SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities

Yuchen Zhou David Evans
University of Virginia
[*yuchen*, *evans*]@*virginia.edu*
http://SSOScan.org

Abstract

Correctly integrating third-party services into web applications is challenging, and mistakes can have grave consequences when third-party services are used for security-critical tasks such as authentication and authorization. Developers often misunderstand integration requirements and make critical mistakes when integrating services such as single sign-on APIs. Since traditional programming techniques are hard to apply to programs running inside black-box web servers, we propose to detect vulnerabilities by probing behaviors of the system. This paper describes the design and implementation of SSOScan, an automatic vulnerability checker for applications using Facebook Single Sign-On (SSO) APIs. We used SSOScan to study the twenty thousand top-ranked websites for five SSO vulnerabilities. Of the 1660 sites in our study that employ Facebook SSO, over 20% were found to suffer from at least one serious vulnerability.

1 Introduction

Single Sign-On (SSO) services are increasingly used to implement authentication for modern applications. SSO-enabled applications allow users to log into an application using an established account (with a service such as Facebook or Twitter) and connect their account on the new site to an established Internet identity. Should the application need more information from the user, it may ask the user for extra permissions from the established service. Once granted, the requested information is returned to the application, which can then be used in the transparent account registration process.

Although these services provide SDKs intended to enable developers without security expertise to integrate their services, actually integrating security-critical third-party services correctly can be difficult. Wang et al. identified several ways applications integrating SSO SDKs can be vulnerable to serious attacks even when developers closely follow the documentation [27].

To better understand and mitigate these risks, we developed SSOScan, an automated vulnerability checker for applications using SSO. SSOScan takes a website URL as input, determines if that site uses Facebook SSO, and automatically signs into the site using Facebook test accounts and completes the registration process when necessary. Then, SSOScan simulates several attacks on the site while observing the responses and monitoring network traffic to automatically determine if the application is vulnerable to any of the tested vulnerabilities. We focus only on Facebook SSO in this work, but our approach could be used to check SSO integrations using other identity providers or other protocols. Many of our techniques could also be adapted to scan for vulnerabilities in integrating other security-critical services such as online payments and file sharing APIs.

1.1 Contributions

Our work makes two types of contributions: those related to the construction of our scanning tool which are largely independent of the particular vulnerabilities, and those resulting from our large-scale study of Facebook SSO implementations.

SSOScan. We explain the design and implementation of SSOScan (Section 3), as well as how to handle some of the challenges in the automation process. We describe techniques that automatically perform user interactions to walk through the SSO process (Section 3.1), including clicking the correct buttons and filling in registration forms. We collected information of almost 30,000 click attempts for sites that implement Facebook SSO which shows in detail how the individual heuristics are affecting SSOScan's behavior (Section 5.2). This provides experimental evidence to support our design choices and shed light on future research that shares a similar goal. SSOScan can detect whether a target application contains any of the five vulnerabilities listed in Section 2.2 with an

average testing time of 3.5 minutes, and is able to check 792 (81%) of the 973 websites that implement functional Facebook SSO from the top 10,000 with *no* human intervention at all.

Large-scale study. We ran SSOScan on the top 20,000 US websites (Section 4). Key results from the study include finding at least one vulnerability in 345 of the 1660 sites that use Facebook SSO (Section 4.1). We also learn how vulnerability rates vary due to different ways of integrating Facebook SSO (Section 4.1.1). We manually analyzed the 228 sites ranked in the top 10,000 that SSOScan cannot test automatically and report on the reasons for failures (Section 4.2). Our study reveals the complexity of automatically interacting with web sites that follow a myriad of designs, while suggesting techniques that could improve future automated testing tools. In Section 6, we discuss our experiences reporting the vulnerabilities to site owners and possible ways SSOScan could be deployed.

2 Background

This section provides a brief introduction to single sign-on systems, describes the vulnerabilities we checked, and summarizes relevant previous work.

2.1 Single Sign-On

A typical single sign-on process involves three parties. Alice first visits a web application and elects to use SSO to login. She is then redirected to the identity provider's SSO entry point (e.g., Facebook's server). After she logs into Facebook, her OAuth credentials are issued to the application server. The application server confirms the identity and authenticates the client.

OAuth uses three different types of (rather confusingly-named) credentials:

Access_token. An *access_token* represents permissions granted by the user. For example, the application may request that user grant permission to access the birthday and friend lists from her Facebook account. Upon the user's consent, a token will be issued and forwarded to the application which may then use it to obtain the granted information from Facebook. An *access_token* eventually expires, but may be valid for a long time.

Code. A *code* is used to exchange for an *access_token* through the identity provider. This exchange requires the application's unique *app_secret* to proceed. If the secret does not match, Facebook will not issue the token. This means a *code* is bound to a user as well as a target application. With Facebook SSO, the *code* expires after being used in the first exchange.

Signed_request. A *signed_request* is a base64 encoded string that contains a user identity, a *code*, and a signature that can be verified using an application's *app_secret* and some other metainformation. Once issued, it is not tied to Facebook (except for the enveloped *code*), and the signature can be verified locally.

2.2 Vulnerabilities

Our interest in building an automatic scanning tool was initially motivated by the *access_token* misuse vulnerability reported by Wang et al. [27]. We further identified four new vulnerabilities that are both serious and suitable for automatic testing. The first two vulnerabilities concern confusions about how authentication and authorization are done; the other three concern failures to protect important secrets.

Access_token misuse. This vulnerability stems from confusion about authentication and authorization. In OAuth 2.0, an *access_token* is intended for authorization purposes only because it is not tied to any specific application. When a service uses an *access_token* to authenticate users, it will also accept ones granted to any other application. Figure 1 illustrates an impersonation attack that exploits this vulnerability: Alice visits Mallory's website (step 1), logs in using Facebook SSO (2), and receives an *access_token* from Facebook (3). Then, Mallory's client-side code running in Alice's browser forwards the *access_token* to Mallory (4), which presents the token to a vulnerable application's server (5). After confirming the token represents Alice, Foo's application server authenticates Mallory as Alice (6).

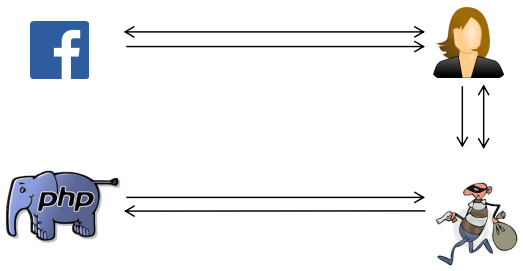

Figure 1: OAuth credential misuse

Signed_request misuse. Sometimes developers have chosen the correct OAuth credentials to use, but still end up with a vulnerable implementation. One way this happens is when information decoded from a *signed_request* is used but the signature is never checked using the *app_secret*. The attack to exploit this vulnerability is

similar to the previous one, except that Mallory needs to reuse the *signed_request* in addition to *access_token*.

App_secret leak. When a developer registers an application with Facebook, she receives an *app_secret*. It is essential for the application owner to keep it a secret because the *app_secret* is used as the key to create *signed_request*s and to access many other privileged functionalities. However, careless developers may reveal this secret to clients, especially when using *code* flow to authenticate users. By design, the *code* and *app_secret* must be sent from the application's back end server to Facebook in exchange for an *access_token*. When this exchange is carried out through the client instead of the server, *app_secret* is exposed to any malicious client.

User OAuth credentials leak. The last two vulnerabilities both leak a user's OAuth credentials. When the Facebook OAuth landing page contains third-party content, requests to retrieve those contents will automatically include OAuth credentials in the referer header, which leaks them to the third-party. To thwart this leakage, Facebook offers a layer of protection by only allowing *access_token* and *signed_request* to appear in the URL fragments, which are not visible in the referer header. Therefore only *code* can be leaked via referer unless the application intentionally pulls the credentials and puts it in the URL[1]. In addition, credentials can be exfiltrated by third-party scripts if they are present in the page content. If a malicious party is able to obtain these credentials, it could carry out impersonation attacks or perform malicious actions using permissions the user granted the original application, such as posting on the user's timeline or accessing sensitive information. Note the difference between embedding OAuth credentials in the URL and in the body content is that the former will directly leak them to third parties, while the latter only leaks the credential when the embedded third party code accesses it explicitly.

2.3 Related Work

Our work builds on extensive previous work on automatically testing applications for vulnerabilities. We briefly describe relevant approaches next, as well as previous works that analyze vulnerabilities in SSO services.

Program analysis. Program analysis techniques such as static analysis [3] and dynamic analysis including symbolic execution [7, 17] automatically identify vulnerabilities with fast testing speed and good code coverage. Runtime instrumentation techniques such as taint tracking [11] and inference [18] also help to safeguard sensitive source-sink pairs. However, these techniques require white-box access to the application (at least at the level of its binary), which is not available for remote web application testing. Automated web application testing tools that work on the server implementation [1, 8, 16] do not apply to large-scale vulnerability testing well. They either require access to application source code or other specific details such as UML or application states. For our purposes, the test target (application server implementation) is only available as a black box.

Automated security testing. Penetration testing is widely used to check applications for vulnerabilities [15, 28]. The tester analyzes the system and performs simulated attacks on it, often requiring substantial manual effort. More automated testing requires an oracle to determine whether or not a test failed. Sprenkle et al. developed a difference metric by comparing two webpages based on DOM structure and n-grams [21] and improved results using machine learning techniques [22]. SSOScan also requires an oracle (Section 3.2) to determine session identity. For our purposes, a targeted oracle works better than their generic approach.

Automated GUI testing. SSOScan is also closely related to automated GUI testing. The GUI element triggering approach we take shares some similarities with recent works to simulate random user interactions on GUI element to explore application execution space on Android system [14], native Windows applications [29], and web applications [5, 10]. Their common goal is to explore app execution space efficiently to discover buggy, abnormal or malicious behavior. By contrast, our goal is to drive the application through a particular SSO process rather than explore its execution space. Further, we need the tests to proceed fast enough for large-scale evaluation. Since each simulated user interaction with the web application involves round-trip traffic and a non-trivial delay to get the response, our primary focus is to develop useful heuristics to quickly prune search space before triggering any user interactions.

SmartDroid [32] and AppIntent [31] both aim to recover sequences of UI events required to reach a particular program state or follow an execution path obtained from static analysis. These approaches target Android applications and rely on client-side information that is not available for our web application scanning tool, where the necessary state only exists on the (inaccessible) server side.

Human cooperative testing. Off-the-shelf testing tools like Selenium [19] and TestingBot [24] can be used to discover bugs in web applications under developers' assistance. These tools replay user interactions based on testing scripts that are manually created by the applica-

[1] Surpsingly, we found several sites doing this (e.g., dealchicken.com and bloglovin.com).

tion developer. BugBuster [6] offers some automatic web application exploration capabilities, but still does not understand the application context enough to perform any non-trivial actions such as those involving authentication and business logic.

To reduce developer effort, Pirolli et al. [13], Elbaum et al. [9], and the Zaddach tool [12] show promising results by collecting interactions from normal users and replaying them to learn application states and invariants for vulnerability scanning. These works do not require extra manual effort from developers to write testing script or specify user interactions. However, one potential problem these works fail to address is user's privacy concerns when submitting interactions. This could be especially sensitive when the actions involve passwords or payments. SSOScan avoids this problem and is complementary to this line of work — SSOScan attempts to scan applications in a fully automatic fashion and does not require traces from any party.

Single sign-on security. Single sign-on has emerged as an important security service and has been well-studied in recent years. Previous works have discovered problems in protocols, bugs in SDK code and missed assumptions in developers' implementations [4, 20, 23, 25, 27]. Automated scanning is especially valuable for vulnerabilities that cannot be simply fixed by upgrading SDKs or improving the protocols, but stem from mistakes integrating the SSO service.

Integuard [30] and AuthScan [2] have similar goals with SSOScan. Integuard infers invariants across requests and responses and uses them to perform intrusion detection on future activities. AuthScan [2] is an automated tool to extract specifications from SSO implementations by using both static program analysis and dynamic behavior probing. Our goals differ in that we focus on detecting specific vulnerabilities rather than generic ones. This enables us to establish clear automation goals and build well-defined state machines for the scanner, and removes the uncertainties the previous works incur when inferring invariants or modeling unknown functions. The drawback is our approach relies on knowledge of particular vulnerabilities. For many integrated web services, including SSO, many vulnerabilities are known or can be obtained using systematic explication [27].

3 SSOScan

SSOScan consists of two main parts: the *Enroller* and the *Vulnerability Tester*. The Enroller automatically registers two test accounts at a web application using Facebook SSO. The Vulnerability Tester simulates attacks and monitors traffic to test for each vulnerability. In this section, we describe the general workflow of these modules necessary to understand the results in Section 4, but defer the details of our heuristics to Section 5.

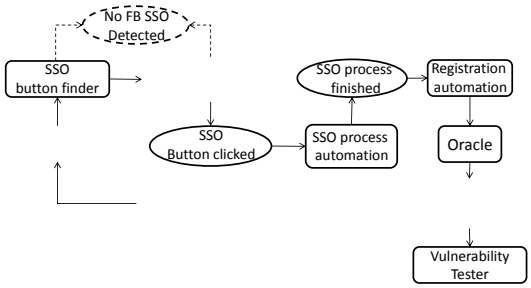

Figure 2: Enroller Overview.
Ovals represent testing states, curved rectangles represent different modules in our tool, and diamonds represent control flow decisions.

3.1 Enroller

Figure 2 shows the workflow of the Enroller. Given a target web application, our tool first removes all cookies from the browser and navigates to the target URL. A short delay after the page has fired its onload event, the SSO button finder (Section 3.1.1) analyzes the DOM and outputs the most likely candidate elements for SSO button. The Enroller then simulates clicks on those elements, monitoring traffic to listen for the Facebook SSO traffic pattern. Once a click or sequence of clicks is found that produces the recognizable SSO traffic, SSOScan automatically logs into Facebook and grants the requested permissions to the application.

About 44% of sites we tested still require a user to register when using SSO, so it is important to automate this process. SSOScan combines heuristics with random inputs to fill in and submit the forms (Section 3.1.2), and then uses an oracle (Section 3.2) to determine if the submission succeeds. If the oracle deems the registration to be a failure, the Enroller tries using different strategies (Section 5) until either the oracle passes or a threshold level of effort is exceeded. The entire process succeeds for 80% of the websites using Facebook SSO in the top 10,000 sites (Section 4 presents detailed results).

3.1.1 SSO Button Finder

A typical starting page, taken from huffingtonpost.com, is shown in Figure 3. SSOScan needs to first find and click the "Log in" button on the main page, and then the "Log in with Facebook" button on the overlay that pops up afterwards. As illustrated in Figure 4, SSOScan first extracts a list of qualifying elements from all nodes in an HTML page, and then extracts content strings from such elements. The Button Finder relies on the assumption that developers put one of a small pre-defined set

of expected words in the text content or attributes of the SSO button. It computes a score for each element by matching its content with regular expressions such as [Ll][Oo][Gg][liOo][Nn] which indicates its resemblance to "login". SSOScan forms a candidate pool consisting of the top-scoring elements and triggers clicks on them. (Section 5 describes the heuristic choices SSOScan uses to filter elements and compute scores.)

3.1.2 Completing Registration

The required interactions to complete the registration process after single sign-on vary significantly across web applications. They range from simply clicking a submit button (e.g., Figure 5, in which all input fields are pre-populated using information taken from the SSO process), to very complicated registration processes that involve interactively filling in multiple forms.

SSOScan attempts to complete all forms on the SSO landing page by leaving pre-populated fields untouched and processing the remaining inputs in the order of radios, selects, checkboxes and finally text inputs. We found this ordering to be very important to achieve higher automation success, as some forms may dynamically change what needs to be filled upon selecting different radio or select elements. Processing these elements first allows SSOScan to rescan for dynamically generated fields and process them accordingly.

For radio and select elements, SSOScan randomly chooses an option; for checkboxes, it simply checks all of them. For text inputs, SSOScan tries to infer their requirements using heuristics and provide satisfactory mock values. Once all the inputs have been filled, the next step is to reuse the SSO Button Finder (Section 3.1.1) with different settings designed to find submit buttons. After SSOScan attempts to click on a submit button candidate, it refers to the oracle to determine if the entire registration process is successful.

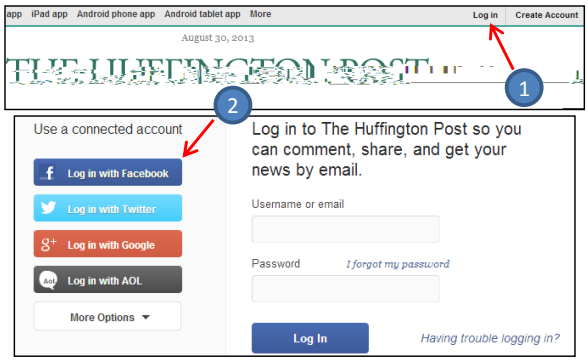

Figure 3: SSO Buttons (huffingtonpost.com)

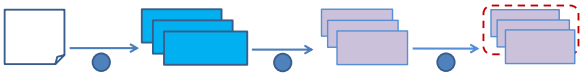

Figure 4: SSO button finder workflow

3.2 Oracle

The Oracle analyzes the application and determines whether it is in an authenticated state, and if so, further identifies the session identity. This module is necessary for SSOScan to decide if a registration attempt is successful. It is also used by the Vulnerability Tester to determine if a simulated impersonation attack succeeds.

The key observation behind the Oracle is that web applications normally remove the original login button and display some identifying information about the user in an authenticated session. For example, after a successful registration many websites display a welcome message that includes the user's first name.

After the page finishes loading, the Oracle searches the entire DOM and **document**.cookie for test account user information (e.g., names, email, or profile images). We evaluate the correctness of our assumptions and effectiveness of our Oracle in Section 4.2.

3.3 Vulnerability Tester

After the Enroller successfully registers two test accounts, control is passed to the Vulnerability Tester which checks the target application for the vulnerabilities described in Section 2.2. We use two different probing approaches to cover the five tested vulnerabilities: *simulated attacks* and *passive monitoring*.

Simulated Attacks. The two credential misuse vulnerabilities are tested using simulated impersonation attacks. We describe how this is done for *signed_request* misuses; the method for checking *access_token* misuses is similar.

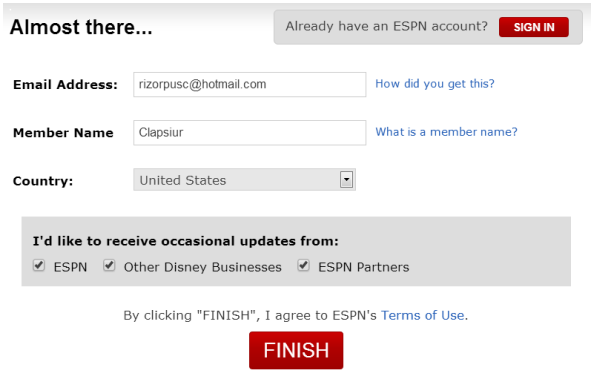

Figure 5: Registration Form (espn.go.com).

To set up the tests, we created a test application *Mal* which uses Facebook SSO, and obtained Alice's *signed_request* for *Mal*. This mimics the scenario where Alice is tricked into visiting and signing into an arbitrary malicious website using Facebook. After the account registration finishes, we use Bob's credentials to sign into Facebook for target application, but replace the *signed_request* in Facebook's response with the prior response received for Alice. For consistency, we also replace all *access_token*s found in the traffic.

The attack is successful if Bob is able to login as Alice using the replayed *signed_request*. The Vulnerability Tester deems the site vulnerable if the Oracle determines that Alice is logged in after the simulated attack.

Passive Monitoring. The three credential leakage vulnerabilities are detected using passive approaches. For brevity, we only explain how leaks through the referrer header are detected; the other leaks are detected similarly by observing network traffic and web page contents.

To check if an application leaks the user's OAuth credentials through the referrer header, SSOScan monitors all request data during the account registration process and compares each referrer header to OAuth credentials recorded in earlier stages. If a match is found, SSOScan then checks if the requesting page contains any third-party content such as scripts, images, or other elements that may generate an HTTP request. SSOScan reports a potential leakage when credentials are found in the referrer header for a page that contains third-party content.

4 Results

We evaluated SSOScan by running it on the list of the most popular 20,000 websites based on US traffic downloaded from quantcast.com as of 7 July 2013. Of those 20,000 sites, 715 of the sites are shown as *hidden profile* (that is, no URL is given, thus excluded from our study).

We ran SSOScan on the remaining 19,285 sites in September 2013, and found that homepages of 1372 sites failed to load during two initial attempts (most likely due to either expired or incorrect domain name, server error, or downtime). We excluded these sites from our data set, leaving a final test dataset containing 17,913 sites.

Completing the tests took about three days, running 15 concurrent sessions spread across three machines. The average time to test a site is 3.5 minutes. We limited the maximum stalling time for each site on any one module to four minutes, and the overall testing time to 25 minutes per site. If this timeout is reached, SSOScan restarts and retries a second time before skipping it. We ran extra rounds on tests that failed or stalled during initial round until either the test is completed or the four rounds maximum limit has been reached. The extra rounds involved fewer sites (<10%) and took a week to complete running on one machine with four concurrent sessions.

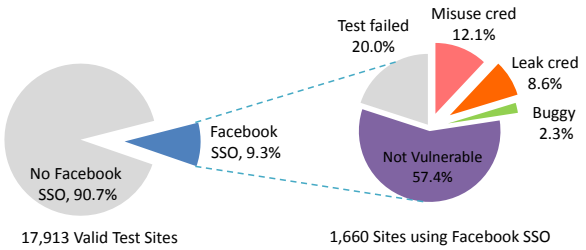

Figure 6: Results overview

In July 2014, we re-ran the tests on the vulnerable sites to see how many sites had corrected the vulnerabilities. The results from that scan are reported in Section 6.2.

4.1 Automated Test Results

Figure 6 presents results purely based on automatic tests run by SSOScan. SSOScan found a total of 1660 sites using Facebook SSO among the 17,913 sites (9.3% of the total). Figure 7 shows the number of Facebook SSO supported sites, sites that misuse credentials, and sites that leak credentials distributed by site ranking. The dotted lines on top of the bars show the average stats of all sites that are more popular than that rank. In Section 4.3, we report on our manual analysis on failed tests for sites ranked in the top 10,000.

Facebook SSO integration. Figure 7 (a) shows that more popular sites are more likely to integrate Facebook SSO. Of the top 1000 sites, 270 (27%) of them include Facebook SSO, compared to only 52 out of the 1000 lowest-ranked sites in our dataset. This supports our believe that covering the top-ranked 20,000 websites is sufficient to get a clear picture of prevailing Facebook SSO usage since less popular sites are both less visited and less likely to use Facebook SSO.

Faulty implementations. To implement Facebook SSO, an application must be configured correctly in the Facebook developer center. Using incorrect parameters to call the SSO entry point also result in errors that will prevent any user from authenticating to that application through SSO. Such cases, automatically identified by SSOScan, were more common than we expected. The most popular errors include setting the application to 'sandbox' mode (for development stage only) in the developer center, or providing a wrong application ID. SSOScan found 39 (2.3% out of 1660 sites that incorporate Facebook SSO buttons) sites that display visible Facebook SSO buttons but have implementations so buggy that no user could ever login using them. A possible explanation is that the

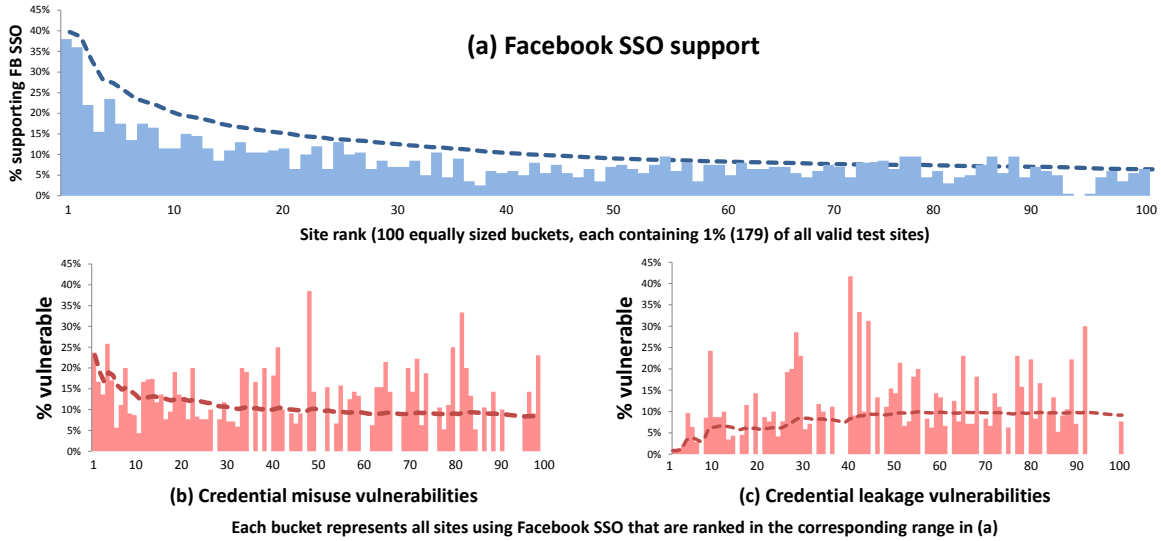

Figure 7: Facebook integration results by site rank

buttons are there for SEO purposes and the developers never actually bothered to implement it, or the developers simply copied and pasted an SSO snippet customized for another application without ever testing it.

Vulnerability trends. We found 202 sites (12.1%) that misuse credentials (126 of which are misusing both *access_token* and *signed_request*) and 146 sites (8.6%) that leak Facebook SSO credentials (of which 72 sites are leaking through both referrer headers and DOM). A total of 345 sites (20.3%) suffered from at least one of the five tested vulnerabilities, and 3 sites suffered from both credential misuse and leakage problems.

It is also worth noting that SSOScan did not find any sites leaking their *app_secret* to the public by calling the token exchange API on the client side. To verify that we implemented the check correctly, we have confirmed that SSOScan does correctly identify this vulnerability on our manually-crafted faulty application. This is an interesting result, especially compared to the high number of sites that have at least one of the other vulnerabilities. We suspect this is partly due to explicit warnings in the documentation and the increased effort required to actually implement the token exchange on the client side.

As shown in Figure 7 (b) and (c), more popular sites appear to be more likely to have credential misuse vulnerabilities, while less popular sites tend to have more credential leakage problems. This fact certainly raises concern — credential leakage could potentially do damage to users' Facebook accounts, and it would be hard to contact numerous low-profile problematic sites to have them all fixed. The victim's Facebook account is in jeopardy if any of the applications he or she uses have such problem. Even though credential misuse cannot harm Facebook accounts directly, the fact that such vulnerabilities exist in high-profile websites is worrisome, as impersonation attacks carried against sites with millions of users have more severe consequences thank similar attacks on lower-profile sites.

Of the top-1000-ranked sites, 60 of the 270 (22.2%) that support Facebook SSO are found to have at least one vulnerability. The vulnerability rate is 21.3% across all sites in the top 10,000 and 18.5% for sites ranked from 10,001 to 20,000. This overal vulnerability rate suggests that development practices at larger companies do not appear to be more stringent (at least with respect to SSO) than they are at less popular sites.

As we do not have access to server side source code, we cannot measure how reusing code may positively or negatively affect the vulnerability trend. However, we did notice that some sites use fourth party services (e.g. Janrain, Gigya) to implement the Facebook SSO. In such scenarios, the user effectively does two SSO processes during authentication — the user, Facebook (IdP) and Janrain (RP) initially; the user, Janrain (IdP) and the true relying party afterwards. As the Facebook SSO process is entirely handled by the fourth party and is hidden to the relying party, the RP's behavior is not relevant to this vulnerability. We have manually tested both Janrain and Gigya's Facebook SSO implementation for credential misuse vulnerabilities and confirmed that both of them correctly implement the process by only using *code* flow to authenticate users. As a result, sites using these services contribute to a lower vulnerability rate. Note that the RP would still need to implement the second SSO process correctly to avoid vulnerabilties, but SSOScan currently

does not check IdPs other than Facebook.

4.1.1 Front-end Integration

There are three basic client-side methods to integrate Facebook SSO: a JavaScript SDK, a pre-configured widget, or a custom implementation. (We have no way to determine how the developers are integrating Facebook SSO at the back end.) We used SSOScan to aggregate front-end integration choices and compare them with vulnerability reports. Table 1 summarizes the results. Websites using client side SDKs and pre-configured widgets are more likely to misuse credentials (29.1% and 15.5% vs. 1.3% in non SDK/widget implementations). Our guess is that this is due to the way SDKs and widgets conveniently expose raw *access_token*, *signed_request*, or even user name Facebook ID values. This convenience may lead to the developers to neglect to check the signature and the intended audience of the credential. However, our results also show that websites using SDKs and widgets are better in hiding credentials (3.6% and 2.2% compared to 12.4% vulnerable rate in SDK/widget implementations). This is likely because such applications use the Facebook-provided landing page which has safe redirect URLs and no third-party content. Applications built this way are secure unless the developers explicitly add the credentials in the page content or URL.

4.1.2 Examples

We describe two examples of vulnerabilities found by SSOScan here to illustrate the potential risks. Section 6 discusses our experiences reporting vulnerabilities to site owners and Facebook.

Match.com. Ranked 118th on the list, Match.com is a popular online dating website. SSOScan revealed that match.com is also vulnerable to *signed_request* replacement attacks. To use match.com services, users need to provide sensitive information including their birthday, location, photos, personal interests, and sexual orientation. Impersonators will not only have access to this information, but also learn whom the victim is dating and possibly the time and location of the dates.

Method	Number	Misuse	Leakage
SDK	578	29.1%	3.6%
Widget	132	15.5%	2.2%
Custom Code	950	1.3%	12.4%
All	1660	12.1%	8.6%

Table 1: Rate of credential misuse and credential leakage for different Facebook SSO front-end implementations

Fodors.com. Fodor's is a travel advice website that is the 217th-ranked US site. Its redirection landing page contains *access_token* information along with some other third-party scripts in its content. The scripts come from various sources including quantserve.com, fonts.com, yahooapis.com, and multiple domains owned by Google. The permission Fodor's requests includes user's basic information, email address, and more importantly, permission to post to user's wall on the his or her behalf. This means if the *access_token* is leaked to a malicious party, it can post to a user's Facebook wall without consent in addition to accessing the user's basic information.

4.2 Detection Accuracy

To evaluate the detection accuracy of SSOScan, we sampled test cases from all results (including sites reported to have no Facebook SSO support, secure and vulnerable cases) and manually examined them. We consider two types of mistakes: misreporting whether the site integrates Facebook SSO, and incorrectly determining whether or not a Facebook SSO-enabled website exhibits a vulnerability.

Facebook Login Detection Correctness. SSOScan searches SSO button based on heuristics and cannot guarantee success for all websites. Indeed, it is not possible for anyone to determine with complete confidence if a website uses Facebook SSO by just browsing the site. To roughly measure how many Facebook SSO-enabled websites were missed by SSOScan, we randomly sampled the 100 sites that were reported by SSOScan to have no Facebook SSO support and manually examined them. To make the samples representative of the whole set, we picked one site out of every 200 sites ordered by their rank. From manually investigating these 100 sites, we could only find one site that included Facebook SSO but was missed by SSOScan. As we introduce later in Section 6, we also deployed SSOScan as a web service that is made available to use in our research group. The web service has received a total of 69 valid submissions so far and we have also manually examined the vulnerability reports.[2] We found four cases (5.8%) where a submitted site included Facebook SSO but SSOScan was not able to trigger it.

The sites that SSOScan fails to find Facebook login present unusual interfaces which our heuristics are not able to navigate to. Specifically, oovoo.com and bitdefender.com do not show any login button on its

[2] These have mostly been sites suggested by people we have demoed SSOScan scan to, since the service has not yet been publicized. Hence, it is a small and non-representative sample, so not clear what we can conclude from this at this point.

homepage, but instead the user needs to click a 'my account' button to initiate the login process. The sears.com site displays a login button on its homepage, but the SSO process is not initiated until the user interacts with the popup window three times, which exceeds the maximum click depths (two) in this evaluation. We have also seen one case (coursesmart.com) in which the login process is rather typical, but SSOScan still missed the correct login button (that button is scored the 4th highest while SSOScan only attempts to click the top 3 candidates.). Most of these issues may be addressed with more relaxed restrictions and more regular expression matching as described in Section 5.2. Finally, our prototype implementation is limited to English-language websites due to its string matching algorithm, but could be extended to include keywords in other languages.

SSOScan may also incorrectly conclude that a website supports Facebook SSO when it does not. We have seen sites (e.g., msn.com) that only use Facebook SSO to download user activities and display them on the page, but do not integrate their identity system with Facebook SSO. Although SSOScan is designed to skip searching on typical Facebook-provided social plugins and widgets, non-standard integration of such functionalities may rarely lead to false positives.

Vulnerability Status Correctness. Since SSOScan simulates potential attacks and verifies their success or failure, detection is likely to be highly accurate. Nevertheless, we consider several possible reasons that might cause false positives/negatives to be reported.

SSOScan should be able to capture all credential leakage vulnerabilities with no false positives. A false negative may occur since SSOScan only looks for exact matches to the original OAuth credential string, so it will not report a leakage if the credential is slightly transformed or encoded. Further, SSOScan only observes traffic involving the web client, so does not detect application that leak OAuth credentials outside the SSO process.

SSOScan only reports a credential misuse vulnerability when it can successfully execute an impersonation attack. So, the only risk for incorrect reports is if the Oracle incorrectly determines the session identity. We designed the Oracle to minimize this risk. For example, information for the test account is chosen carefully to be unlikely to appear otherwise but to be close enough to real names to pass sanity checks. For example, the randomly generated name "Syxvq Ldswpk" was rejected by a small number of websites, but "Jiskamesda Quanarista" always passed sanity checks and only appeared in an authenticated session in all of our tests. Barring an unlikely name collision, there does not appear to be any way SSOScan would produce a false positive credential misuse report.

The Oracle checks the whole response for identifying information instead of only the DOM content to handle sites which only embed such information in first-party cookies after logging in. In some rare cases, these cookies could be issued even before SSOScan finishes registration forms. This means that before the Enroller searches for registration forms to fill in, the Oracle deems registration as unnecessary because it concludes that the application is already in an authenticated state. Although SSOScan is able to proceed and determine vulnerability status, the application never enters an authenticated state and false negatives might occur.

Trusted Third-Party Domains. For credential leakage vulnerabilities, SSOScan reports an application as vulnerable if it identifies visible credentials co-existing with *any* content or script that comes from *any* origin other than the host or Facebook. This could overestimate the vulnerable sites because the host may own other domains and serve content over them, which should not be considered untrusted. For example, content delivery networks and sub-company scenarios (e.g., cnn.com embedding content from turner.com which owns CNN) are common among popular websites.

4.3 Automation Failures

For about 19% of the top 10,000 tested site that include functional Facebook SSO, SSOScan is not able to fully automate the checking process. Figure 8 shows the distribution of rank of failed test websites.

To better understand the reasons why SSOScan fails, we manually studied all 228 failed cases reported by SSOScan for sites ranked in the top 10,000. We found that although 47 out of these 228 cases set their Facebook application configurations and SSO entry points properly, they never respond to credentials returned by Facebook SSO, which means no users would be able to successfully log into these sites through Facebook SSO.

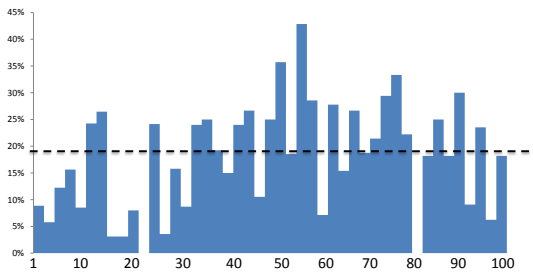

Figure 8: Failed tests rank distribution

Failure reason	Number	Percent
linking/subscription	51	28.1%
CAPTCHAs	34	18.8%
identity invisible to oracle	28	15.5%
atypical input elements	20	11.0%
atypical submit buttons	19	10.5%
email verification	10	5.5%
non-HTTPS submission forms	9	5.0%
other (e.g., timeouts)	10	5.5%
Total failures	181	100.0%

Table 2: Automation Failure Causes (top 10,000 sites)

Excluding these 47 left us a total of 181 failure cases.

Registration automation failure. By far the most common reason for SSOScan to fail is due to complicated or highly-customized registration process. We found 43.7% of the sites that implement Facebook SSO still require users to perform additional actions to complete the registration (roughly evenly distributed by site popularity). SSOScan failed to complete registration on 143 (33.6%) of them. Table 2 shows the major reasons contributing to this failure ordered by their occurrences: 1) sites that require SSO users to link to an existing account or provide payment information to subscribe to the service; Currently SSOScan cannot handle the "linking" action: automatically registering a "traditional" account and perform the linking poses an out-of-scope challenge — doing so often requires solving CAPTCHAs[3]. 2) registration forms after the SSO process include CAPTCHAs; 3) special input elements (e.g. div, span or image as opposed to input) cannot be found automatically, or special requirements for the input that cannot be fulfilled; 4) sites where the registration submission button cannot be located; 5) sites that requires users to confirm email addresses before continuing (usually this involves clicking a link in an email sent by the server to the user's email address); and 6) sites that insecurely send registration data using a non-HTTPS form which causes the testing browser to pop up a warning and stall.

Oracle confusion. SSOScan may also fail because the oracle reports failure (15.5%), which occurs when it detects the login button no longer exists after Facebook SSO but cannot identify the session identity. We manually analyzed such cases and found the biggest obstacle is that the application homepage does not include any identifying information at all. For example, instead of showing 'Welcome, *f*username*g*', it shows 'Welcome, customer', or simply 'Welcome', and the user name is only displayed when accessing the account information page. In other cases, SSO authentication serves only a sub-service of the website such as its affiliated forum, but not the homepage which does not display any identifying information.

Others. During the testing, we have also seen a number of sites with extremely long loading time or inconsistent network latencies after Facebook SSO or upon navigating to certain pages. While the latency spikes can likely be resolved by re-running the tests, frequent long delays which accumulate to SSOScan's maximum timeout will always halt the automation process. For example, this happens when SSOScan accidentally triggers a browser confirmation dialog that requires user interaction, or asking users to stop a busy script execution.

5 Heuristics Evaluation

The ability of SSOScan to successfully complete the Facebook single signlre--2265 f SSOvide SSO

[3] On the contrary, most tested applications (942 out of 973, see Section 4.3) do not ask users to solve CAPTCHAs when an account is created through SSO. This is a reasonable practice, since the user who is able to provide a valid Facebook account should have already passed Facebook's requirements, and adding additional CAPTCHAs would be unnecessarily annoying to the users.

Candidate rank. The button finder produces a candidate element list ranked by score. SSOScan will first attempt clicking on the highest-ranked element, but sometimes the correct element is ranked lower. This option controls the maximum number of click attempts SSOScan makes before succeeding or giving up. For Section 4's experiment, the lowest ranked element SSOScan clicks is the third.

Visibility filter. Most websites only expect users to click on UI elements that are visible, so the button finder includes a filter that ignores all invisible elements (e.g., elements with zero height or width, shadowed in the background layer, or those which appear only when the user scrolls the initial screen position).

Position filter. We noticed that SSOScan sometimes gets distracted by a search box submit button when completing the registration form, even if it is able to correctly fill in the required information in all input elements. To eliminate these misclicks, the position filter eliminates the submit buttons which are displayed above any inputs based on our observation that submit buttons nearly always come last in a registration form.

Registration form filter. As mentioned earlier in Section 4.3, many websites provide two actions for the user after SSO is completed: 'create new account' or 'link an existing account'. The latter option requires the user to enter the user name and password of an existing account to finish the enrollment process. To avoid these, the registration form filter rejects a candidate submit button if its parent form contains only two visible text inputs, one has the meaning of 'name' or 'email' and the other is of type password, since such an element is most likely to be a submit button of a linking form.

Element content matching. SSOScan searches for elements whose labels are close to "login with Facebook" for SSO buttons by default. However, quite a few popular websites (e.g. coupons.com, right side of Figure 9) only allow users to "sign up with Facebook" first before logging in with Facebook. If the user has yet to do this, attempting to login with Facebook will produce an error. To handle this situation, SSOScan will search for elements with semantics similar to "sign up with Facebook" when it fails to register using the "login" buttons.

A filter may significantly reduce the number of misclicks. However, it may also occasionally exclude correct elements. For example, not every correct submit button is below all inputs (e.g., left of Figure 9, and expedia.com's submit button would have been missed with the element position filter enabled).

Hence, SSOScan is designed to explore target sites using different option settings if enrollment does not succeed with the initial settings. It will continue to attempt to complete the enrollment process using different settings until either all configurations have been exhausted or the timeout threshold is reached. SSOScan avoids doing duplicate work by detecting if a click attempt has resulted in a previously visited or completely explored state (see Appendix B for details).

5.2 Experiment Setup

In theory, SSOScan could exhaustively trigger clicks on every element on the page (and on all response pages up to some maximum depth), which would result in nearly 100% success rate. This would be prohibitively slow in practice, though, so the number of attempted clicks must be limited for any realistic test. Given the time needed for each click attempt, it is important to configure our scoring heuristics well to maximize the probability of a successful enrollment in the minimum amount of time.

To gather statistics about the candidate elements, we modified SSOScan to try all possible strategies even if it has already *found* the correct login button and to record information about all attempted clicks, including for example their size, position, visibility to the user, content string feature and whether it is *successful*. We define a click as *successful* if it is included in any sequence of clicks from the start page to triggering the SSO process, regardless of whether it appeared in an attempt that failed to trigger the process. Because SSOScan skips previously explored states to avoid redundant effort, it automatically rejects click sequences which involves cyclic state transitions such as clicking on an irrelevant link and then clicking on a logo which returns to the initial state.

We set up SSOScan to expand the candidate pool size for each configuration from 3 to 8, add more matching regular expressions (e.g., to match the string "forum" which occasionally leads to a login page on sites where no login is visible on the start page), and use equal weight for each of them. We also removed all candidate filters described in Section 5.1. Our goal is to capture as many ways to trigger the SSO process as possible by doing as close to an exhaustive search as is feasible. This increases the time required to scan a typical site to almost an hour (compared to a few minutes with the setup used in the full study).

We ran the test on the 973 sites from the top-10,000 ranked sites that were detected by SSOScan to support Facebook SSO in our main study (Section 4). This biases the study slightly, since it only includes sites where the configuration used in the initial study was able to find Facebook SSO. Ideally we would like to run all top-10,000 sites to avoid any bias introduced by the data set, but the significantly increased testing time prohibits us to do so, and the result of our subsequent study on a random

sample of sites (Section 5.4) supports the claim that only few sites containing Facebook SSO were missed by the main study.

5.3 Results

The experiment recorded 29,539 unique[4] click attempts, of which 5086 (17.2%) are successful (that is, they either directly trigger SSO, or lead to subsequent clicks that trigger SSO). This amounts to approximately 30 unique clicks attempted per site, but the number varies significantly based on the site design, from a few up to 109.

Element type and content. Figure 10 shows how different button types and properties impact success rates. We report the success rate as the number of times that element appeared as a successful click divided by the total number of clicks attempted on elements of that type. The number beneath the element feature gives the total number of times that type of element occurred as a successful click target across all the test sites. For example, the *BUTTON* element type has an excellent success rate — 60% of all *BUTTON* candidates are true positives for the Facebook SSO button. But since it only appears as a successful click on 78 out of 973 sites in our sample, it is rarely useful. By contrast, clicking on *DIV* elements are much less likely to trigger the Facebook login, but such elements are more common. The right side of Figure 10 shows that elements that are directly visible to the user has a higher success rate than invisible ones, and elements residing in iframes are twice as likely to be

[4] If two clicks happens on pages with the same URL, same element XPath and same element outerHTML, we consider them the same click.

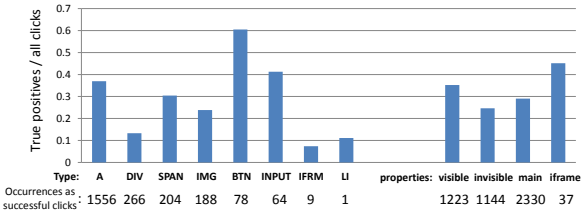

Figure 10: Login button type statistics

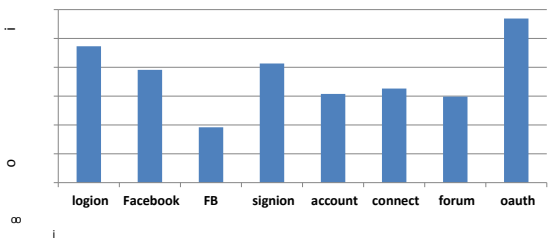

Figure 11: Login button content statistics

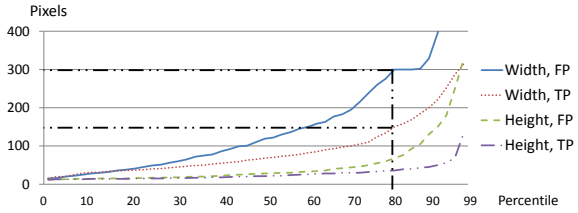

Figure 12: Impact of Login Button Size

the correct target as their counterparts in the main page. These results suggest ways of weighting element types to improve the scoring function and increase the likelihood of finding successful clicks early.

Figure 11 shows how the success rate varies with attribute content (matched by the given regular expression). The keyword "oauth" rarely exists in any content, but when it appears it is very likely to identify the target element. The result also shows that "FB" is not a good indicator to predict the target, and we think this is probably because it is very short and may be used for similarly named JavaScript variables or random abbreviations.

Both figures include data for the *first* click only (but do measure first click success based on subsequent clicks). Data for the second clicks are noticeably different from the first, and overall success rates are lower on second clicks. The most interesting fact we found is that "connect" (39%) and "Facebook" (36%) become the most successful matches of all regular expressions, followed by "oauth" at (26%). No other regular expressions exceed 20% success for the second click.

Element size. Figure 12 gives the cumulative distribution function of the width and height of target elements. For example, the 80th percentile width of the true positive elements is approximately 150px, compared to 300px for false positive elements. We did not find any significant difference between first and second clicks, so the figure combines data from all clicks. The key result is that wide elements are less likely to be true positives, possibly due to SSOScan incorrectly including many large underlay elements as candidates. The result is similar for element height (the lower two lines in the figure). This suggests that it would be useful to add a filter function that excludes candidates whose width is greater than 300px. We would expect it to eliminate 20% of the false positives while hardly missing any of the true positives. Alternatively, SSOScan could adjust the final score of a node according to its size based on these results.

Element position. Figure 13 shows the heatmap of the login button's position in a page. The intensity at a location indicates the number of elements found there satisfying the property. Only visible elements are shown, and each successful click only attributes to the intensity

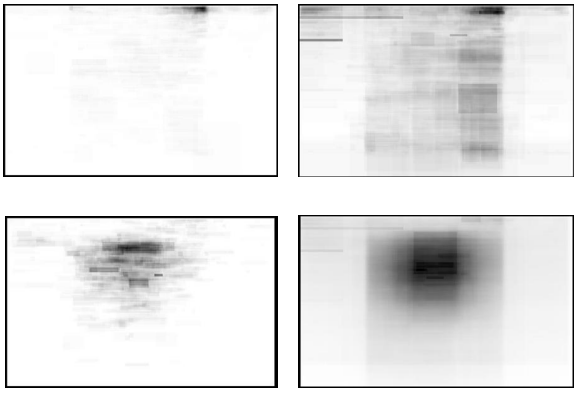

Figure 13: Login button location heatmap

once. All four figures are normalized with respect to their maximum intensity (i.e., element density).

The figures show an interesting distinction from first click to second click: successful first clicks almost exclusively appear in the upper right corner of the page, while the second click appears generally in the upper-middle part of a page. The false positives are relatively more scattered everywhere on the page[5]. This result suggest we should assign a higher weight for elements for these locations, and focus on elements in the vicinity of the upper right corner for the first click. We could potentially even ignore the other criteria and only consider position to find login buttons on foreign-language sites.

5.4 Validation

After incorporating what we learned from these results (e.g., weight adjustment for different button sizes and types), we reran the SSOScan with the new heuristics on the sites ranked from 10,000 to 20,000 that SSOScan determined to support Facebook in the original study, which were not included in the heuristics evaluation. We compare the results with those obtained by using a "control" version of SSOScan, with equal weights on all features and no candidate filtering. All other settings such as candidate pool size are the same between two versions.

The results support the hypothesis that adjusting heuristics according to the results of the evaluation can improve the speed and robustness of detection of Facebook SSO integrations. The naïve control version missed 72 out of the 601 sites while the new heuristics missed only two. The average rank of correct candidate elements for the first and second click is 1.32 and 1.23 for the control experiment, which improves to 1.23 and 1.17 respectively with the new heuristics.

We also randomly picked 500 random sites from the sites that SSOScan have yet to find Facebook support in the experiment in Section 4. We tested the expanded heuristics on these sites, and further increased the maximum click depth to three to see if more SSO integrations could be found. Individual tests took an average of 31 minutes to finish, but varies significantly from a few minutes up to an hour (threshold) based on site content.

Four additional sites were found that support Facebook SSO from this sample in total. Two are found due to the added regular expression [Ff][Oo][Rr][Uu][Mm], one of which required three clicks to trigger the SSO process. Another site is found due to the improved candidate ranking algorithm, and the fourth was found using the new candidate selection method that includes all elements in the right corner of the page, even if they do not match any regular expressions. This provides a reasonable degree of confidence that our original study found a large enough fraction of all the popular sites using Facebook SSO to be representative, although likely missed around 1% of Facebook SSO sites. We did not try click depths greater than 3 because of the exponential time growth required to complete each test, but we feel confident that the number of Facebook SSO interfaces that can only be discovered by attempting more than 3 clicks is very low.

6 Discussion

This section concludes by discussing limitations of SSOScan, sharing our experiences reporting vulnerabilities, and suggesting ways SSOScan can be deployed to help secure applications integrating SSO services.

6.1 Limitations

While SSOScan is able to automatically synthesize basic user interactions and analyze traffic patterns, this approach is not suitable for detecting all types of vulnerabilities. It only works for vulnerabilities that can be checked by observing traffic or simulating predictable user events, and falls short if the vulnerability testing involves deep server-side application scanning or complicated interactions. For example, Wang et al. [27] point out that the application's *app_secret* might be leaked to arbitrary party if any page including Facebook's PHP SDK invokes two functions in a specific way. This type of vulnerability could be checked at the developer side using program analysis techniques, but cannot be checked by an external tool with no awareness of the sites' implementation details or internal state.

[5] The figures also show a clear width boundary. In the experiments the browser resolution is 1920x1200, and it seems that most developers' designs follow a standard width of approximately 960px, which is why the density appears to be cut off.

6.2 Communication and Responses

We started contacting the site owners shortly after obtaining our first list of vulnerable sites, manually sending out notifications to 20 vulnerable websites that we thought were interesting. We contacted them either through email or by submitting forms on their website. The responses were very disappointing, especially compared with our previous experiences reporting SDK-level vulnerabilities to identity providers who tend to respond quickly and effectively to vulnerability reports [27]. The vulnerabilities found by SSOScan, on the other hand, are primarily in consumer-oriented sites without dedicated security teams or clear ways to effectively report security issues.

Of the 20 notifications, we only received 8 responses, most of which appear to be automated. After the initial response, three websites sent us follow-up status updates. ESPN.com thanked us and told us the message has been passed onto appropriate personnel, but no follow up actions ensued. One of answers.com's engineers asked us for more details, but failed to respond again after we replied with proposed fix. As of July 7th 2014, both sites are still vulnerable. Four months after getting the automated reply from ehow.com, we received a response stating that they have removed Facebook SSO from their website due to "content deemed inappropriate", and we have confirmed that the Facebook SSO button has indeed been removed. Sadly, we think their staff likely did not (bother to) understand our explanation for the fix and simply removed the feature for their convenience.

The other instance where a reported vulnerability was fixed was for hipmunk.com. Hipmunk was found to be vulnerable to both the *access_token* and *signed_request* replacement attacks. We did not get any response from Hipmunk when the vulnerability was reported through the normal channels, but through a personal connection we were able to contact them directly. This led to a quick response and series of emails with one of Hipmunk's engineers. We explained how to check the signature of a *signed_request*, which should fix both vulnerabilities. However, when they got back to us believing that the fix was complete, we re-ran SSOScan and found that Hipmunk was still vulnerable to the *access_token* replacement attack. This meant Hipmunk checked the signature of *signed_request* after the fix, but never decoded the signed message body and compared its Facebook ID with the one returned by exchanging *access_token*. This surprised us, as we implicitly assumed the developers will consume the signed message body after verifying its signature, and thus only included 'verifying signature' in the proposed fix. After further explanation, the site was fixed and now passes all our tests.

Retesting vulnerable sites. We retested all 345 vulnerable sites in May 2014, nine months after our initial experiment, including the 20 websites we had notified directly. SSOScan found that 48 of the sites had eliminated the vulnerabilities (including one out of the 20 sites we contacted, mapquest.com). Of the 48 fixed sites, 22 had previously been diagnosed as credential leaking sites, and 27 were misusing credentials (one site, trove.com, fixed both problems). We further examined these sites manually to investigate the possible reasons and measures to fix the problems. As for sites that fixed credential misuses, we found that many had abandoned the *token* or *signed_request* flow in favor of the more secure *code* flow, which automatically protects them from credential reuse attacks. For credential leakages, we found that a number of sites redesigned their SSO process to feature a smoother user experience, e.g., replaced traditional redirection flows with AJAX operations, which naturally eliminated credential leakage via referer header.

Communication with Facebook. Due to the ineffective direct communication with site owners, we reached out to Facebook and were contacted by their platform integrity team in May 2014. Facebook's response indicated that they are particularly worried about *access_token* leakage through referer headers (because a malicious party in possess of the token may perform privileged Facebook actions on behalf of the user, which potentially harms Facebook themselves), but are also concerned with the credential misuse scenario. Facebook asked for a list of the vulnerable applications and contacted all the sites with *access_token* leakage and credential misuse vulnerabilities (a total of 95 sites that we are able to re-confirm at the time of report), and informed us that they would "take enforcement action as necessary" upon the 10 sites that are leaking *access_token*s in

to improve the security of integrated applications.

App center integration. We believe SSOScan would be most effective when used by an application distribution center (e.g. Apple store, Google Play) or identity provider (e.g., Facebook) as part of the application validation process. The identity provider has a strong motivation to protect users who use its service for SSO, and could use SSOScan to identify sites that can compromise those users. It could then deliver warning messages to visitors of vulnerable applications during the log in through Facebook SSO process, or even go so far as to shut down SSO logins for that application. We also believe our results can provide guidance to vendors developing SSO services. The results in Section 4.1 indicate that sites are more likely to misuse credentials when using the Facebook JavaScript SDK. With Facebook's help, this problem could be mitigated by placing detailed instructions inside the SDK. The instructions could be presented as (non-executable) code in the SDK rather than as comments, so that the developers cannot get by without reading and removing them.

Checking-as-a-service. Without involving an centralized infrastructure, the best opportunity to deploy SSOScan is as a vulnerability scanning service that developers can use to check their implementations before their applications are launched (our prototype service at http://www.ssoscan.org/ can be used for this now). For a developer-directed test, it would be reasonable to ask the developer to either guide the tool through the registration process or provide a special test account that bypasses this step in cases where it cannot be fully automated. Even if we assume no aid from the developers, they should at least be able to tolerate a longer testing time than is feasible in doing a large-scale scan.

Availability

SSOScan is available at http://www.SSOScan.org/ as a public web service. The source code is available (linked from that site) under an open source license.

Acknowledgements

We thank Jonathan Burket, Longze Chen, Shuo Chen, Steve Huffman, Jaeyeon Jung, Haina Li, Chris Slowe, Ankur Taly, Rui Wang, Westley Weimer, Eugene Zarakhovsky and anonymous reviewers for their valuable inputs and constructive comments. This work has been supported by a Research Award from Google and research grants from the National Science Foundation and Air Force Office of Scientific Research.

References

[1] N. Alshahwan and M. Harman. Automated Web Application Testing Using Search Based Software Engineering. In *26th IEEE/ACM International Conference on Automated Software Engineering*, 2011.

[2] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Suny, Y. Liuz, and J. S. Dong. AuthScan: Automatic Extraction of Web Authentication Protocols from Implementations. In *20th Network and Distributed System Security Symposium*, 2013.

[3] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *29th ACM Symposium on Principles of Programming Languages*, 2002.

[4] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In *25th Computer Security Foundations Symposium*, 2012.

[5] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically Testing Dynamic Web Sites. In *11th International World Wide Web Conference*, 2002.

[6] BugBuster. BugBuster is a Software-as-a-Service to Test Web Applications. http://bugbuster.com/.

[7] C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *12th International Conference on Model Checking Software*, 2005.

[8] G. Di Lucca, A. Fasolino, F. Faralli, and U. De Carlini. Testing Web applications. In *Journal of Software Maintenance*, 2002.

[9] S. Elbaum, S. Karre, and G. Rothermel. Improving Web Application Testing with User Session Data. In *25th International Conference on Software Engineering*, 2003.

[10] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *12th International Conference on World Wide Web*, 2003.

[11] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *14th Network and Distributed System Security Symposium*, 2007.

[12] G. Pellegrino and D. Balzarotti. Toward Black-Box Detection of Logic Flaws in Web Applications. In

21st Network and Distributed System Security Symposium, 2014.

[13] P. Pirolli, W.-T. Fu, R. Reeder, and S. K. Card. A User-tracing Architecture for Modeling Interaction with the World Wide Web. In *First Working Conference on Advanced Visual Interfaces*, 2002.

[14] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Third ACM Conference on Data and Application Security and Privacy*, 2013.

[15] Redspin Inc. Penetration Testing, Vulnerability Assessments and IT Security Audits. https://www.redspin.com/.

[16] F. Ricca and P. Tonella. Analysis and Testing of Web Applications. In *23rd International Conference on Software Engineering*, 2001.

[17] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *31st IEEE Symposium on Security and Privacy*, 2010.

[18] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. In *16th Network and Distributed System Security Symposium*, 2009.

[19] Selenium development team. Selenium: Web application testing system. https://selenium.org/.

[20] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen. On Breaking SAML: Be Whoever You Want to Be. In *21st USENIX Security Symposium*, 2012.

[21] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated Replay and Failure Detection for Web Applications. In *20th IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[22] S. Sprenkle, E. Hill, and L. Pollock. Learning Effective Oracle Comparator Combinations for Web Applications. In *International Conference on Quality Software*, 2007.

[23] S.-T. Sun and K. Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *19th ACM Conference on Computer and Communications Security*, 2012.

[24] TestingBot. Selenium Testing in the Cloud - Run Your Cross Browser Tests in Our Online Selenium Grid. http://testingbot.com/.

[25] 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 *33rd IEEE Symposium on Security and Privacy*, 2012.

[26] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. In *32nd IEEE Symposium on Security and Privacy*, 2011.

[27] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *22nd USENIX Security Symposium*, 2013.

[28] Whitehat Security. Your Web Application Security Company. https://www.whitehatsec.com/.

[29] Q. Xie and A. M. Memon. Model-Based Testing of Community-Driven Open-Source GUI Applications. In *22nd IEEE International Conference on Software Maintenance*, 2006.

[30] L. Xing, Y. Chen, X. Wang, and S. Chen. InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations. In *20th Network and Distributed System Security Symposium*, 2013.

[31] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *20th ACM Conference on Computer and Communications Security*, 2013.

[32] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

A Additional Options

These options are supported by SSOScan but were not enabled for the experiments described in Section 4.

Test accounts. As mentioned in Section 3.2, we have also seen rare cases that some applications reject the user registration because the test account's name is either too long, too short, or because its newly created password does not meet certain requirements, or due to other requirements such as being from a certain country. Since it is hard to create universally applicable accounts that meet all requirements, we created a pool of eight different Facebook test accounts so that SSOScan can retry with random different test accounts for each site after registration fails.

Element size During the initial investigation, we found that the login buttons that the user need to click to trigger the SSO process have rather predictable sizes — specifically, we have noticed that SSOScan can get confused by many false positives that match many regular expressions but are just big underlays for the target element. Therefore, a filter is implemented that ignores elements with an unreasonable big width or height.

Weight assignments SSOScan computes a score on the extracted content of an candidate element using regular expression matching. However, using the same weight for each expression is unfair — while the expression [Ll][Oo][Gg][liOo][Nn] may likely be matched exclusively by login buttons, [Aa][Cc][Cc][Oo][Uu][Nn][Tt]$ may appear in arbitrary site content as it represents a much more popular word. The same is true for element types, such as A elements could be much more common to be used as a login button than DIV with a JavaScript event handler. The weight assignments can also be dynamically adjusted between attempts for best performance, such as using different values when searching for login buttons for the second click than the first.

Alternative checkbox inputs This option allows SSOScan to try out different checkbox values before submitting the registration form, rather than simply checking all of them. In some scenarios the registration form is asking if the user would like to subscribe to the premium service, which may hinder registration process as it needs user's financial information.

Element visibility requirement By default, SSOScan requires an element to be displayed in the initial browser area to be flagged as visible. This eliminates all elements positioned in lower sections of a document, which might miss some target elements as we observed. Therefore SSOScan provides an alternative to lift this requirement and include the lower positioned elements as visible candidates.

B Implementation Details

This appendix describes some additional techniques that are necessary to improve automation success rate. None of these are particularly interesting or novel, but are included here for completeness since they were necessary to achieve the results reported in our experiments.

Determining element visibility. As introduced in Section 5.1, SSOScan uses a filter that rejects elements which are not directly visible to the user. We implement the filter by using **document**.elementFromPoint API. For any given element E, we first get its current top-left corner coordinates, width and length. Then, **document**.elementFromPoint is called on ten points distributed equally along the diagonal inside this area. If more than five of the elements returned are either E itself or a child of E, E is considered to be on top. Depending on current strategy, SSOScan may call scrollIntoView to include searching elements displayed at lower sections of a document. Using **document**.elementFromPoint, however, has its limitations — it always returns the underlying canvas element when called at any point on that canvas, therefore onTopLayer always returns false for elements placed on canvases. We hope future browsers can fix this bug, but for now SSOScan relies on relaxed strategies (Section 5.1) that include invisible buttons to handle such scenarios.

Avoiding futile and duplicate clicks. To further eliminate false positives in candidate lists as early as possible, SSOScan detects clicks which have no effect or have lead the web application into a previously visited state in the same attempt. Naïvely using URL or full **document**.body.innerHTML string to represent application state does not work well, because these information are not stable as variations exist across different requests and time (e.g. GET parameters and advertising content). Fortunately, getCandidates provides a representative feature naturally — the candidate list. To detect previous clicks that are futile, SSOScan records candidate information before a click happens (line 17), and if any candidate list computed for subsequent clicks is exactly the same as previously recorded, SSOScan considers previous click(s) as futile and fast forward to the next candidate. Similarly, SSOScan stores candidate information when the current state is fully explored, i.e. all candidates on the current page have been clicked or the num-

ber of attempts have reached a certain threshold, and if any future click lands on a page which matches the same set of candidates, SSOScan immediately moves on to the next candidate to avoid duplicate work.

Triggering Event Handlers. Programatically changing values of option, checkbox and radio elements using JavaScript does not trigger their onChange event handler. However, in practice we have found several websites rely on this event handler to deliver different inputs to the user. Therefore, we explicitly trigger the event handler after modifying element values.

Circumventing Same-Origin Policy. SSOScan needs to iterate through DOM elements while searching for candidates, but it cannot reach elements inside iframes from a different origin. This is because the content scripts run as the principal of the host, and thus their accesses to iframes are subject to the same-origin policy. However, we saw many cases where the target button is in an iframe which originates from the HTTPS domain of tested website. To handle this, we inject content scripts into all iframes that have an HTTPS origin. Excluding HTTP iframes will cause some content to be missed, however, including them generates problems as sites sometimes include iframes from a complete different website which may include SSO/submit buttons. Besides, login and registration requests should be served over secure connections to prevent eavesdropping. Some rare exceptions usually lead to more serious vulnerabilities such as password disclosure, but our prototype implementation does not check for this.