates and (ii) with expiration dates set to 1 year in the future. If an RP responds differently to these requests, it means that they rely on the browser to enforce session expiration (client-side). We examined the collected daily screenshots using our state-detection method and manual verification and found 6 RPs including popular websites (`masterclass.com`, `yourstory.com`, `crowdrise.com`), and less popular ones (yet ranked above 20k) like `freeart.com`, `carvana.com`, and `webike.net` that responded differently. All these websites had cookies that had expired before we extended the expiration. Interestingly, apart from `freeart`, the rest of the RPs used a common method of setting the expiration date in the past and hoping that the browser takes care of removing those cookies. We note that the actual expiration time, when the server-side session termination occurs, differs between these RPs. For instance, `masterclass` accepted the modified cookies up to one month after while `webike` and `yourstory` accepted the cookies up to the last day of our data collection (i.e., for 70 days).

Session termination (Logout): We revisit the RPs from our revocation experiment for which the presence of cookies changed their state. Out of 382 correctly processed RPs, we were able to automatically complete the logout process in 138. We proceeded with testing how servers will treat the cookies in subsequent requests. Alarmingly, 40.5% of the RPs allowed us to access the account after logout.

Takeaway 6: Server-side session management and session invalidation are not implemented consistently, as we found 6 RPs that relied on clients' browsers to invalidate cookies, and 40.5% of the RPs did not invalidate cookies after logout.

GDPR Compliance. Under GDPR developers are required to provide a method for users to request the deletion of their data. In Facebook, developers can either provide a link with instructions on how users can do that, or a *Data Deletion Request Callback* [41] that Facebook pings when the users remove the app and request data deletion. Unlike the *De-authorization Callback URL*, the data deletion URL can be collected from the app portal. We found that only 15 Relying Parties have a deletion URL, i.e., if users request data deletion after removing the RPs, only 15 RPs will get notified. Nevertheless, we note that this does not mean the remaining RPs do not accept data deletion requests directly from their website.

C. Account Merging

Methodology. Guided by our merging definition in §III-C, we use two Facebook accounts (A_{SSO}, B_{SSO}) with distinct personal information (A_{info},B_{info}) and emails (A_{email}, B_{email}) to perform the account merging test. Our key assumption here is that RPs use email addresses retrieved from the IdP, or received directly from the sign up forms, to create a local notion of identity. In the first test case, we create accounts with each RP using SSO (A_{SSO}) and later attempt to register an account using A_{email} and B_{info}. If we detect an error message indicating the account already exists, we count this event as a *type-1* merge. In the second case, we first register accounts using A_{info} and B_{email} and later we sign up with SSO (B_{SSO}). We call this a *type-2* merge if the process results in an account with A_{info}. In both cases, our framework detects if a specific RP violates the merging policy upon detecting a pre-existing account, which can occur if the RP creates a separate account or overwrites the account with new personal information.

To prepare for type-1 we logged into the RP accounts using SSO. This is shown as transition $\delta(S_0,Reg(SSO))=S_1$ in Figure 3. Next we initiated type-2 by creating accounts using our automated account registration module, denoted by transition $\delta(S_0,Reg(Email))=S_3$. After logging in with SSO (type-1), we created accounts using credentials and then checked for errors. If the RP displayed an error indicating the account already exists, we added the RP to the correct merging group. For the RPs that we registered using credentials (type-2), we proceeded with SSO login and searched for similar identifiers. If detected $(\delta(\delta(S_0,\ Reg(Email)),\ Reg(SSO,\ EQ))\ =\ S_1)$, we classified the RP as correctly merging the accounts. In contrast, if we detected different identifiers, $(\delta(\delta(S_0,Reg(Email)),Reg(SSO,NEQ))=S_4)$, we flagged it as non-merging. For both cases, we manually verified the results.

Type-1 results. According to our merge definition, RP accounts created using SSO (A_{SSO}) should show an error when there is an attempt to create an account using (A_{email}). We identified 118 RPs that displayed an error indicating an account with (A_{email}) already exists. Among the successfully created accounts (50) we found 4 RPs that did not correctly merge: `surveymonkey`, `diffen`, `connpass`, and `pakwheels` violated the merge policy and allowed

account creation. While not against policy, we note that `92y.org` merged the accounts but overwrote the name. The lack of account merging can have security implications, as the attacker can cut off the user's access to the account until some demand is met (i.e., ransom-style account takeover [6]). We provide an example in the Appendix.

Type-2 results. Using our account registration component, we were able to successfully create accounts in 243 RPs using the traditional credential-based approach. Out of those, 115 have the identifiers that are suitable for the merge test. On the other hand, out of the 1,223 RPs where we created an account over SSO, 354 displayed detectable identifiers and were suitable for this test. We note that not all RPs from the first dataset were also found in the second dataset. For instance, accounts could be registered correctly but when we tried to log in with SSO we encountered app errors (§IV-A), or at the time of this experiment SSO support had been dropped. In total, we had 34 RPs shared between the two datasets. By comparing the identifiers, we identified 30 RPs that correctly merged the accounts and 4 RPs that violated the merge policy: our manual investigation showed that while `diffen` uses email address during the registration it relies on the username, and `pakwheels` does not use the email address when SSO is used. However, in the case of `surveymonkey` and `gifyu`, our experiment resulted in two accounts that share the same email address but have different account information.

Takeaway 7: 10% of the RPs violated the merging guidelines when SSO is used with preexisting accounts, and 11% violated them when the initial account creation was done over SSO.

D. Cross-IdP Generalizability

Next, we explore how the SAAT auditing workflow generalizes across IdPs supported by a given RP and are not tied to the implementation of a specific IdP (i.e., Facebook). To that end, we conduct a series of experiments in a subset of the RPs from our previous experiments that *also* support Google or Apple as IdPs. These IdPs support the universal features requisite for automation using the IDPController module (discussed in §III-B) and are thus amenable to SAAT's auditing workflow. As such, while there are no technical barriers to incorporating additional IdPs into our automated implementation, the additional engineering effort required to develop the appropriate IDPController modules for Google and Apple is outside the scope of our work. As such, we resort to a manual process that *exactly* replicates the steps followed by our system's automated workflow, allowing us to explore the generalizability of our findings across different IdPs.

First, we created a Google and an Apple account with new email addresses ($email_{Apple}$ and $email_{Google}$), to ensure that an experiment with one IdP will not affect another IdP's experiment. We selected a subset of RPs from our previous experiments that also support Google and/or Apple in addition to Facebook, and verified that their SSO procedure is error free (e.g., they are not in development mode). The RPs were selected randomly to eliminate the potential rank bias in the samples. We selected 50 RPs for Google and 50 for Apple, while allowing partial overlap between the two sets so as to obtain additional evidence for certain RPs that their behavior remains consistent across all three IdPs. In total, we conduct 100 sets of comparative experiments across 91 unique RPs. We omit the cookie expiration measurements and session termination (i.e., logout) experiments as they are RP-wide and not tied to an IdP's implementation.

First, we examined whether using Apple or Google as the IdP would produce a different account merging behavior than when using Facebook's SSO implementation. We used two separate browsers, in one browser we logged into each RP using Apple's SSO, and then we attempted to create an account using $email_{Apple}$. We note that while Apple allows users to hide their real email by sending a randomly generated email to the RPs, we did not select this option and opted to share the real email address to mimic the process followed for Facebook. Our findings showed that all of the RPs tested with Apple's SSO produced the same merging behavior (since none of the problematic RPs supported Apple, all 50 RPs we tested had correct merging behavior). Interestingly, we performed the same experiment for Google and found one RP (`pakwheels.com`) exhibiting a different merging behavior: using Google's SSO and creating an account with $email_{Google}$ resulted in correctly creating one account whereas the same process using Facebook's SSO incorrectly resulted in two separate accounts (Section IV-C). When creating an account with Google's SSO `pakwheels.com` sets the user's email address to $email_{Google}$, while with Facebook it does not despite requesting access to the email address. Next, we used the same subset of RPs to compare the effect of access revocation across different IdPs. We began by logging into each RP over SSO and then removed the app (i.e., revoked app's access) from within Apple or Google. We then checked whether revoking access impacted the RP's state and compared to the Facebook results. Unsurprisingly, regardless of the IdP, all results remained consistent. Overall, our comparative analysis showed consistent results when using Google or Apple instead of Facebook in 99% of the cases. This strongly suggests that the flaws uncovered by our system and the overall takeaways of our study are predominantly IdP-agnostic. Nonetheless, we consider larger-scale experimental verification using SAAT an interesting future direction.

V. Countermeasures

Our experiments reveal a series of flaws, misconfigurations, and non-compliance in RPs. In practice, developers can use the official SSO SDKs and also leverage online guides that detail how to correctly implement session and account management processes (e.g., the extensive OWASP cheatsheets [12]). However, RPs may lack the incentives or the technical know-how for addressing these flaws. Thus, we propose two additional strategies for better protecting users.

Transparency report. Our framework can be utilized as a continuous testing framework for generating transparency reports that shed light on RPs' bad practices. We built our framework to be as general as possible to support different SSO implementations. While our main focus was on Facebook, the auditing workflows remain the same for other IdPs. As discussed in previous sections, the SSO ecosystem is highly volatile and any policy checks should be performed over time, therefore proposals that focus on auditing RPs during registration are not sufficient. By leveraging our framework, IdPs can continuously audit RPs and either block problematic RPs or, less intrusively, generate a transparency report that can be used in extensions like the one we describe next, to warn users about RPs that do not adhere to secure account and session management practices.

Browser extension. Complementary to our framework, we have developed an extension that informs users visiting an RP about some malpractices. We provide more details in the Appendix.

VI. Discussion

Automated account registration. Drakonakis et al. [17] implemented and released an automated account registration tool for auditing authentication and authorization flows in web applications. While account creation is only a subset of our system, the high-level

non-SSO registration methods in both systems are quite similar, with a few key differences that we will highlight here. In contrast to their implementation, our system uses Puppeteer to control and automate browser interaction. We chose Puppeteer instead of Selenium due to Puppeteer's improved performance, as well as other key features like the ability to interact with Chrome's DevTools Protocol (CDP), listen on network events and modify requests (Selenium 4 introduced support for CDP API, but it is still in alpha version at the time of this writing). Additionally, Puppeteer provides more control over when and how cookies are loaded/injected and stored, which is a crucial part of our auditing framework. We also leverage Puppeteer's CDP API to obtain corresponding DOM nodes in the accessibility tree which are not typically exposed (discussed in §III-B). Our system also includes CAPTCHA-solving, which was one of the two main causes for failed registrations in [17]. The second main cause for failed registrations was the lack functionality for detecting and fixing input errors during registration (e.g., due to formatting constraints), which we have included in our system. We have also extended the account activation process to support SMS-based activation. For us to have a unified framework and also address these shortcomings, we decided to not directly utilize their tool but instead incorporate their key ideas into our own version of the non-SSO automated account registration using Puppeteer. We note that all of our SSO-related processes were *not* modelled after their design.

Ethics and disclosure. It is important to emphasize that all experiments were conducted using test accounts registered by our framework. During our experiments we *did not* interact with or affect actual users in any way. To facilitate remediation efforts we notified affected RPs following established strategies [42], [43], [44], [17] for identifying contact emails. This included collecting websites' `security.txt` files, leveraging search engines, crawling the websites, and obtaining each domain's `WHOIS` record. While we are still waiting for feedback from other RPs, we received confirmation from `gifyu.com` that their current system does not merge accounts as they do not collect email addresses, but plan to use Hybridauth [45]. We have also shared our work with Facebook.

Limitations. Certain caveats are inherent to any study, such as ours, that relies on a fully automated system and analysis pipeline. This includes the inability to create an account on certain RPs, or potential false positives/negatives during the testing phase. For the former issue, while our system was able to successfully complete the login process on 1,900 RPs, in practice researchers could supplement this by manually creating accounts on problematic websites of interest. For the latter, as aforementioned, when designing our system we opted for correctness (i.e., minimizing false positives) and also manually verified all of our findings to ensure validity.

VII. Related Work

Protocol Verification on the Web. Authentication and authorization using third parties is a complex, critical, and security-sensitive component of the modern web, necessitating the standardization and evaluation of appropriate protocols. OpenID Connect, the standardized protocol used in most implementations of Web SSO, has been studied extensively; see the formal analysis of the protocol in Fett et al. [46] and an overview of the scholarship in this area related to protocol vulnerabilities by Mainka et al. [1]. While these efforts are substantial and necessary for securing the SSO ecosystem, the vulnerabilities that we consider in this paper are beyond the scope of such tools due to the vulnerabilities that arise out of the composition of SSO and non-SSO authentication mechanisms.

Protocol and implementation mismatches. While the analysis of these protocols is a necessary component of ensuring their security, very often the devil is in the implementation details. Researchers have investigated SSO implementations and found various vulnerabilities [47], [48], [4]. Our approach is complementary to investigations of attack models that directly target the SSO implementations themselves,e.g., Sudhodanan et al.'s work evincing various vulnerabilities of Multi-Party Web Applications [49] and Cao et al.'s investigation of relying party impersonation attacks [50]. A necessary precondition for evaluating attacks at scale is the ability to create and interact with valid authenticated sessions; SAAT complements these investigations by providing a framework for evaluating large swaths of the Internet for vulnerabilities. Zhou and Evans [2] built an automated system that handled the SSO-registration process and detected implementation flaws in SSO protocols; while some of the automation techniques have inspired our SSO registration process, their system has not been publicly maintained in the past six years, and thus cannot be readily applied for auditing contemporary web application implementations.

While our paper focuses on SSO-based account creation and session management, Shernan et al. [5] performed a crawl-based investigation of a CSRF vulnerability in OAUTH 2.0 (a precursor to contemporary SSO implementations) which was able to automatically audit sites for potential vulnerabilities, and necessitated manual inspection to identify true vulnerabilities. Recently Liu et al. [7], explored how email reuse attacks can allow an adversary to takeover accounts in SSO RPs. More closely related to our work is that of Ghasemisharif et al. [6], which investigated the interplay between accounts managed by relying parties and the connection to the accounts managed by IdPs, but did so with substantial manual investigation and at a small scale, the likes of which our automated auditing system is designed to streamline and standardize. Furthermore, their study did not explore how revocation, session termination and cookie expiration actually affect RPs over time. Considering the web ecosystem more broadly, researchers have also investigated the security of various protocols and implementations thereof for other web security primitives including Certificate Authorities [51], TLS [52], [53], HSTS [54], and CSP [55]. In many cases, dynamic analysis via crawling-style auditing was able to identify numerous vulnerable implementations of these protocols.

VIII. Conclusion

SSO has revolutionized web authentication by allowing services to essentially outsource the identity verification process to major IdPs While the authentication process in these services is typically well-protected, leading to security benefits for the RPs, the co-existence and interplay of two separate account authentication pathways creates additional security pitfalls. As such, we developed an approach to fully automating black-box auditing framework for detecting violations and non-compliance of secure practices in Facebook's RPs. We implemented this tool for Facebook and manually verified the approach on Apple and Google. Our large-scale analysis revealed a series of flaws, ranging from insecure cookie management practices and a lack of token-liveness checks to incorrect account-merging practices. Overall, our research highlights that adopting SSO is not a panacea against authentication flaws but, instead, a process fraught with multiple nuanced opportunities for mistakes. Apart from our responsible disclosure to the affected RPs, we will also share our framework with researchers and IdPs, as we envision it being used by major IdPs for ensuring a safer SSO ecosystem through continuous testing and reporting of insecure practices.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was supported by the National Science Foundation (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.

REFERENCES

[1] C. Mainka, V. Mladenov, J. Schwenk, and T. Wich, "Sok: single sign-on security-an evaluation of openid connect," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 251–266.

[2] Y. Zhou and D. Evans, "Ssoscan: Automated testing of web applications for single sign-on vulnerabilities," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 495–510.

[3] W. Li and C. J. Mitchell, "Security issues in oauth 2.0 sso implementations," in *International Conference on Information Security*. Springer, 2014, pp. 529–541.

[4] S.-T. Sun and K. Beznosov, "The devil is in the (implementation) details: An empirical analysis of oauth sso systems," in *Proceedings of CCS 2012*.

[5] E. Shernan, H. Carter, D. Tian, P. Traynor, and K. Butler, "More guidelines than rules: Csrf vulnerabilities from noncompliant oauth 2.0 implementations," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 239–260.

[6] M. Ghasemisharif, A. Ramesh, S. Checkoway, C. Kanich, and J. Polakis, "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)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1475–1492. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/ghasemisharif

[7] G. Liu, X. Gao, and H. Wang, "An investigation of identity-account inconsistency in single sign-on," in *Proceedings of the Web Conference*, 2021.

[8] S. Sivakorn, J. Polakis, and A. D. Keromytis, "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*, ser. S&P '16, 2016.

[9] M. Isaac and K. Conger, "The New York Times - Facebook Hack Puts Thousands of Other Sites at Risk," 2018. [Online]. Available: https://www.nytimes.com/2018/10/02/technology/facebook-hack-other-sites.html

[10] Facebook, "Access tokens - facebook login," 2021. [Online]. Available: https://developers.facebook.com/docs/facebook-login/access-tokens/

[11] ——, "Using facebook login with existing login systems," 2021. [Online]. Available: https://developers.facebook.com/docs/facebook-login/multiple-providers/

[12] OWASP, "Session management cheat sheet," 2021. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

[13] C. A. Vlsaggio and L. C. Blasio, "Session management vulnerabilities in today's web," *IEEE Security Privacy*, vol. 8, no. 5, pp. 48–56, 2010.

[14] R. Wang, S. Chen, and X. Wang, "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*. IEEE, 2012, pp. 365–379.

[15] S. Calzavara, A. Rabitti, A. Ragazzo, and M. Bugliesi, "Testing for integrity flaws in web sessions," in *European Symposium on Research in Computer Security*. Springer, 2019, pp. 606–624.

[16] S. Sivakorn, A. D. Keromytis, and J. Polakis, "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*, 2016, pp. 71–81.

[17] K. Drakonakis, S. Ioannidis, and J. Polakis, "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*, 2020, pp. 1953–1970.

[18] A. Shen, "Google Threat Analysis Group - Phishing campaign targets YouTube creators with cookie theft malware," 2021. [Online]. Available: https://blog.google/threat-analysis-group/phishing-campaign-targets-youtube-creators-cookie-theft-malware/

[19] Google, "Puppeteer," 2021. [Online]. Available: https://github.com/puppeteer/puppeteer

[20] Fent, "Randexp," 2021. [Online]. Available: https://github.com/fent/randexp.js

[21] Google, "Gmail api," 2021. [Online]. Available: https://developers.google.com/gmail/api

[22] Twilio, "Communication apis for sms, voice, video and authentication," 2021. [Online]. Available: https://www.twilio.com/

[23] S. Solanki, G. Krishanan, V. Sampath, and J. Polakis, "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*, ser. AISec '17, 2017.

[24] K. Bock, D. Patel, G. Hughey, and D. Levin, "uncaptcha: a low-resource defeat of recaptcha's audio challenge," in *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

[25] Wit.ai, "Build natural language experience," 2021. [Online]. Available: https://wit.ai/

[26] E. Sangaline, "It is not possible to detect and block chrome headless," 2021. [Online]. Available: https://intoli.com/blog/not-possible-to-block-chrome-headless/

[27] Berstend, "puppeteer-extra-plugin-stealth," 2021. [Online]. Available: https://github.com/berstend/puppeteer-extra/tree/master/packages/puppeteer-extra-plugin-stealth

[28] J. Jueckstock, S. Sarker, P. Snyder, A. Beggs, P. Papadopoulos, M. Varvello, B. Livshits, and A. Kapravelos, "Towards realistic and reproducibleweb crawl measurements," ser. WWW '21, 2021.

[29] Chromium, "Accessibility overview," 2021. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/accessibility/overview.md

[30] Majestic, "The majestic million," 2020. [Online]. Available: https://majestic.com/reports/majestic-million

[31] A. Aliyeva and M. Egele, "Oversharing is not caring: How cname cloaking can expose your session cookies," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 123–134.

[32] Y. Nakatsuka, A. Paverd, and G. Tsudik, "Pdot: Private dns-over-tls with tee support," *Digital Threats: Research and Practice*, vol. 2, no. 1, Feb. 2021. [Online]. Available: https://doi.org/10.1145/3431171

[33] F. Quinkert, T. Lauinger, W. Robertson, E. Kirda, and T. Holz, "It's not what it looks like: Measuring attacks and defensive registrations of homograph domains," in *2019 IEEE Conference on Communications and Network Security (CNS)*, 2019, pp. 259–267.

[34] K. Borgolte, C. Kruegel, and G. Vigna, "Meerkat: Detecting website defacements through image-based object recognition," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 595–610.

[35] O. Tange, "Gnu parallel 20200522 ('kraftwerk')," May 2020, GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them. [Online]. Available: https://doi.org/10.5281/zenodo.3841377

[36] Facebook, "Re-authentication - facebook login," 2021. [Online]. Available: https://developers.facebook.com/docs/facebook-login/reauthentication/

[37] N. Sakimura, J. Bradley, M. Jones, B. De Medeiros, and C. Mortimore, "Final: Openid connect core 1.0 incorporating errata set 1," 2021. [Online]. Available: https://openid.net/specs/openid-connect-core-1_0.html

[38] Google, "Openid connect," 2021. [Online]. Available: https://developers.google.com/identity/protocols/oauth2/openid-connect

[39] Facebook, "Facebook login update - about facebook," 2021. [Online]. Available: https://about.fb.com/news/2018/10/facebook-login-update/

[40] ——, "Login security - facebook login," 2021. [Online]. Available: https://developers.facebook.com/docs/facebook-login/security/

[41] ——, "Data deletion callback - app development," 2021. [Online]. Available: https://developers.facebook.com/docs/apps/delete-data/

[42] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, "Complex security policy? a longitudinal analysis of deployed content security policies," in *NDSS*, 2020.

[43] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes, "Hey, you have a problem: On the feasibility of large-scale web vulnerability notification," in *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016.

[44] F. Li, Z. Durumeric, J. Czyz, M. Karami, M. Bailey, D. McCoy, S. Savage, and V. Paxson, "You've got vulnerability: Exploring effective vulnerability

notifications," in *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016.

[45] Hybridauth, "Hybridauth social login php library," 2021. [Online]. Available: https://hybridauth.github.io/

[46] D. Fett, R. Küsters, and G. Schmitz, "The web sso standard openid connect: In-depth formal security analysis and security guidelines," in *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, 2017, pp. 189–202.

[47] R. Yang, G. Li, W. C. Lau, K. Zhang, and P. Hu, "Model-based security testing: An empirical study on oauth 2.0 implementations," in *Proceedings of ASIACCS 2016*. ACM, May 2016, pp. 651–662.

[48] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong, "Authscan: Automatic extraction of web authentication protocols from implementations." in *Proceedings of NDSS 2013*, Feb. 2013.

[49] A. Sudhodanan, A. Armando, R. Carbone, L. Compagna *et al.*, "Attack patterns for black-box security testing of multi-party web applications." in *NDSS*, 2016.

[50] Y. Cao, Y. Shoshitaishvili, K. Borgolte, C. Kruegel, G. Vigna, and Y. Chen, "Protecting Web Single Sign-on against Relying Party Impersonation Attacks through a Bi-directional Secure Channel with Authentication," in *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defense*, ser. RAID. Springer, September 2014.

[51] D. Kumar, Z. Wang, M. Hyder, J. Dickinson, G. Beck, D. Adrian, J. Mason, Z. Durumeric, J. A. Halderman, and M. Bailey, "Tracking certificate misissuance in the wild," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 785–798.

[52] D. Kaloper-Meršinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell, "Not-quite-so-broken {TLS}: Lessons in re-engineering a security protocol specification and implementation," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 223–238.

[53] S. Calzavara, R. Focardi, M. Nemec, A. Rabitti, and M. Squarcina, "Postcards from the post-http world: Amplification of https vulnerabilities in the web ecosystem," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 281–298.

[54] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz, "Measuring https adoption on the web," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1323–1338.

[55] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "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*, 2016, pp. 1376–1387.

APPENDIX

Here we include additional analysis and findings from our large-scale study.

A. Relationship between RPs and IdP

In §IV-A we discussed the subtle distinction between IdP-side applications (i.e., *App IDs*) and RPs; each IdP-side application can belong to a website called an RP, or provide a service to different websites by positioning itself between the IdP and the websites and form a one-to-many relationship. Figure 9 visualizes the one-to-many relationship between the IdP, IdP-side applications and websites in our data. The green nodes are the IdP-side applications that likely belong to the same organization (based on domain names) and red nodes are the IdP-side applications with different domains from the websites. Table III presents an example of a shared *App ID* between seven websites, where three of the websites have errors. We also discussed the negative implication of such one-to-many relationship particularly for the ones that are not managed by the same organizations (red nodes): apart from the obvious privacy implications, the potential misconfigurations in the IdP-side applications can indirectly propagate to the websites and its impact is multiplied by the number of websites connected to them.

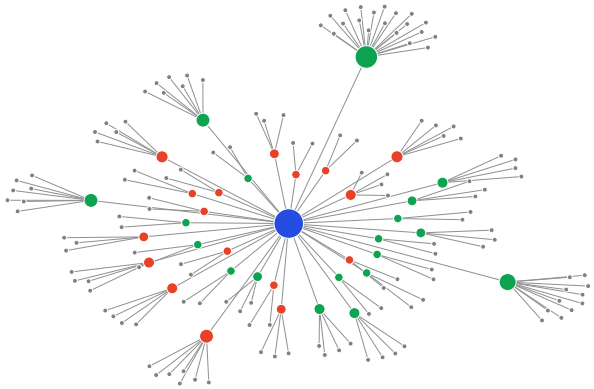

Fig. 9: One-to-many relationship between the *App IDs* (red,green) and websites (grey). Websites with the same *App ID* are connected to an intermediate node which is colored based on whether the connected websites have the same (green) or different (red) second-level domains. The blue node represents the Identity Provider (Facebook).

TABLE III: Example of application that is shared between seven websites and not configured correctly on three of them.

App ID	**Rank**	**Domain**	**Login URL**	**Error**
1140740696088074	883	jotform.com	www.jotform.com/signup/	
1140740696088074	3753	jotformeu.com	jotformeu.com/signup	Redirect URI not whitelisted
1140740696088074	8077	jotform.us	www.jotform.us/signup	
1140740696088074	8328	jotform.me	www.jotform.me/signup	
1140740696088074	11844	jotformpro.com	www.jotformpro.com/signup	Redirect URI not whitelisted
1140740696088074	19929	jotform.co	www.jotform.co/signup	Redirect URI not whitelisted
1140740696088074	53896	jotformz.com	www.jotformz.com/signup	

B. App Permissions

The majority of the collected Facebook applications discussed in §IV-A belong to the `applications` category, and Figure 11 illustrates their SSO permission distribution. A handful of collected applications in our data belonged to the `business_tools` category where the applications can request different permissions for managing pages and groups in addition to typical permissions such as `public_profile`, `email` and `user_posts`. Figure 10 illustrates the permission distributions for apps in the `business_tools` category. We note that `public_profile` and `email` are requested more frequently in both categories, which is intuitive since they provide basic personal information about the users. In contrast, `user_posts` is requested less frequently in both categories, which may indicate that either apps are not interested in interacting with users' posts or requesting such permissions may generate negative feedback from users and decrease the adoption rate. Additional exploration is needed for understanding the underlying cause, since this falls out of the scope of our study.

C. Cookie Rejection Ratio

In §IV-B, we studied the impact of cookie expiration over time in RPs. Figure 12 shows the number of RPs that accepted or rejected cookies per their rank, in the first day after login and 40 days later. As can be observed, both popular (i.e., highly ranked) and unpopular RPs accepted cookies even 40 days after login.

TABLE IV: Permission combination frequencies in applications.

Permission Combinations	**Applications**
public_profile, email	1,325
public_profile	54
public_profile, user_friends, email	29
public_profile, email, user_birthday	9

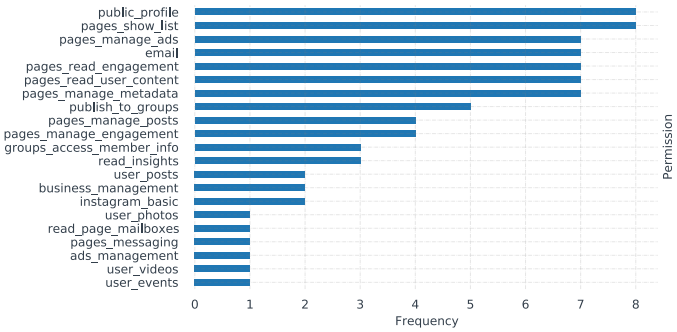

Fig. 10: Permission distribution of apps in Business Tools.

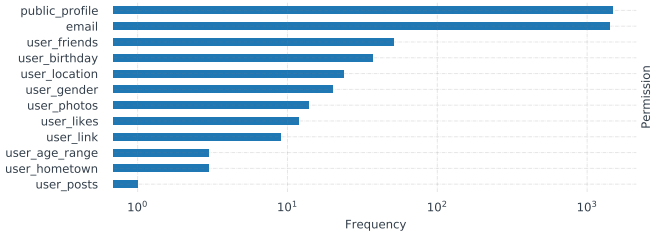

Fig. 11: Permission distribution of apps in Applications.

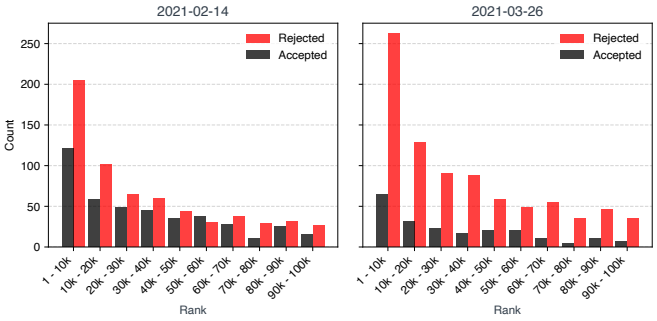

Fig. 12: Counts of accepted/rejected cookies per rank for the first and the last day (40).

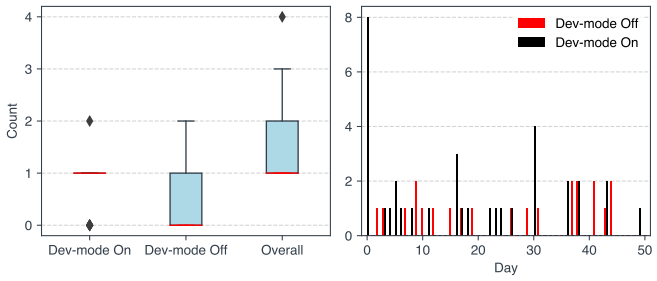

Fig. 13: Dev-mode switched on/off during 50 days.

D. Development Mode

In §IV-A we discussed how enabling the development mode essentially renders an RP inaccessible. By observing and measuring how often the IdP-side apps disabled and enabled the development mode over 50 days, we illustrated the importance of interacting with the RPs for accurately measuring SSO related features. Figure 13 shows how frequently IdP-side apps switched development mode on/off over the span of 50 days.

E. Web Accessibility API

In §III-B, we discussed how we leveraged Chromium's Web Accessibility API to obtain the accessibility tree, which is a tree of objects resembling the HTML elements and used by assistive technologies to facilitate website interactions for users with disabilities. Listing 1 depicts an HTML code segment for a login page, which contains two text boxes for the username and password, a submit button, and Facebook's SSO button. For brevity, we do not include the entire HTML code in this sample. Listing 2 illustrates the accessibility tree obtained from Chromium's Web Accessibility API upon visiting the page. We modified Puppeteer's accessibility API to also include the node information that is not included in the accessibility tree to map each object back to its DOM element. Note that the accessibility tree only contains simplified information, which includes a subset of HTML elements that are deemed to be useful (e.g., it does not include the hidden `input` element). For the purpose of SSO detection, we apply a set of regular expressions on the accessibility tree values to find potential candidates for SSO buttons, and username and password fields related to our test workflows. For instance, in Listing 2, the last node is a candidate for Facebook's SSO button that can initiate the login process.

Listing 1: A sample HTML code for a login page.

```
<form method="get" action="/submit" validate>
   <h3>Login Form</h3>
      <div>
         <label for="username">Email</label>
         <input type="text" name="email" required>
         <label for="password">Password</label>
         <input type="password" name="password" required>
         <input type="hidden" id="test" name="secret"
                value="1234">
      </div>
      <div>
         <button type="submit">Login</button>
      </div>
      <fb:login-button scope="public_profile,email"
          onlogin="checkLoginState();">
      </fb:login-button>
</form>
```

Listing 2: A sample output of accessibility tree obtained from Web Accessibility API

```
{
   "role": "WebArea",
   "name": "",
   "children": [
      {
        "role": "heading",
        "name": "Login Form",
        "level": 3
      },
      {
        "role": "text",
        "name": "Email"
      },
      {
        "role": "textbox",
        "name": "",
        "required": true,
```

```
        "invalid": "true"
      },
      {
        "role": "text",
        "name": "Password"
      },
      {
        "role": "textbox",
        "name": "",
        "required": true,
        "invalid": "true"
      },
      {
        "role": "button",
        "name": "Login"
      },
      {
        "role": "Iframe",
        "name": "fb:login_button Facebook Social Plugin"
      }
    ]
}
```

F. Ransom-style Example

Here, we provide an instance of a ransom-style attack on `connpass`. An attacker can log into `connpass` using the IdP's hijacked session, de-link the user's IdP from `connpass` after adding her own email address and setting a password or linking another IdP (e.g., Twitter). As a result of the de-linking and re-linking, the user will not be able to access their RP account since there is no other viable authentication path for them. Had the RP followed the policy and merged the accounts, the user would have had another path (e.g., over email) to take back control of their account.

G. Countermeasures

Extension for user awareness. As mentioned in §V, we have developed a Chrome extension that informs users visiting an RP about a subset of the issues detected by our system, which can be inferred by visiting the website and without conducting our entire black-box auditing workflow. For instance, if the access tokens of RPs that use Facebook's official SDK are automatically and frequently validated and whether their redirection traffic is protected [10]. By checking for the presence of SDKs, we display a set of preferred IdPs to the user to choose from, based on whether they are using the official SDKs. Our extension uses the `chrome.debugger` API to get a copy of the accessibility tree and looks for potential IdPs. We also pre-load and instrument the well-know IdP SDKs (i.e., Facebook, Apple, and Google) to track whether they are used by the page. Upon detecting the IdPs, we check for the presence of the SDKs, and for Facebook we also check whether it has been initialized. For the compliance tests that can't be done in a live setting (e.g., merge tests), our extension could incorporate IdPs' transparency reports to help users make more informed decisions. While our prototype's functionality is limited due to the inherently complex nature of our framework's auditing process, we hope that our work motivates major IdPs and leads to stricter RP-compliance requirements being enforced.