Open University of the Netherlands
faculty of Management, Science & Technology
Bachelor Computer Science

Shepherd: Enabling Large-Scale Scanning of Websites after Social Single Sign-on

Authors:
Jelle Kalkman
Alan Verresen

Chair(wo)man:
prof. dr. Tanja Vos
Supervisors:
dr. ir. Hugo Jonker
Benjamin Krumnow, MSc.

Presentation date:
02-08-2019

July 31, 2019

Course code:
IB9906

Abstract

Session security for web applications should keep users safe from session hijacking and ensure privacy and security in their online lives. A substantial part of their online lives are hidden behind a login field. To study how secure and private their online lives are, it is needed to do this from the same, authenticated, perspective. However, for a large-scale study, this would require us to automate authentication and account creation for a large number of web applications.

This has proven to be a major challenge because web application can roll out countermeasures against automated account creation, valid credentials should be used, the web is very heterogeneous, and several ethical concerns need to be addressed.

We found that we could leverage Single Sign-On, such as signing in with Facebook or Google, to automate authentication for a large amount of websites. At least 6.3 % of the web applications in the daily Alexa top 1M lists offer the option to use Single Sign-On to authenticate and in 56 % of the cases this is sufficient to reach the authenticated state. We extended the Shepherd framework to increase the reach of research on session security by adding modules to automatically detect if a web application offers Social Single Sign-On and, if so, the ability to automatically authenticate for that web application.

With these extensions Shepherd is able to detect Single Sign-On and managed to authenticate in almost all cases where no additional account creation handling is needed.

Contents

Chapter 1

Introduction

Can we steal your identity on the Web? With the integration of the Internet and the World Wide Web into our daily lives, the security of digital services has become more important than ever. Yet, we regularly hear about data breaches and hackers taking over the accounts of users, so how common and widespread are the vulnerabilities at the basis of these problems?

Security vulnerabilities are everywhere, as demonstrated by newspaper articles [1], vulnerability databases containing disclosed vulnerabilities [2], and yearly reports published by companies that are specialized in tracking these vulnerabilities down. Piessens [3] points out that security vulnerabilities can be introduced at any stage of the software development life cycle and survive due to a lack of development practices aimed at preventing, detecting, or mitigating these security vulnerabilities. Even if a flaw is discovered, developers might be unable or unwilling to address it [4], the security patch trying to fix the flaw might be insufficient [5] or late, or users may never install security patches in the first place [6].

Session management vulnerabilities are an example of common security vulnerabilities found in web applications. In 2014, a session management vulnerability was discovered at wordpress.com [7], a widely used WordPress hosting provider. As a result of this security issue, session hijacking attacks were possible whereby an attacker could identify and authenticate themselves as another user and gain full control over a wordpress.com account. Yahoo! also suffered from similar security breaches in 2013 and 2014, caused by a session management vulnerability, which potentially affected 3 billion users [8, 9]. A session management vulnerability also played a role in the largest security breach to date at Facebook [10], which might have affected 50 million users in 2017 and further illustrates how widespread and persistent these security issues can be.

To study online security and privacy, it is often necessary to investigate this from the perspective of an authenticated user for a large number of web applications. When conducting such large-scale experiments, any form of manual interaction quickly becomes infeasible. Zhou and Evans [11] have demonstrated that authentication processes can be automated by leveraging Social Single Sign-On, SSSO. Ghasemisharif et al. [12] have found that more than 6.3 % (lower bound) of the domains in the Alexa top million ranking let users authenticate via SSSO, so we argue that automating authentication processes through the use of SSSO can make large-scale experiments from the perspective of an authenticated user possible.

Mundada, Feamster, and Krishnamurthy [13] and Takamatsu, Kosuga, and Kono [14] describe automated methods for detecting session management vulnerabilities, but were limited in scale. Zhou and Evans [11] used SSSO to automate the process of authentication, but focused on detecting vulnerabilities related to the integration of SSSO for a large number of websites rather than using it as a stepping stone for further research. As such, there are no existing large-scale studies from the perspective of an authenticated user that have tried to answer the question of how common and widespread session management vulnerabilities are.

1.1 Aim, Scope, and Contributions of Study

Our research aims to make large-scale vulnerability scans possible from the perspective of authenticated users by using Social Single Sign-On, so that we can investigate how common and widespread session management vulnerabilities in web applications are. The contributions of our study are:

- a method to find entry points used to initiate authentication via a known SSSO provider
- a method for authenticating a user via a known SSSO provider in an automated way
- a modular platform that can be used to conduct the envisioned large-scale studies
- implementations of the methods listed above, to be used for the developed platform

Although the Shepherd Project has already investigated the problem of session hijacking from the perspective of authenticated users in Jonker et al. [15], the amount of domains that were examined was limited by the availability of valid credentials.

1.2 Context of Study

This study was done within the context of a Bachelor graduation project at the Open University of the Netherlands. Each student spent approximately 400 hours on this graduation project, of which a considerable amount of hours were spent on individual research topics to prove our competence as researchers.

Our study follows up on the Bachelor thesis 'Counting Sheep' by M. Sleegers, and contributes to the Shepherd Project headed by dr. ir. H.L. Jonker and M.Sc.B. Krumnow. We were asked to increase the reach of the web security vulnerabilities research in the Shepherd project by making use of Social Single Sign-On.

1.3 Overview of Thesis

After a brief overview of all required background knowledge, and a discussion of related work, we first present the Shepherd framework. The development of this software platform was an important first step towards making further research possible.

Next, we will present our individual research, which is focused on harnessing SSSO for automating user authentication. Identifying the entry points to initiate the authentication process was investigated by Alan Verresen, and is presented in the chapter 'Automated Detection of Social Single Sign-On'. The automation of the authentication process was investigated by Jelle Kalkman, who presents his research in the chapter 'Automated Authentication via Social Single Sign-On'.

We base our conclusions on extensive testing of our modules and provide an overview of future work.

Chapter 2

Background

This chapter outlines the fundamental background knowledge which the rest of this thesis further builds upon. Central to our research are web session management, and the workings of Social Single Sign-On, SSSO. Sections on web session security and web session hijacking were added in order to understand the large-scale vulnerability scan presented at the end of this thesis. Finally, a brief introduction to web crawling is provided, as a web crawler is used to search web pages for entry points to initiate the authentication process.

2.1 Web Session Management

Since the inception of the Web in 1990, web browsers and web servers have used the Hypertext Transfer Protocol, HTTP, to communicate with each other. Originally, HTTP was a simple request-response protocol intended to let web servers serve static documents to web browsers, and was inherently stateless, meaning that no state was to be maintained on the server's side between subsequent requests. However, as the Web grew in both size and complexity, the HTTP protocol was altered and extended, so that stateful applications could be built on top of the HTTP protocol, by adding a state management mechanism in the form of HTTP cookies.

Nowadays, many web applications are complex systems where many HTTP messages are sent back and forth between the browser and the server. A sequence of HTTP requests and responses associated with the same user is called a web session. Because the HTTP protocol itself is stateless, the browser has to pass on a session identifier, or session ID, with every request to the web server, so that the server knows which session the message belongs to, and by extension, which user has sent the request.

1. The browser sends an HTTP request to the server without a session ID, because the user hasn't been assigned a session identifier yet.

2. The server assigns a new session ID upon receiving a request from an unidentified user, and sends it back with the next HTTP response.

3. The browser sends a session ID with every subsequent request so that the server can identify the user sending the request.

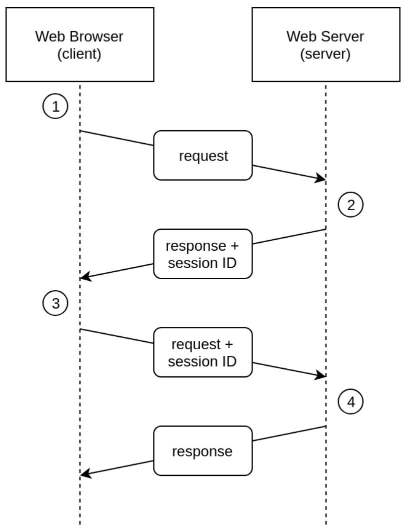


Figure 2.1: A diagram showing a session ID exchange.

Example 2.1
A list of attributes of the session cookie that was stored by the browser after visiting 'https://www.icann.org/' for the first time.

```
name: _new_icann_session
value: "09a6f91d6"
domain: "www.icann.org"
expires: "session"
path: "/"
hostOnly: true
httpOnly: true
secure: true
sameSite: "unset"
creationTime:"Sat, 13 Apr 2019 15:05:59 GMT"
lastAccessed:"Sat, 13 Apr 2019 15:05:59 GMT"
```

Example 2.2
Header of a HTTP response that sets a new session cookie.

```
Set-Cookie: _new_icann_session=09a6f91d6; path=/; secure; HttpOnly
```

4. The web server uses the session ID to associate the request with a user, processes the request, and sends a response back.

Nowadays, HTTP cookies have become the standard mechanism for session ID exchange. Cookies that are used to store the session ID are commonly referred to as session cookies. OWASP [16] lists other mechanisms that can be used to store and pass along session IDs to the server, but argues that cookies are the only acceptable mechanism for session ID exchange, because cookies provide the necessary capabilities for granular control that are needed to protect the session ID, while other mechanisms do not.

The server passes cookies to the browser by adding one or more Set-Cookie headers to an HTTP response. This header needs to contain the name and value of the cookie, but can also set additional attributes for granular control over when the cookie is sent back, and how it can be accessed.

When the browser sends an HTTP request to the server, it can use the Cookie header to send along the cookies that were set by the server. However, a cookie may not be sent back under certain circumstances, depending on its attributes, which we will discuss in the next section.

More information about the HTTP protocol can be found in RFC2616 [17] and its many updates. The workings of HTTP cookies are described in-depth in RFC6265 [18].

2.2 Web Session Hijacking

If a web application fails to protect the session ID stored in a cookie then it could be vulnerable to session hijacking. A successful session hijack attack can result in an attacker taking over the

Example 2.3
Header of a HTTP request that passes the session cookie to the server.

```
Cookie: _new_icann_session=09a6f91d6
```

session and an attacker impersonating a valid user. This is commonly accomplished by access to the session ID by an attacker [19]. There are various well known methods for executing a session hijacking attack [20, 21, 22, 23], such as cross-site scripting, eavesdropping and session fixation.

Cross-Site Scripting In a cross-site scripting attack (XSS), a malicious script is run by the browser that has been injected by the attacker. The malicious script can gain access to the session ID of a badly configured session cookie and disclose it to the attacker [24].

Eavesdropping In an eavesdropping attack an attempt is made to intercept a session ID. This is possible when the cookie in an HTTP message is not sent over a secure connection using the HTTPS protocol, or when an attacker successfully sets up a Man-in-the-Middle attack. Even if HTTPS is used by a web application, a user's browser might be tricked into sending an unencrypted HTTP request with the session cookies [25].

Session Fixation If the server does not assigns a new session ID to the user after successful authentication, then the web application could be vulnerable to session fixation attacks. An attacker gains access to the session ID by XSS or altering intercepted HTTP messages when the user has not authenticated yet. After the user successfully authenticates and is not assigned a new session ID, then the attacker can continue the session and is authenticated [26].

2.3 Web Session Security

The security of an application's session management is of utmost importance. If an attacker is able to obtain the session ID of another user, the attacker can impersonate that user. This attack is commonly referred to as session hijacking.

There are several ways in which session management is inherently vulnerable to session hijacking, regardless of the underlying mechanism. First of all, an attacker should not be able to predict or guess the session ID of another user. Second, an application should handle session IDs with care, and make sure that their value is not leaked anywhere, such as server logs, or when transmitted over communication channels. Third, the server-side life cycle of a session ID must be correctly implemented. When a user is identified as a known user after successful authentication, a new session ID should be assigned to this user, and the old session ID should be invalidated. Session IDs should expire after a properly assessed period of time, and if a user terminates the session, the session ID should be invalidated as well.

In addition, because web session management is usually implemented by using HTTP cookies, badly configured session cookies are also a potential vulnerability. Session cookies must be secured properly to avoid leakage, which can be done by setting their security-related attributes correctly. The settings for cookie attributes are requested by the server to the client in the HTTP response, this is a request, it is not something the server can enforce a client to do [16].

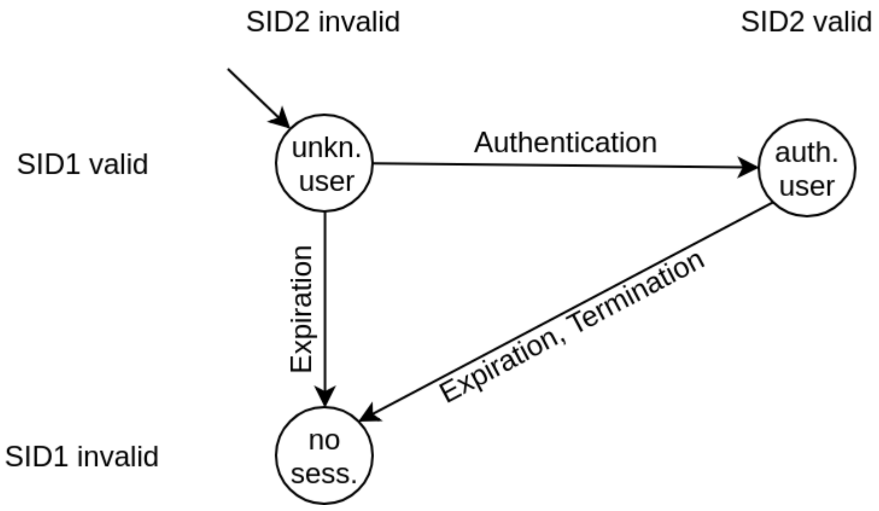


Figure 2.2: Server-side state of session IDs during a typical session of a secure application. SID1 is assigned when the user first accesses the application, and SID2 is assigned after successful authentication.

The secure attribute This attribute instructs the browser to only send the cookie when HTTPS is used. HTTPS is an extension of the HTTP protocol that ensures encryption of the application layer of the TCP/IP model. If this attribute is set for a session cookie, the session ID is not exposed when an HTTP message is intercepted, because, either the HTTP message is encrypted, or the session ID was not passed along with the HTTP message in the first place.

The HttpOnly attribute This attribute restricts access to a cookie. A client is only allowed to send the cookie with an HTTP request and the value of this cookie can only be changed by HTTP communication. Thus when this attribute is set, a client side script, for example JavaScript, can not access the cookie. This attribute can be used to protect the session ID against malicious JavaScript.

Domain and Path attributes These attributes are used to further control when a cookie should be passed along with an HTTP request, based on the URL of the HTTP request. The domain of the URL must match the domain attribute of the cookie [18] and the path of the URL must match the path attribute of the cookie [18]. By setting these attributes, the scope of session cookies can be limited to parts of a website where they are needed.

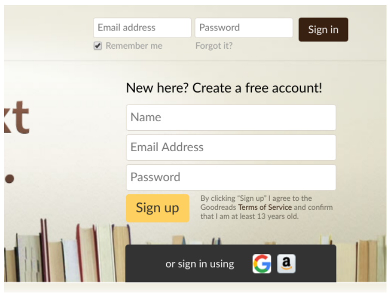
Expire and Max-Age attributes These attributes are used to control the client-side lifespan of a cookie. By default a cookie is deleted by a browser when the browser instance is closed. Cookies set with one of these attributes can be considered persistent cookies and will not be removed when the browser instance closes. When the expires attribute is set to a specific time and date the cookie will be removed at that moment. When the max-age attribute is set with a life span, the cookie will be removed when the lifespan expires. By setting one of these attributes for a session cookie, a session can stay intact when the browser has been closed, and thus the user does not have to be re-authenticated when the browser is re-opened. On the other hand, the session cookie can be made inaccessible on the client-side upon closing the browser.

SameSite attribute By default a browser sends all cookies with an HTTP request. Cross-origin requests are used for example to load images or use the API from another website. For these type or requests it is not always desirable to send specific cookies. This attribute is not widely used yet, but can be crucial in preventing Cross-Site Forgery Requests. If this attribute is set, the cookie is not sent along with cross-origin requests.

2.4 Social Single Sign-On

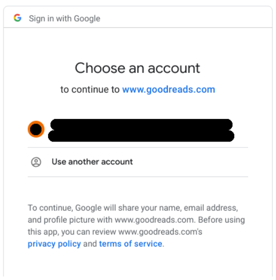
Social Single Sign-On (SSSO) authentication (also called Social Login, Social Authentication, or Social Sign-In) is an authentication method where social media platforms, such as Facebook, Google or Twitter, are used by other systems to authenticate users.

The term 'Single Sign-On' indicates that users have to provide security credentials only once in order to gain access to multiple independent systems. Instead of using different credentials for every system, an Identity Provider (IdP) is used to coordinate the identity of a user between the different systems, called Service Providers (SPs), or relying parties. When a user's identity needs to be authenticated by an SP, the IdP provides the user with an authentication token that can be used to authenticate their identity. In the case of Social Single Sign-On authentication, social media platforms act as an IdP, and let their users rely on them for authentication by any SP that supports them (Figures 2.3, 2.4 and 2.5).



By clicking "Sign up" I agree to the Goodreads Terms of Service and confirm that I am at least 13 years old.

Figure 2.3: A website that allows users to authenticate via Google or Amazon (see bottom).



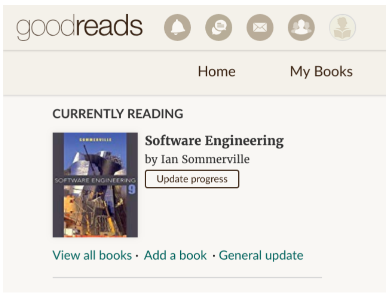
Choose an account

to continue to www.goodreads.com

[redacted]

To continue, Google will share your name, email address, and profile picture with www.goodreads.com. Before using this app, you can review www.goodreads.com's privacy policy and terms of service.

Figure 2.4: User is prompted with which Google account they would like to authenticate.



CURRENTLY READING

Software Engineering
by Ian Sommerville

Figure 2.5: After successful authentication, the user can access their account on the website.

Usage of SSO authentication is preferred over username and password credentials by users [27, 28]. One reason is that SSO provides a solution to password fatigue caused by the large amount of password required on a daily basis [29]. It also avoids bad password habits [30] and should provide more secure passwords in theory [30].

Users only need to remember the security credentials to sign into the IdP, and do not need security credentials for every SP they would like to access. However, the use of SSO authentication also introduces several concerns. First of all, if the IdP is unavailable, or the user forgets their security credentials, the user cannot be authenticated, and thus also loses access to all their other accounts. If a user's access to an IdP is somehow compromised, attackers can access any SP impersonating as the user. Such compromises can have cascading and long lasting effects [12].

Several existing protocols, such as OpenId or OAuth, can be used for implementing SSSO authentication, but Mainka et al. [31] have observed that the sequence of steps that all of these protocols go through, can be generalized, as depicted in figure 2.6.

1. The user initiates the SP's authentication process, indicating that they would like to rely on the IdP for authentication.

2. If necessary, the SP contacts the IdP to gather information on how to authenticate the user, how to verify subsequent messages, or where to redirect the user to for retrieving

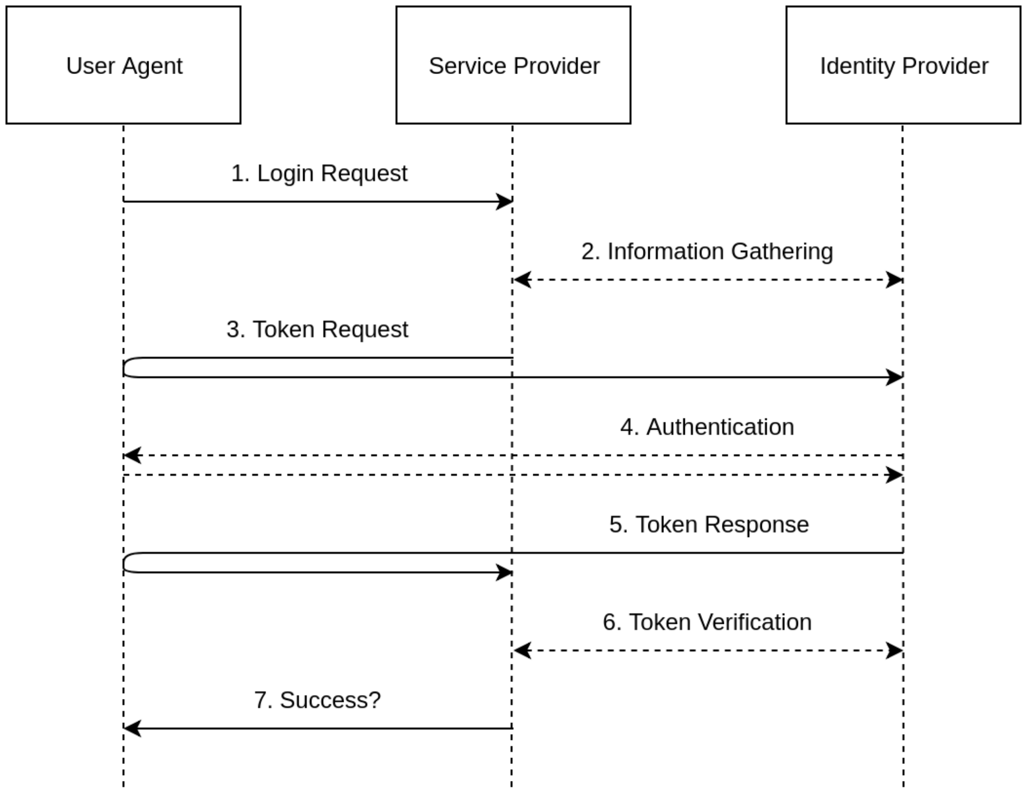


Figure 2.6: The general SSSO authentication flow [31]

their authentication token.

3. The SP requests the user to provide an authentication token that they can obtain from the IdP, and redirects them to the IdP.
4. If the user has not been authenticated yet, the user needs to be authenticated by the IdP.
5. The IdP passes the authentication token on to the user, and redirects the user back to the SP, so that the user can pass on the authentication token to the SP.
6. If necessary, the SP can verify the authentication token by contacting the IdP.
7. If nothing went wrong, the user has now been authenticated by the SP.

2.5 Web Crawling

Web crawlers (also called spiders, or robots) are programs that explore the Web with the purpose of extracting information from web pages. Web crawlers have many applications, but all generally work the same way.

Every web crawl session, depicted in figure 2.5, starts with a non-empty queue containing one or multiple seed URLs that will be crawled first. Until the queue of URLs is empty, or a certain condition is met, the following steps are repeated:

1. the next URL is popped from the front of the queue of URLs
2. document at the URL is downloaded
3. new URLs are extracted from the document
4. the document is stored for later processing

In order to avoid redundant work, a crawl history needs to be maintained to remember which URLs are currently queued up, or have been visited in the past. Even though this basic approach of a web crawler is the same for all web crawlers, many variations exist that uniquely modify their behaviour.

The first distinction to be made is how the next URL is decided. Most web crawlers used by search engines tend to prefer a breadth-first strategy. Breadth-first strategies visit directly referenced pages first, which tend to be of much higher quality than pages that are several jumps away. Another common strategy employed by preferential web crawlers is using a priority queue to queue the URLs, and assign a priority to every URL that was found during the URL extraction phase.

A second major distinction is the scope of web crawlers. Universal web crawlers try to reach as many web pages as possible, such as the web crawlers employed by search engines which try to index every web page according to their content. On the other hand, focused web crawlers attempt to keep their exploration limited to pages of interest, such as pages that are related to sport or commerce, or pages that have certain functionality.

Web crawlers face many different challenges. Not only do they have to correctly fetch, parse, and store web pages in a scalable way, but they also have to extract new URLs from these pages to drive their search. Older web pages used to be static documents containing references to other pages as simple hyperlinks, which were easy to extract. But nowadays modern web applications may use JavaScript to dynamically construct web pages, and can hide references to other pages server-side or deeply hidden in page scripts.

A crawler also can get stuck in the deep web, a term reserved for web pages that are dynamically constructed based on prior interactions of a user with a website. This situation is common on e-commerce websites, where every product page contains unique URLs to even more product pages. These pages are aptly called spider traps, and every web crawler should have a strategy to deal with them, such as limiting the search depth (the length of the path between a seeded page, and the accessed page), limiting the amount of pages per domain that may be inspected, or inspecting the URL itself for potential trouble before adding it to the queue.

Another aspect that should be considered is web crawler etiquette. Crawlers can put a lot of strain on the servers serving the content that they are exploring. Some websites do not wish to be crawled, and thus provide a robots.txt file in the root directory to let conforming crawlers know which web pages may be crawled.

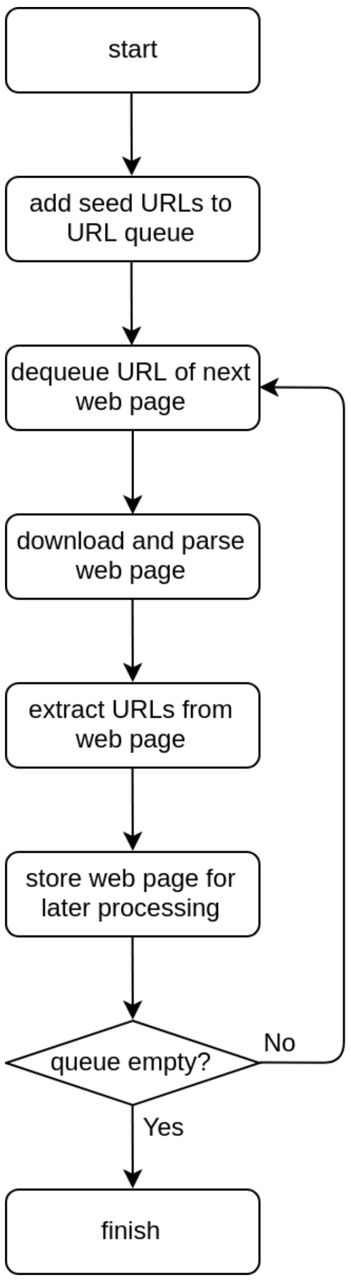

start
add seed URLs to URL queue
dequeue URL of next web page
download and parse web page
extract URLs from web page
store web page for later processing
queue empty?
No
Yes
finish

Figure 2.7: A diagram of the general workflow of a web crawler.

Example 2.4
This ’robots.txt’ file lists instructions for crawler bots. The first set of lines allows ’Twitterbot’ to crawl the website, with the exception of web pages that path-match ’/search’ or ’/admin’. The second set of lines disallows any other crawler from crawling any part of the website.

```
User-agent: Twitterbot
Crawl-delay: 120
Disallow: /search
Disallow: /admin

User-agent: *
Disallow: /
```

2.6 Browser Automation Tools

The need for automated verification of client-side behaviour of web applications in software testing has led to the development of browser automation tools. Browser automation tools are used to automate a sequence of browser interactions, effectively emulating a user’s behaviour without human interaction, which makes them an ideal tool for automated software testing.

Browser automation tools tend to use one of two popular approaches. A first popular approach lets the tester record a sequence of browser interactions, which can then be replayed in the same way during tests. This allows the tester to add new tests in a fast way, usually without having to write any code. However, to change the test, the tester needs to record a new sequence of interactions from start to end. Another potential problem with this approach is that, depending on the implementation, these tests might break when the UI changes, and thus tests using this approach can be rather fragile. The other popular approach is the use of scripts for browser automation. This approach has the advantage that a tester only has to change relevant parts of the test, and these tests tend to be less fragile.

Although browser automation tools were originally only used for software testing, other uses have been found for browser automation tools, including web crawling. The use of browser automation tools for web crawling has several advantages over other tools: first and foremost, they are commonly used to evaluate JavaScript, which is necessary to render modern web applications. This also helps with avoiding countermeasures. The only real disadvantage to using a browser automation tool for web crawling is the amount of resources used. Another disadvantage of using a browser automation tool for web crawling is that their interfaces are designed with software testing in mind, resulting in the omission of certain functionality that would be convenient for web crawling.

Selenium is a popular script-based browser automation tool for software testing that can be used to automate multiple types of browsers, and is available for multiple programming languages. Selenium uses webdrivers, browser-specific programs that execute interactions with a browser. Selenium’s development has inspired the creation of the WebDriver protocol, which has been considered to be the de facto standard for webdrivers since long. Because all webdrivers implement the same interface according to the WebDriver protocol, any browser can be automated as long as a conforming webdriver is available for that browser. In reality, however, differences between browsers and their respective webdrivers still forces testers to write a lot of browser-specific code.

Browser automation is achieved by using interprocess communication, in the following way:

- application uses selenium to send requests to webdriver instance
- webdriver instance sends requests to browser instance

- browser instance processes requests and sends responses with feedback to webdriver
- webdriver instance receives response from browser instance
- application uses selenium to receive response from webdriver instance

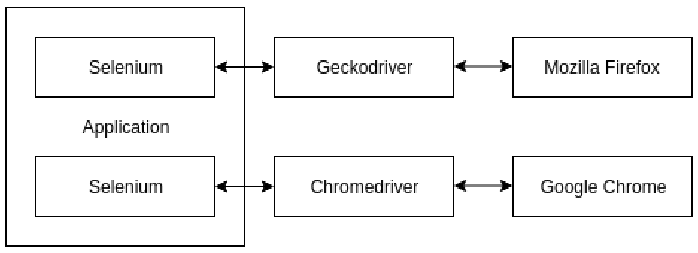


Figure 2.8: Data flow in system using Selenium to automate several popular browsers.

Recently, other browser automation tools besides Selenium have been gaining popularity, where the focus has shifted away from software testing, and more towards browser automation itself. Instead of supporting multiple browsers with an interface designed for software testing in mind, these browser automation tools support only one single browser with a broad and advanced interface. An example of this is Puppeteer, a JavaScript-based tool for Google Chrome.

Another recent change has been the integration of browser automation endpoints in the browsers themselves, which can be communicated with browser-specific protocols, and thus removing the need for a mediator such as a webdriver. As these things tend to go, popular browser-specific protocols are later standardized and adopted by other browsers as well. This is the case for Google's Chrome DevTools Protocol, which is now also supported by Mozilla's Firefox browser.

Another fairly recent trend is the adoption of a headless mode by most popular browsers, coinciding with the rise in popularity of the term 'headless browser' over browser automation tool. When using a headless browser, the graphical user interface is not rendered, resulting in less resources being used.

2.7 Summary

In this chapter we provided fundamental knowledge that is required to understand the rest of this thesis. Our research is centered around web session management, and authentication through SSSO. However, this research would not have been possible without writing a web crawler to explore the web. This background is also necessary for the next chapter on related work, in which we explore the frontiers of related research.

Chapter 3

Related Work

Our study touches upon several areas of active research. We will discuss the most relevant studies in those areas of research and their contributions, in order to further shape the context of our study. We will do this by showing how our study is different from other studies, clarify how our study is new and original. Lastly, we would also like to discuss the Shepherd Project, a study investigating session management vulnerabilities, which our study directly contributes to.

3.1 SSO Security

The first area of research that is related to our study is the security of Single Sign-On (SSO). This area of research features a wide range of studies, ranging from theoretical analysis to large-scale real-world experiments.

A field study on popular Web SSO systems by Wang, Chen, and Wang [32] identified several session hijacking vulnerabilities in the implementation of SSO by identity providers and service providers. They manually logged into several websites, while intercepting web traffic between the web browser and web server. A tool was used to intercept and analyze HTTP requests and responses related to SSO. Intercepted HTTP requests were then modified, and sent back to the web server, after which the HTTP responses that were sent back by the web server were inspected for inconsistencies, indicating vulnerabilities.

The same approach of interception of and analyzing web traffic between the web browser and web servers was used by Li and Mitchell [33] in 2014. They probed implementations of Web SSO based on the OAuth 2.0 protocol for logic flaws and cross-site request forgery (CSRF) vulnerabilities, which allowed for session hijacking to happen. Their study investigated 60 Chinese relying parties, covering 10 different Chinese identity providers. They concluded that 21 of the 60 relying parties were vulnerable. They also tried to identify a potential reason as to why these vulnerabilities existed, and determined that the OAuth 2.0 protocol was unclear and lacked information.

A similar study was conducted again by Li and Mitchell [34] in 2016. This time they investigated all domains supporting Google's OpenID Connect in the GTMetrix Top 1000 sites, 103 in total for similar flaws, with a similar approach. Multiple flaws in the implementation of OpenID Connect were detected via black-box testing. These flaws could allow an attacker access to the victim's account using XSS or CSRF. They also identified potential causes for the implementation flaws and provided recommendations to the identity providers and relying parties.

These studies are only some of many similar studies that were performed on a relatively small scale, or needed manual intervention to handle the authentication process. Our study affects this area of research by allowing these studies to be conducted on a large-scale, without any manual intervention.

3.2 SSO Detection

A small number of existing studies related to SSO security are of particular interest to our study, because they already investigated the automated detection of SSO in some way or another, which we will discuss here. So, although they technically belong in the previous section, we would like to emphasize the effort of automated SSO detection that was done by these studies.

EsPReSSO is a penetration testing tool built on top of Burpsuite, a popular penetration testing tool, by Mainka et al. [31]. This tool intercepts web traffic in order to automatically detect and analyze SSO security. To detect if, and identify which, SSO was being used, they established a method for fingerprinting several SSO protocols. These fingerprints were then used to process intercepted web traffic in an automated way. If the use of SSO was recognized, then the security of SSO was analyzed through the manipulation of HTTP messages, in a way that is similar to the aforementioned studies of the previous section. The results were then presented in a user-friendly way for further investigation.

Ghasemisharif et al. [12] explored the potential consequences of an attacker taking over an account that can be used with SSO, and highlights the lack of countermeasures foreseen by most protocols to reduce harm in this situation. They discuss multiple ways in which an attacker can hijack a session, which then allows the attacker to gain access to, or even create, accounts for other services. They also explored several ways in which an attacker can retain control over compromised accounts, and further showed how attackers can abuse hijacked accounts of relying parties due to incorrect integration, even after revocation of access rights. These attacks are hard or even impossible to recover from in most cases, especially due to the invisibility and lack of oversight when such an attack happens. For this reason, they proposed to extend the OpenID Connect protocol with 'Single Sign-Off', a feature that revokes access to all accounts of relying parties, and thus prevents attackers from accessing those other services. Their study also includes a practical component where they built a web crawler to determine the prevalence of SSO on the Web. They searched for login and registration pages by analyzing the landing page for potential references to such pages. If no pages were initially found, the crawler would also try to guess the links to these pages by using common link patterns, such as the URL path `/login/`. If a login or registration page was found, their crawler would try to infer the use of SSO by using regular expressions and searching for references to known SSO API endpoints. With this crawler they analyzed the Alexa top 1 million ranking and determined that a lower bound of 6.3 % of the domains has integrated some form of SSO, with Facebook being the most commonly supported SSO provider, followed by Google and Twitter.

Zhou and Evans [11] investigated security vulnerabilities related to the integration of Facebook's SSO by websites in 2014. For this purpose, they built a web crawler that tried to detect the integration of Facebook's SSO by a website, and if this was the case, would attempt to log in using Facebook's SSO. Web traffic was intercepted to determine whether or not Facebook SSO was actually being used, and was also analyzed to determine the presence of security vulnerabilities. They probed 20.000 of the most popular websites in the US for several vulnerabilities caused by improperly integrating Facebook's SSO. 1660 of these websites were found to have integrated Facebook's SSO, and of those websites that had integrated Facebook's SSO, 20 % were found to be vulnerable to one or multiple vulnerabilities. This study was the first of its kind, in the sense that they were the first to largely automate the process of detecting SSO, attempting to log in, and verifying the logged in state. In a sense, they have set the stage for our study. Their paper describes in an elaborate way how they detected Facebook's SSO, how they performed the login attempt, and verified a successful login attempt. An additional step in this process is further account creation after successfully logging in, which was still necessary in 44 % of the cases. Their study revealed a lot of the complexities that make the automation of this such a complicated problem. They managed to successfully log into 80 % of the sites that supported Facebook SSSO. Although the lack of support for non-English websites and only detecting Facebook's SSO limits SSOScan, their methods can be extended to detect and use multiple identity providers.

These studies are highly relevant, because automatically detecting the integration of SSO, and using SSO for authenticating to websites are the main focus points of our study. We will contribute to this area of research by evaluating and exploring the methods used by these studies in greater detail, proposing several improvements, so that future studies can benefit.

3.3 The Shepherd Project

Our research is a direct contribution to the Shepherd Project, a tool used to automatically scan a large amount of websites for session management vulnerabilities. The scanner built in this project was used in February 2018 to scan 50,000 domains and was able to automatically login and verify the login attempt in 12.4 %, 6,273, of the domains. From the domains from which Shepherd managed to log in and verify the log in, 41,4 % was found vulnerable to session hijacking. Shepherd is designed to work in three distinctive phases: the targeting phase, the log in phase, and the scanning phase.

The targeting phase can be run stand alone prior to the actual scanning and comprises the selection of the targets and an attempt to fetch credentials. To authenticate Shepherd as a valid user, Shepherd tries to source credentials for each given domain from BugMeNot, a service where users can share credentials. When using this service, the validity of the credentials is a limiting factor. However no legal alternatives are available.

If any credentials are found, Shepherd searches the page with the login elements and attempts to authenticate with the sourced credentials in the next phase. If a successful, Shepherd determines how to verify the logged in state, which is important because this result is used in determining the susceptibility to session hijacking. To determine this susceptibility, Shepherd first needs to determine the set of cookies containing the authorization token. If that succeeds Shepherd can inspect if all the cookies containing the authorization token are well protected. It is only needed to determine if the secure flag is set for these cookies.

To assess the accuracy and to reveal shortcomings in the scanners heuristics manual evaluations were performed. Detection of the login page/elements is reasonable. Authentication failures were mostly caused by invalid credentials or a misinterpretation of the web page. Any improvements made in the verification of the authenticated state could contribute to the scanners performance. Most importantly, the confidence is high for the authentication and verification.

In February 2018 a first trial attempt was performed to also log in using Facebook SSSO to gain some first hand experience with automatically logging in using SSSO and to determine difficulties and discover pitfalls.

3.4 Summary

In this chapter we discussed studies which were highly relevant to our contributions to the Shepherd project, provide knowledge or methods how to solve expected problems. We suspect our work will have a lot in common with the works of Zhou and Evans [11] and Ghasemisharif et al. [12].

Chapter 4

The Shepherd Framework

4.1 Introduction

Our contributions are built on top of an existing implementation to crawl the web for the login options for a web application. In the Shepherd Project was already decided what programming languages, Python and JavaScript, and which browser automation tool, Selenium, would be used. In this chapter we explain what we needed to do to make our contributions possible, namely the development of the Shepherd framework.

4.2 Problem Description

We decided to create a proof of concept implementation to detect and to authenticate with Facebook SSSO to gain first hand experience with the existing framework and automated scanning. This also served the purpose of identifying and deciding what improvements and changes needed to be made to the existing framework.

We built the proof of concept Facebook SSSO implementation on top of the current version. From this we concluded that:

- The existing implementation was hard to reuse for our intended goals.
- Refactoring the existing implementation enables us to use the existing tool as a basis for our intended goals.

We managed to create a proof of concept scanner which was used in the Shepherd Project with limited success. However, one of our intended goals is to support multiple SSSO providers and we did not see any option how this could be logically done with the current implementation. We proposed to spend time on refactoring the current version to create reusable core classes and to develop an architecture that is suitable for extension.

4.3 Shepherd's Architecture

Because we envisioned a more extensible and logical architecture we decided to use the following main architecture consisting of the following components: apps, core, modules and settings.

Core Functionality

The core functionality is placed in this package because this will be widely used in multiple modules and to provide a stable interface.

Browser The main goal of this module is to provide a Pythonic interface for managing and interacting with a browser instance provided by Selenium. This provides an independent interface for any browser and if changes need to be made because of changes in a browser or a webdriver, it is done within this module. Because the browser classes directly interact with the browser instances, they also provide the error handling for this.

This module also contains a Pythonic class which provides a wrapper around an HTML element and handles the interaction and errors with such an element.

Other functionality in this module is for example code for handling the loading of pages and the loading of browser extensions.

Framework Classes in this module provide base classes for our core concepts apps and modules which we will explain separately and code to handle multiple worker threads simultaneously. This makes it possible to process multiple domain simultaneously, each in its own thread with and its own browser instance.

Resources, Utils and Scanner These modules contain utility classes for example to fetch data from the Alexa ranking, base class implementations to queue pages that need to be scanned and many others.

4.3.1 Apps and Modules

We needed to develop an architecture where it was possible to create multiple and run multiple configurations and that it is possible to easily extend these with new functionality. This resulted in three core concepts: websites, modules and apps.

Website A Website object in Shepherd is an object that is created to store all properties and results from modules for the current domain that is processed. A Website object is passed from module to module and each module that receives this object can retrieve and store results for and from other modules through a well defined interface.

Modules A Module in Shepherd represents one defined feature or task. For example a module to detect SSSO on a website or a module that is only concerned with the verification of the authenticated state.

A module receives a Website object from which it can retrieve information and results from a previously run module. If other modules depend on the results of the current module, these are stored in the Website object. This creates the possibility to run multiple configurations and to create reusable components.

Apps An App actually defines what configuration to run. In an App subclasses are used from Worker and Leader classes from the framework. The Leader base class manages workers in multiple threads for processing a collection of domains. A Worker class actually runs a chain of modules for a given domain.

4.4 Stable Web Pages

When we created the proof of concept Facebook SSSO scanner with the initial version of Shepherd, we observed that fixed time outs were used after interaction with the browser instance. For example, when a link is clicked and a page is loaded the time out for a page to load could only be set to a specified time. This does not take into account if a page has actually loaded. We see two problems with this approach. First it could be that interaction with the browser instance is resumed however a page has not loaded completely. This can affect the stability and results. Second, waiting is done while nothing happens.

Stability Within Selenium there is no support to determine if a page has stabilized, meaning that no changes are expected on a short notice. Our solution was to monitor the evolution of the amount of DOM elements that are loaded and the amount of outstanding HTTP responses on a dynamic basis. For this all open tabs from the browser instance need to be evaluated.

The Algorithm On an interval basis an algorithm runs to determine if the amount of elements in the DOM did not changed in a short period of time and that the amount of outstanding HTTP responses did not changed over a long period of time. The interval is continually adjusted based on the results. When a lot of changes in the amount of DOM elements were found, then the interval time increases, and when the amount of outstanding HTTP responses is stable over a long time, the interval time is reduced.

ShepherdRequestTracker Because no support exists in Selenium to keep track of outstanding HTTP responses, it was needed to create a browser extension for this, the ShepherdRequestTracker. This browser extension keeps track of how many HTTP requests are waiting for a response. Unfortunately its use is limited to Chrome only.

Performance Impact Although this has some effect on the run time, the performance gain as a result of a more stable platform should also be taken into consideration.

4.5 Summary

In this chapter we discussed only a limited number of improvements we had to apply to Shepherd to make it an actual framework. We were convinced that these were actually needed to do to make our intended goals possible. We explained our design choices and highlighted how we solved the stability problem. This resulted in a modular and extensible architecture which moved away from a largely functional programming paradigm to a more object oriented design with the use of utility functions. We are convinced that this work is a substantial contribution to the Shepherd Project.

Chapter 5

Automated Detection of Social Single Sign-On Authentication

Using Social Single Sign-On (SSSO) to automate authentication for a large amount of websites has been shown to be a promising endeavour by prior research. Zhou and Evans [11] have demonstrated that it is possible to automate SSSO authentication processes, and Ghasemisharif et al. [12] have found that a significant number of websites let users log in with SSSO. However, before we can automate SSSO authentication processes, we need a reliable way to discover the entry points of these processes.

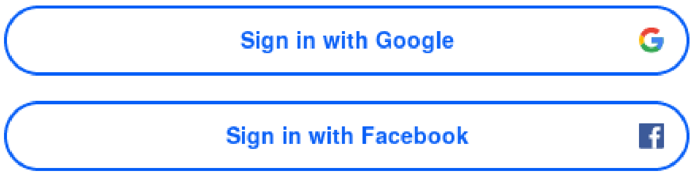


Figure 5.1: (Indeed) An example of the buttons used to log in with SSSO.

As described in background section 2.4, a user typically needs to click a button in order to log in with SSSO. The discovery of such a button is a trivial task for a human user, but is far from trivial for a computer program. In this chapter, we present our research that tries to address the following research question:

How can we discover the entry points of SSSO authentication processes, in a fully automated way, for any website, for any SSSO provider, in an efficient and reliable way?

This research question applies several restrictions upon any possible solutions. *"In a fully automated way"* implies that a solution should not involve any human interactions. *"For any website"* reflects our wish that a solution may only make generic assumptions about the structure and workings of a website, and similarly, *"for any SSSO provider"* suggests that a solution should work for most, if not all, SSSO providers. Last but not least, a solution should work *"in an efficient and reliable way"*, hence a solution should not explore every possibility. Instead, a solution should make sensible decisions based on experimental evidence, and produce reliable results in a relatively fast amount of time with the resources at hand.

5.1 Overview

After confirming that using SSSO for automating authentication is an undertaking that's worth investigating, we will discuss and motivate the general approach of our solution. If a user wants to log in with SSSO, then they typically need to click on an SSSO button, such as the ones shown in figure 5.1. We have found that this is the only consistent way to initiate almost all SSSO authentication processes, and thus our solution revolves around clicking on these SSSO buttons.

We will also discuss the methods that have been used by prior studies to detect whether or not SSSO authentication has been integrated by a web application. We have identified several pitfalls that need to avoided in order to prevent skewing results. We discuss several ways to address these pitfalls, and present a new and reliable method that can be used to verify whether or not an element that we have clicked is an SSSO button.

Before we can click on SSSO buttons, we need a way to discover SSSO buttons on any website. We explore several problems related to the discovery of SSSO buttons that need to be addressed first:

- **Where to look for SSSO buttons?**
  We should only look for SSSO buttons in locations where we expect to find other components related to authentication to be, such as username-password forms on login pages.

- **How to recognize SSSO buttons?**
  We can recognize SSSO buttons by using an approach that is primarily based on the presence of relevant keywords and by using heuristics for filtering/sorting results.

- **How to click on SSSO buttons?**
  We have investigated the sequences of actions that may be required to click on an SSSO button, and present an appropriate data structure to store such sequences.

- **How to find the URLs of relevant web pages?**
  To drive our exploration of a website, we need a way to discover the URLs of pages where we are likely to find SSSO buttons.

After having explored several problems related to the discovery of SSSSO buttons, we then present a method for discovering SSSO buttons as our solution to this chapter's research problem. This method operates on two levels:

- **How to find an SSSO button on a web page?**
  We use a trial-and-error-based approach where we execute sequences of actions in an attempt to discover the next step that might lead to clicking on an SSSO button.

- **How to find an SSSO button on a website?**
  We use a preferential web crawler that prioritizes pages that are deemed more likely to contain an SSSO button.

We end this chapter by scanning a set of websites for SSSO buttons using a web scanner that implements our solution. We analyze its results, and report any existing problems, potential improvements, and future work.

5.2 Prevalence of SSSO On The Web

5.2.1 Problem Description

In 2018, Ghasemisharif et al. [12] conducted an experiment to determine the prevalence of SSSO on the Web, as part of their study investigating account takeovers with SSSO. They scanned all domains of a daily Alexa Global Top 1M list for evidence of SSSO integration, and claimed to have found that at least 57,555 out of 1 million domains hosted a website that would let users log in with SSSO. This led us to believe that a significant amount of websites let users log in with SSSO. However, we grew concerned about skewed results when we evaluated the methods that were used for this experiment, and found a significant amount of false positives in the published data set. This encouraged us to validate their findings by providing our own experimental evidence.

Scheitle et al. [35] scrutinized the way in which 'Internet top lists' have been (mis)used in prior studies. They examined the impact that the use of different Internet top lists can have on research results, and urge to be careful when doing research using these lists. We needed to ensure that we were using these Internet top lists correctly, and that our findings were correct.

5.2.2 Observations

Ghasemisharif et al. [12] created a web scanner which would first try to discover a web page where users could log in, and then would scan that page for evidence of SSSO authentication. While examining the published data set, we found that 37 domains out of the 100 positive results that we checked, were false positives. We observed several systematical flaws in the used methodology that could've potentially skewed the results in a significant way:

- social media buttons, share buttons, etc. were misinterpreted as evidence
- blogs and other similar services which were hosted by a host website (e.g. tumblr.com) were treated as standalone websites, and the authentication systems of the host website were treated as the authentication system of the hosted blog or website
- SSSO authentication used by embedded third party services, such as embedded commenting systems (e.g. disqus.com), was treated as being part of the website itself
- evidence of SSSO was searched for on web pages of unrelated websites

We will also observe in section 5.5 that SSSO buttons are commonly found in modal windows and pop-up menus, which are hidden initially, or sometimes are loaded only after the user has interacted with the page. Our solution needs to properly address these issues to ensure accurate results.

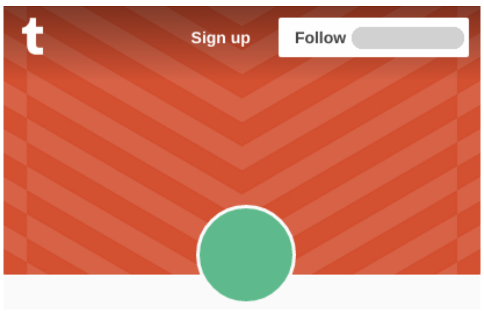
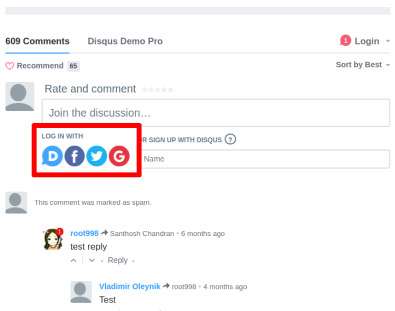


Figure 5.2: (tumblr.com) A 'sign up' button belonging to the blog hosting service is shown at the top of most blogs.



This comment was marked as spam.

root998 ➦ Santhosh Chandran • 6 months ago
test reply
Reply

Vladimir Oleynik ➦ root998 • 4 months ago
Test

Figure 5.3: (disqus.com) A third party embedded comment system with SSSO authentication (marked in red), below an article.

To validate the results of the experiment by Ghasemisharif et al. [12], we manually investigated 2000 web domains sampled from the Alexa Global Top 1M list of July 3 2019 in experiment A.1, on July 4 2019. In addition, we also extended the original experiment by determining whether or not any user could register and log into a website.

Out of 2000 web domains, we found that 593 domains (29.65% ∓ 2.00%, 95% CI) hosted a website with user registration, and that 139 of all websites (6.95% ± 1.11%, 95% CI) had integrated SSSO for at least one SSSO provider. If we were to generalize our findings to the whole population of the daily Alexa Top 1M rankings list, this would mean that we could use SSSO to log in for:

- 6.95% ± 1.11% (95% CI) of all domains
- 23.44% ± 3.41% (95% CI) of websites with user registration

Despite the flaws that we had found in the data sets published by Ghasemisharif et al. [12], our experiment came to similar conclusions. We confirmed the trend that more popular websites are more likely to have integrated SSSO, which is probably related to the trend that more popular websites are more likely to have an authentication system in place, as shown in figures 5.4 and 5.5. We also confirmed the dominance of Facebook as an SSSO provider among websites listed in the Alexa Top 1M list, followed by Google, Twitter, and a long tail of other SSSO providers, as shown in figure 5.6.

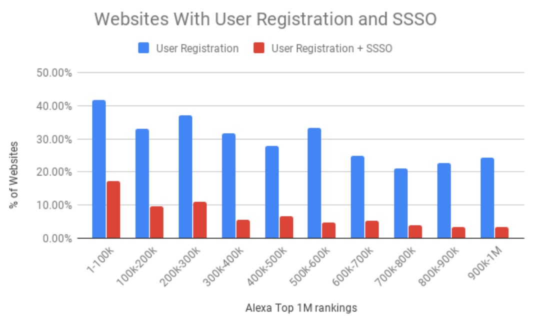


Figure 5.4: Percentage of websites with user registration and websites with SSSO, organized per range of 100,000 domains in top list.

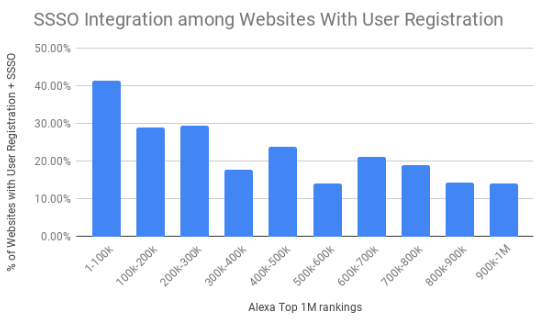


Figure 5.5: Percentage of SSSO integration among websites with user registration, organized per range of 100,000 domains in top list.

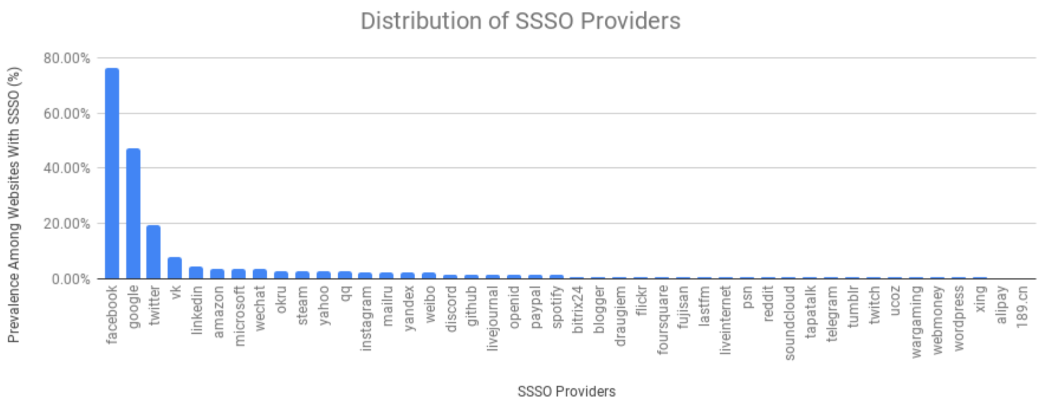


Figure 5.6: Coverage of SSSO providers among websites that have integrated SSSO.

However, we should be careful when drawing inferences about the whole Web from these experiments. Scheitle et al. [35] have investigated the use of top lists in existing research, and found that:

- sources and effects of bias for these top lists are often unclear
- there are large differences in characteristics between different top lists
- list contents may vary immensely based on the date (daily changes, weekday vs. weekend)
- results based on top lists are largely overestimated when compared to the whole Web

Although using a large sample that is representative of the whole Web is recommended by Scheitle et al. [35] to avoid these problems, we have decided to keep using the Alexa Global Top 1M list for our experiments. This top list is biased towards web-based traffic, and deemed a great fit for sampling popular websites, while avoiding domains that are used to host embedded content. SSSO authentication is more prevalent among popular websites, and thus using this top list helps us reduce our research efforts.

We just need to be careful to avoid relying on any properties that are correlated with the popularity of a website. Looking ahead, we can say that we never found any statistically significant correlation between the popularity of a website, and any of the properties that our solution relies upon. As such, we have no reason to expect that the effectiveness of our solution is different for any websites outside of the Alexa Global Top 1M lists.

5.2.3 Conclusions

We validated the results found by Ghasemisharif et al. [12] regarding the prevalence of SSSO. However, we cannot reliably infer any information about the whole Web from these experiments. As such, we can only conclude that:

- a significant amount of websites uses SSSO authentication in absolute terms
- popular websites are more likely to use SSSO authentication

Because we use the Alexa Global Top 1M list for our research, we should avoid relying on a property when there's a statistically significant correlation between that property and a website's popularity.

5.3 Integration of SSSO Authentication

5.3.1 Problem Description

To identify suitable approaches for solving this chapter's research problem, we analyzed the integration of SSSO authentication by web applications. Zhou and Evans [11] already demonstrated that clicking on SSSO buttons is a common entry point for SSSO authentication processes, and that the discovery of SSSO buttons can be automated. However, their research dates back to more than 5 years ago, was limited to a single SSSO provider, and did not make mention of any other viable ways to approach this problem.

5.3.2 Observations

The integration of SSSO authentication is a complicated situation due to the wide variety of existing SSSO providers, SSO protocols, and the many different technologies that can be used to implement SSSO authentication. The only trend that is largely consistent among different web applications is that the user typically needs to click on an SSSO button to log in with SSSO. Everything that happens after clicking on an SSSO button depends on how SSSO authentication has been integrated by the web application.

A large source of diversity among implementations of SSSO authentication is the variety of browser technologies that can be used. The user's web browser is used to relay messages between the SSSO provider and the web application, but there's no preferred mechanism to handle these communications:

- *URL-Based Messaging*
  URLs can be used to store tokens, information about the application, and other information that is used to implement SSSO.

- *Cross-Window Messaging*
  Cross-origin communication with pop-up windows and embedded frames is possible by using the Window.postMessage API.

- *AJAX (Asynchronous Javascript And XML)*
  AJAX is commonly used for direct communication between the web application's client-side scripts and API end-points provided by the SSSO provider.

The use of different browser technologies for messaging goes hand in hand with varying degrees of server-side handling of SSSO authentication. Some web applications have reduced the client-side integration of SSSO to just a button and a URL, and completely handle SSSO authentication on the server-side, while other web applications handle almost everything using client-side scripts.

Another source of diversity among different implementations of SSSO authentication is the large amount of tools that can be used by developers to integrate SSSO authentication:

- standard development kits (SDKs) (e.g. Facebook)
- third-party services (e.g. Auth0)
- third-party libraries (e.g. Twitter)
- developers can write their own implementations

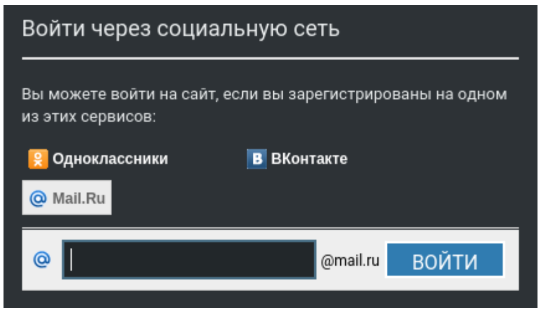
No matter how SSSO authentication is integrated by its developers, its implementation will always be a part of a bigger whole, the web application itself. Therefore, every implementation of SSSO authentication can be considered unique, and thus any approach based on the way that SSSO authentication is implemented, is not a viable approach.

The only option that remains, is to base our approach on clicking on SSSO buttons to initiate SSSO authentication processes. Anything beyond this point is too complex and varied to base a reliable approach upon. Zhou and Evans [11] already used such an approach to successfully automate SSSO authentication processes with Facebook, and there is no apparent reason for why this approach wouldn't also work with other SSSO providers. This led us to the following hypothesis, which serves as the fundamental basis that our solution is based upon.

Hypothesis 5.1
We can reliably initiate SSSO authentication processes by clicking on SSSO buttons.

This hypothesis is supported by experimental evidence collected during experiment A.2. We investigated 139 websites with SSSO authentication, and found a total of 396 different entry points of SSSO authentication processes. Hypothesis 5.1 held true for 382 out of 396 entry points (96.46% $\pm$ 1.82%, 95% CI).

When the hypothesis did not hold true, an implementation based on the OpenId protocol was used, SSSO authentication was implemented using QR codes, or extra preconditions needed to be fulfilled before we could initiate SSSO authentication. However, these situations are not deemed common or impactful enough to really disprove the usefulness of our hypothesis.


Войти через социальную сеть

Вы можете войти на сайт, если вы зарегистрированы на одном из этих сервисов:

Одноклассники

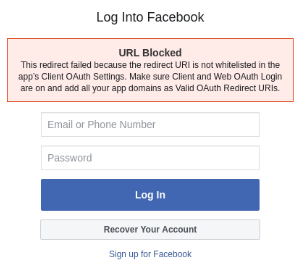
ВКонтакте

Mail.Ru

@mail.ru

ВОЙТИ

Figure 5.7: (darxton.ru) Users need to providing their *mail.ru* email address.


Log Into Facebook

URL Blocked
This redirect failed because the redirect URI is not whitelisted in the app's Client OAuth Settings. Make sure Client and Web OAuth Login are on and add all your app domains as Valid OAuth Redirect URIs.

Email or Phone Number

Password

Log In

Recover Your Account

Sign up for Facebook

Figure 5.8: A warning message is shown to notify users that SSSO authentication is misconfigured on the SSSO provider's end.

A much larger issue was the amount of broken SSSO authentication processes. In 50 out of the remaining 382 cases (13.09% ± 3.38%, 95% CI), we observed one of the following two problems after clicking on an SSSO button:

- in 34 cases, SSSO authentication was badly integrated by the web application
- in 16 cases, SSSO authentication was badly misconfigured on the SSSO provider's end

We found no correlation between the ranking of a website, and the likelihood of an SSSO authentication process being broken. At least one SSSO authentication process was broken for 27 out of 139 websites, and all SSSO authentication processes were broken for 8 out of 139 websites.

5.3.3 Conclusions

Due to the wide variety of ways to implement SSSO authentication, and SSSO buttons being the only consistency among most web applications, we have concluded that our solution should be based around clicking on SSSO buttons to initiate SSSO authentication processes. We confirmed that this is a reliable way to initiate SSSO authentication processes in experiment A.2, but also observed that a significant amount of SSSO authentication processes are simply broken.

5.4 Detection of SSSO Authentication

5.4.1 Problem Description

Zhou and Evans [11] were able to accurately identify SSSO buttons by monitoring HTTP traffic for messages related to SSSO after clicking on buttons. This was in stark contrast to the inaccurate methods used by Ghasemisharif et al. [12], who tried to find evidence of SSSO authentication by scanning a login page for URLs and matches of regular expressions. When we evaluated their methods, and examined the published data set with results, we noticed several problems with their approach, that also applied to Zhou and Evans [11]. A reliable method for discovering SSSO buttons should not only be able to correctly identify SSSO buttons, but should also ensure that the button is part of the website's own authentication system.

5.4.2 Observations

After we click on an SSSO button, the browser usually handles the rest of the SSSO authentication process without any human interaction, as described in background section 2.4. However, user interaction may be required in one of the following cases:

- the user has not been authenticated yet by the SSSO provider
- the user needs to give permission before logging into the website with SSSO

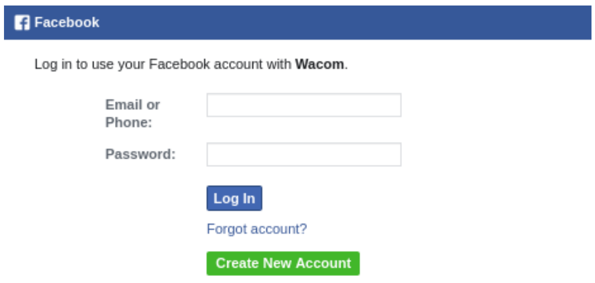
If the user has not been authenticated yet by the SSSO provider, then the SSSO provider typically redirects the browser to a web page for authentication before the user can continue with SSSO authentication, and this page often contains evidence of having clicked an SSSO button. We believe that we can use this behaviour to reliably determine whether or not we have clicked on an SSSO button, as we demonstrate in example 1 for Facebook.

Hypothesis 5.2
If a user has not been authenticated yet by the SSSO provider, and we click on an SSSO button, then the browser is redirected to a page for authentication by the SSSO provider, that contains evidence of having clicked on an SSSO button.

We found in experiment A.3 that this hypothesis always held true as long as SSSO authentication was integrated and configured correctly, which implies that we can use this method to reliably verify the identity of any SSSO button that we can use to automate SSSO authentication. Evidence of clicking on an SSSO button could be found by examining:

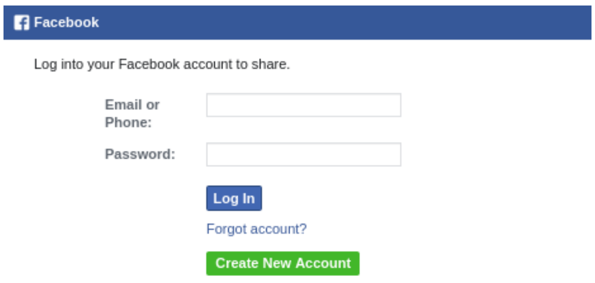
- the URL of the page that we were redirected to
- user information found on the page itself

This way, we can avoid the pitfall of confusing share buttons and other similar services for SSSO buttons. However, we listed several other pitfalls in section 5.2 that we need to address.


Facebook

Log in to use your Facebook account with **Wacom**.

Figure 5.9: (wacom.com) We are redirected to a page that tells us that we clicked on an SSSO button.

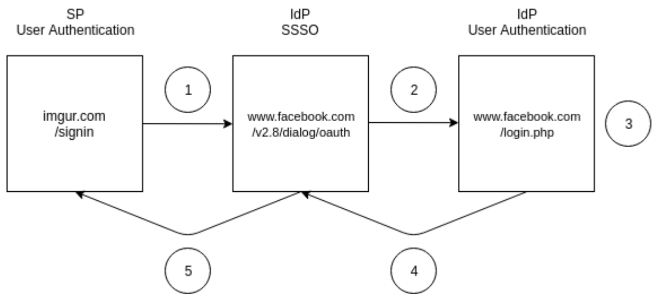

Facebook

Log into your Facebook account to share.

Figure 5.10: (zdnet.com) We are redirected to a page that tells us that we clicked on a share button.

Example 5.1
Typical flow of an SSSO authentication process where the user that has not been authenticated yet by the SSSO provider. This allows us to reliably verify whether or not we clicked on an SSSO button.

When a user visits Imgur, and clicks on Facebook's SSSO button to sign in, the following happens:



1. the browser is redirected to Facebook's SSSO authentication page
2. the browser is redirected to Facebook's user authentication page
3. the user logs into Facebook
4. the browser is redirected back to Facebook's SSSO authentication page
5. the user confirms that they want to use Facebook to log into imgur.com, and the browser is redirected back to imgur.com

Facebook's redirection is implemented by passing URLs as arguments. For example, in step 2, the URL of the user authentication page that the browser is redirected to, has a URL with a 'next' parameter that contains the URL of the SSSO authentication page.

```
https://www.facebook.com/login.php?...&next=<URL>&...
```

We can inspect the value of this 'next' parameter in step 3 to confirm that we were redirected from the SSO authentication page, which in turn confirms that we originally clicked on Facebook's SSSO button.

We can use a similar approach to gather information about the application that we are trying to use SSSO authentication for. The 'next' URL always contains a unique identifier of the application that we are trying to use SSSO authentication for. In addition, Facebook often also shows the display name of an application on the user authentication page.

If we want to ensure that an SSSO button belongs to the web application that we are investigating, then we should attempt to verify that the SSSO button does not belong to another application. This is a potential problem when:

- the website that we are investigating is hosted by a website that includes its own authentication system on the pages of hosted websites (see figure 5.2)
- an embedded third party service uses SSSO authentication for its users (see figure 5.3)

During experiment A.3, the page that the browser was redirected to after clicking on an SSSO button, contained information to identify the application in 323 out of 332 cases (97.29% ± 1.75%, 95% CI):

- in 303 cases (91.27% ± 3.04%, 95% CI), evidence was found in the URL
- in 190 cases (57.23% ± 5.32%, 95% CI), evidence was found within the page itself

In the remaining 9 cases where we could not find any evidence, specific SSSO providers were used, such as Amazon Pay, and thus we can anticipate when we cannot find any evidence.

We can use this information to fingerprint blogging websites and common third party services that can be embedded on other websites, in order to avoid confusing SSSO buttons of other services with SSSO buttons of the website that we are investigating.

In section 5.2, we also listed two other pitfalls related to SSSO detection. We can avoid finding SSSO buttons on unrelated websites by containing our search to web pages of the same domain, which we will discuss further in-depth in section 5.8. The other pitfall is missing evidence of SSSO detection that is hidden in harder-to-reach places, which we will discuss in the next section.

5.4.3 Conclusions

Hypothesis 5.2 can be used to verify whether or not a button that we clicked upon was an SSSO button. In addition, we have also established that the page that we are redirected to, usually contains enough information to identify and avoid pitfall situations.

5.5 Locations of SSSO Buttons

5.5.1 Problem Description

In order to reliably discover SSSO buttons, we need to know where to look for them. SSSO buttons can be found in various locations on websites, and we need to make sure that we do not miss any SSSO buttons because we forgot to look in a certain place. Vice versa, we should avoid searching for SSSO buttons in locations where we are unlikely to discover them.

5.5.2 Observations

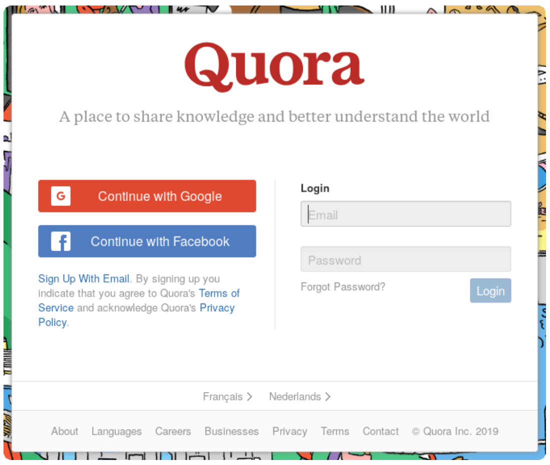
Initial observations led us to believe that we would find SSSO buttons in the same locations as where we would expect to find username-password forms, registration forms, and other components related to authentication.

Hypothesis 5.3
We expect to find SSSO buttons in the same locations as where we expect to find other components related to authentication.

We confirmed this hypothesis during experiment A.4, where all SSSO buttons where find in a typical location. and identified the following locations where we could find components related to authentication:

- main/landing page, main content
- login page, main content
- registration page, main content
- any page, base template
- any page, part of modal window
- any page, part of pop-up menu

It is important to note that these locations are not exclusive. A website can have dedicated pages for letting users log in, and register, while also making it possible for the users to log in through a pop-up menu.



A place to share knowledge and better understand the world

Sign Up With Email. By signing up you indicate that you agree to Quora's Terms of Service and acknowledge Quora's Privacy Policy.

Figure 5.11: (Quora) Unauthenticated users are prompted to log in or register when visiting the landing page.

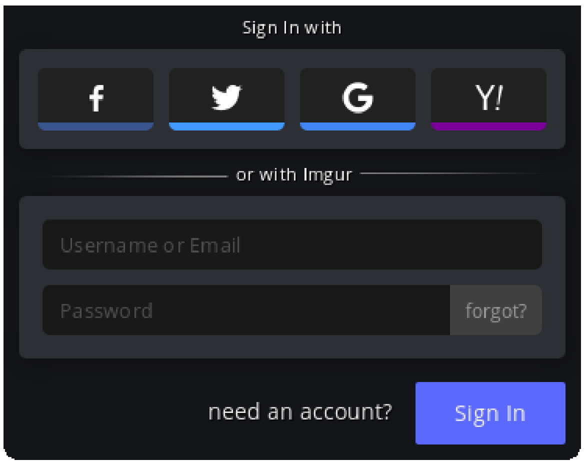


Figure 5.12: (Imgur) A dedicated login page where all possible methods for logging in are presented.

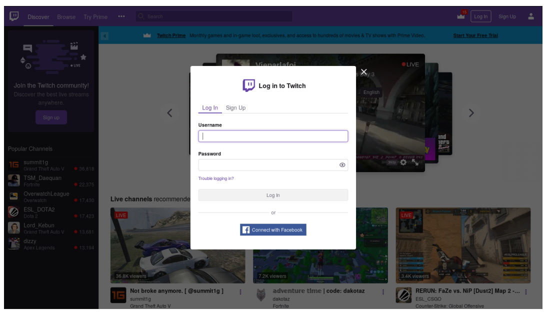


Figure 5.13: (Twitch) A modal window appears in front of the web page after clicking the "Log In" button.

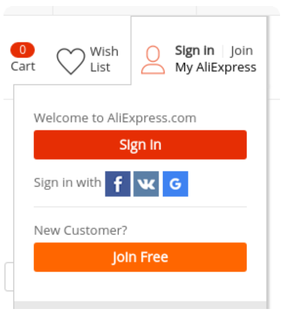


Figure 5.14: (AliExpress) While shopping, an unauthenticated user can access the following pop-up menu to sign in.

During experiment A.4, we observed that a fifth of all websites with user registration did not have a dedicated login or registration page, and instead, users had to log in and register on the landing/main page. We would typically find references to the login and registration pages on the main page, and when the registration page was not referenced on the main page, a reference to it could be found on the login page. Login and registration pages often contain references to each other, and sometimes, the same web page was used for both logging in and registration.

page	**count**		**depth = 0**		**depth = 1**		**depth = 2**		**depth = 3**	
login	33	66.00%	1	3.03%	32	96.97%	0	0.00%	0	0.00%
registration	32	64.00%	0	0.00%	27	84.38%	5	15.63%	0	0.00%

Table 5.1: Experimental data from investigating 50 different websites with user registration.

On almost half of all websites, we discovered components related to authentication, such as references to a login page or username-password forms, in window modals or popup-menus. Although there are cases where the contents of window modals and popup-menu can be accessed by checking the DOM of a web page, websites may also load their contents only after the user has clicked on a button.

Not only did we find SSSO buttons in the same locations as where we would expect to find other elements related to authentication, but when we found an SSSO button, then they were almost always in the presence of other authentication methods, such as username-password forms, or registration forms. This implies that we are much more likely to find SSSO buttons near other authentication methods than anywhere else on a page.

If we discover an SSSO button and have verified its identity using hypothesis 5.2 while avoiding known pitfalls, then we can be quite certain that a website supports SSSO. The other way around, the lack of an SSSO button in the vicinity of a login form or a registration form, can also be a reliable indicator to determine whether or not a website supports SSSO authentication or not. This gives rise to our next hypothesis.

Hypothesis 5.4
If we find a way for users to authenticate, such as a username-password form, or a registration form, but do not find any evidence of SSSO authentication in their vicinity, then it is very likely that the website has not integrated SSSO.

Although the hypothesis is not waterproof, it allows us to make a predictions about the integration of SSSO authentication by a website in cases where we are able to find a username-password forms, or other ways to authenticate users.

5.5.3 Conclusions

Based on hypothesis 5.3 and the observations made during experiment A.4, we conclude that an effective search strategy for discovering SSSO buttons at least entails the following:

- searching the main, login, and registration pages
- a maximum search depth of 1 of 2
- interacting with pages to discover the contents of window modals and popup-menus

We are able to make reliable predictions about the integration of SSSO authentication by using hypothesis 5.4, in cases where we have discovered components related to authentication.

5.6 Recognizing SSSO Buttons

5.6.1 Problem Description

We already established where to search for SSSO buttons in section 5.5, but we never addressed the problem of how to recognize these buttons. Human users can easily recognize SSSO buttons based on visual cues, but there is no straightforward way for a computer program to recognize an SSSO button. Developers use a combination of HTML, CSS, and JavaScript to make web applications behave and look exactly the way that they want, and this flexibility stands in the way of programmatically recognizing SSSO buttons.

We can rely on hypothesis 5.2 to reliably verify that an HTML element is an SSSO button, but we cannot use a brute force approach that involves trying out every element on a web page, because that would take too much time. Along the same lines, if we use a method that might result in too many candidates that we need to test, we need a way to cut down the amount of candidates by filtering or sorting candidates, without removing any SSSO buttons from the pool of candidates.

5.6.2 Observations

Zhou and Evans [11] have shown that it's possible to recognize SSSO buttons in a reliable way by searching for keywords in the source of HTML elements. We also identified another potentially viable approach that uses visual recognition to recognize SSSO buttons.

Visual recognition is a form of machine learning where a computer program learns how to recognize things based on images containing examples. This is done by building a data set of images in which all items of interest are tagged with a classifier, and then training a computer program to recognize those items using the data set. For example, a data set that is used to teach computer programs how to recognize different types of furniture, might include a picture of a dining room, in which all chairs are marked as a chair, a table would be marked as a table, etc. In our case, we would attempt to tackle this problem by taking screenshots of a large amount of different web pages or HTML elements, and would tag every SSSO button with a classifier. We think that such an approach could be highly effective due to the distinct looks of SSSO buttons.

However, despite their similarities, SSSO buttons of the same SSSO provider can still look very different looks, as demonstrated in figure 5.15. Another problem is that adding support for each SSSO provider might be a long and intensive task, where we have to provide thousands of examples of buttons. And last but not least, we did not have any prior experience with machine learning and visual recognition.

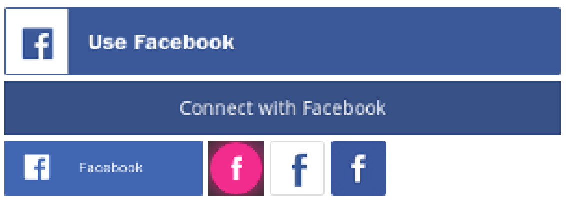
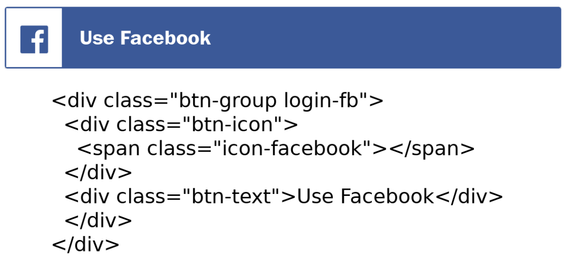


Figure 5.15: Examples of SSSO buttons for Facebook found on different websites.



```
<div class="btn-group login-fb">
  <div class="btn-icon">
    <span class="icon-facebook"></span>
  </div>
  <div class="btn-text">Use Facebook</div>
  </div>
</div>
```

Figure 5.16: (Washington Post) An example of the HTML underlying an SSSO button.

So we opted to use an approach that is similar to the one used by Zhou and Evans [11], where we attempt to recognize SSSO buttons by searching the source of HTML elements for specific keywords related to the SSSO provider.

Hypothesis 5.5
We can recognize SSSO buttons by searching for keywords related to an SSSO provider.

We investigated the validity of this hypothesis in experiment A.5, and confirmed that this hypothesis held true in close to all cases. Just like in figure 5.16, the HTML source of SSSO buttons contained one or multiple references to its SSSO provider in 122 out of 126 cases. When the hypothesis failed, the button looked like a generic login button with no hint of it being an SSSO button.

Zhou and Evans [11] searched the attributes and visible text of elements for keywords by using the case insensitive regex `facebook`. However, during our experiment, we observed several cases where this approach was not enough. We found that multiple keywords were used for most SSSO providers, and that these references weren't always found by only checking the attributes or visible text of buttons. For this reason, we needed to take another look at the following questions:

- Where do we search for keywords?
- Which keywords do we look for?

To determine where to search for keywords, we compared different options. Out of 122 cases where hypothesis 5.5 held true, we found that we could use:

- inspecting visible text based approach in 63 out of 122 cases (51.64%)
- inspecting the id and class attributes in 80 out of 122 cases (65.57%)
- inspecting all attributes in x out of 122 cases (100.00%)
- inspecting all attributes and visible text in x out of 122 cases (100.00%)

Ideally, we would scan the full outer HTML of an element for keywords, which includes the opening tag, the closing tag, and everything in between. However, a major downside of doing this is that any element with a descendant that contains a keyword would also be tagged as a potential candidate.

As mentioned earlier, Zhou and Evans [11] used the case-insensitive regex `facebook` to identify potential Facebook SSSO buttons. During our experiment, it quickly came to our attention that this is not adequate enough in a significant amount of cases. For example, the regex `facebook` only matched in 51 out of 55 cases (92.73%), and the regex `google` only matched in 29 out of 33 cases (87.88%). In several cases, references were made to the SSSO provider using abbreviations. Based on our observations in experiment A.5, we put together the following table of regexes for different SSSO providers:

SSSO provider	**long keywords**	**short keywords**
Facebook	`(facebook|fbook)`	`fb`
Google	`(google|gplus|g[\-_]?oauth)`	`gp`
Twitter	`(twitter|twttr)`	`tw`
VKontakte	`vkontakte`	`vk`
mail.ru	`(mailru)`	`mru`

Table 5.2: Regexes used to match long and short keywords related to SSSO providers.

We decided to make a distinction between regexes for long keywords and regexes for short keywords. Long keywords suffice in most cases, but shorter keywords are necessary to have complete coverage. However, the use of shorter keywords results in finding a lot of false positives. If we want to use short keywords for recognizing SSSO buttons, we also need a way to cut down the pool of possible candidates.

An approach that searches the attributes and visible text of elements for long keywords related to the SSSO provider, typically results in a small pool of candidates and suffices to find a large amount of SSSO buttons. However, for an approach with quasi-complete coverage of all SSSO buttons, we would need to search the full outer HTML of an element for long and short keywords. Both of these aspects result in a lot of candidates that are false positives, which means that, if we want to use such an approach, we need to drastically cut down the amount of candidates in an effective way.

A first way to cut down the amount of potential candidates for SSSO buttons is to filter out any HTML elements that are not buttons. However, due to the flexibility and diversity of the Web platform, any HTML element can be a button:

- The 'button' tag name should be used when an element is a button, but we have found many examples of HTML elements that behave and look just like a button, but have different tag names, such as *input*, *a*, *li*, *div*, *span*, *img*, etc.
- Technically, we can click on any HTML element due to the nature of event capturing and event bubbling. A click event passes by many different elements. When we click on a descendant of an HTML element, we also click on the HTML element itself.
- Default behaviour when clicked upon is defined for area elements, anchor elements, button elements, and input elements with type 'button' or 'submit'. However, default behaviour can be disabled or overwritten.
- JavaScript can be used to attach a click event listener to any element, and there is no out-of-the-box way to determine whether or not an event listener of a certain type is attached to an HTML element.
- By default, the mouse cursor changes when it is positioned on top of an element that can be clicked. CSS can be used to emulate the changing cursor by changing the value of an element's 'cursor' CSS property to have the value 'pointer'.
- Web developers can add attributes to the elements of web pages to help improve accessibility for people with disabilities (e.g. microdata, ARIA). However, the number of websites that actually add these attributes is small.

None of these aspects can be used on their own to reliably determine whether or not an element is a button. However, by combining these different aspects, we create a very reliable indicator of what a button is.

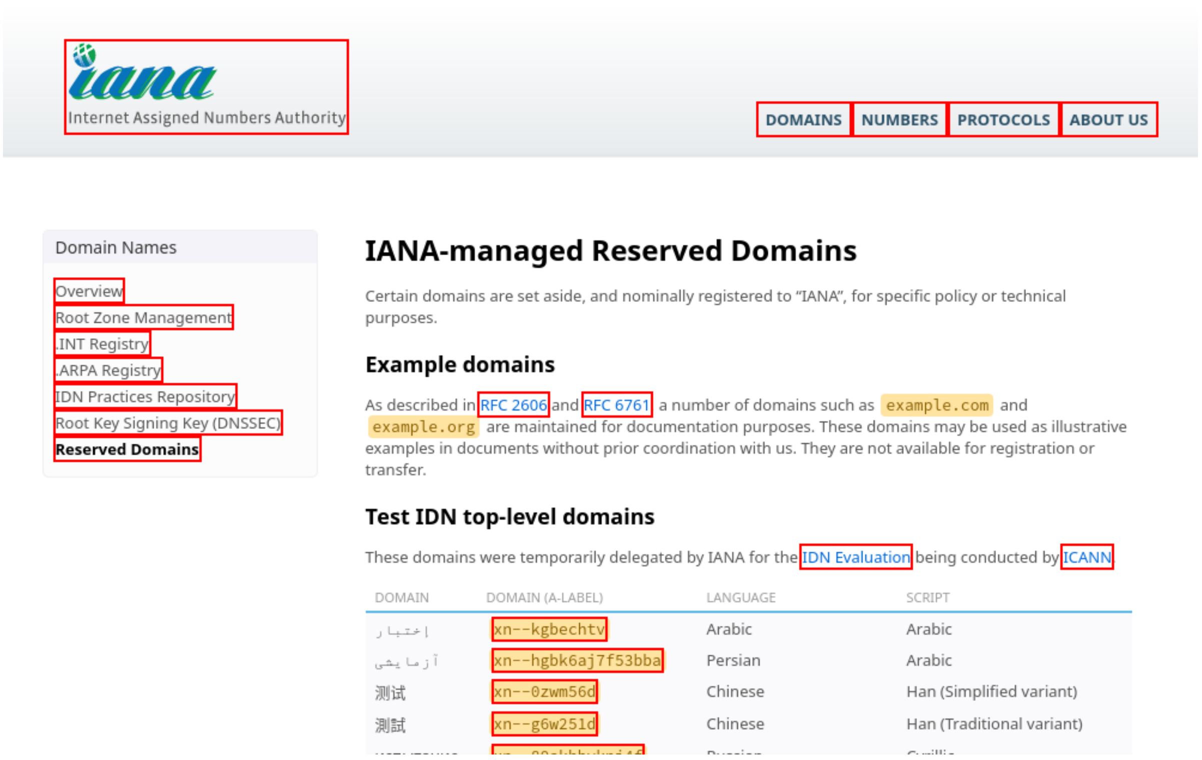
Hypothesis 5.6
A HTML element is considered a clickable button if one of the following applies:

- default behaviour to handle a click-related event defined for element
- a click-related event listener is attached to the element
- cursor is a pointer when position above the element
- it is marked as a button using an attribute related to accessibility

---

Note that the following are click-related events: 'click', 'dblclick', 'mouseup', 'mousedown'.

We can easily check each indicator of hypothesis 5.6, with the exception of the second one. Determining whether or not a click-related event listener is attached to an element is not possible out of the box, but we used a browser extension to add this functionality. We injected a script that uses a technique called monkey patching to intercept all function calls that attach event listeners to HTML elements. This way, we could keep track of which event listeners were attached to each HTML element.



IANA-managed Reserved Domains

Certain domains are set aside, and nominally registered to "IANA", for specific policy or technical purposes.

Example domains

As described in RFC 2606 and RFC 6761 a number of domains such as example.com and example.org are maintained for documentation purposes. These domains may be used as illustrative examples in documents without prior coordination with us. They are not available for registration or transfer.

Test IDN top-level domains

These domains were temporarily delegated by IANA for the IDN Evaluation being conducted by ICANN.

DOMAIN	DOMAIN (A-LABEL)	LANGUAGE	SCRIPT
إختبار	xn--kgbechtv	Arabic	Arabic
آزمایشی	xn--hgbk6aj7f53bba	Persian	Arabic
测试	xn--0zwm56d	Chinese	Han (Simplified variant)
測試	xn--g6w251d	Chinese	Han (Traditional variant)

Figure 5.17: (IANA) A filter that only marks clickable elements at work.

However, this is not the only way to filter candidates for SSSO buttons. Based on our observations, we found several other traits among SSSO buttons that are universally shared.

Hypothesis 5.7
We can filter candidates based on the following nearly-universal traits of SSSO buttons:

- SSSO buttons are expected to be clickable buttons (see hypotheses 5.1, 5.6)
- SSSO buttons are expected to be visible to the user
- the height of an SSSO button tends to be within certain margins (20px $<$ height $<$ 400px)
- the length of visible text within an element has to be limited ($<$50 characters).

In sections 5.7 and 5.9, we will discover that another way to limit the amount of candidates, is to only search in parts of the page that have changed. For example, when we click a button, and a pop-up menu appears, we should only search for new candidates in that popup-menu.

Even if we are able to filter out a large amount of candidates that are unlikely to be SSSO buttons, the pool of candidates might still be too large. At this point, we have only limited the pool of candidates to clickable buttons that contain references to an SSSO provider, and share several properties that are common among SSSO buttons. This might include buttons related to other services provided by the SSSO provider, such as buttons for sharing articles, or joining a fan page, etc. Another case where this might be a problem, is if one of the SSSO provider's keywords commonly appears on a web page.

To solve this problem, we can sort these candidates by the likelihood of being SSSO buttons, and only test the most likely candidates. Zhou and Evans [11] had identified several indicators that increased or decreased the likelihood of an HTML element being an SSSO button:

- the position of an element on a web page
- the presence of certain terms, such as 'auth' (increase) or 'share' (decrease)

On login and registration pages, SSSO buttons were typically part of the main content, while on other pages, they could be found anywhere, including in window modals, and pop-up menus. Terms indicative of authentication, such as *login*, *signin*, *auth* increase the likelihood, while terms such as *share*, *support*, etc. led to decreases in likelihood. Other factors, such as the width or height of an element were deemed less indicative, and thus less helpful when sorting candidates.

Hypothesis 5.8
We can sort the candidates based on the following traits:

- candidates that are further away from positions where we expect to find SSSO buttons, are less likely to be SSSO buttons
- candidates that contain terms that are related to authentication are more likely to be SSSO buttons
- candidates that contain terms that are related to other services provided by an SSSO are less likely to be SSSO buttons
- candidates that are further away from other authentication methods, are less likely to be SSSO buttons (see hypothesis 5.4)

After having scored every candidate, we can apply a cautious cut-off, so that any of the downsides of of checking the full outer HTML of elements for long and short keywords have been reduced to a manageable level.

Now that we have established a way to reliably select candidates for SSSO buttons, we can combine this selection method with hypothesis 5.2 to find all SSSO buttons. By clicking on each candidate, we can accurately determine whether or not they are SSSO buttons.

5.6.3 Conclusions

We can recognize SSSO buttons by searching the source of HTML elements for keywords related to SSSO providers. Different approaches have advantages and disadvantages. To have as much coverage as possible, an approach that searches the full outer HTML of elements for long and short keywords should be used, in combination with limiting the amount of candidates through filtering and sorting by likelihood of being an SSSO button. We can verify whether or not a candidate is an SSSO button by using hypothesis 5.2.

5.7 Clicking on SSSO Buttons

5.7.1 Problem Description

We have already established where to look for SSSO buttons, and how to recognize them. However, as we observed in section 5.5, SSSO buttons may be located in pop-up menus or modal windows, in which case, prior interactions with other elements of the web page are typically required to reveal those SSSO buttons.

These prior interactions often involve clicking on a button that results in a pop-up menu, or a modal menu appearing. Similarly to how we investigated where to look for SSSO buttons, and how to recognize them, we need to investigate these same problems for these buttons.

Besides identifying which sequences of actions might be necessary to click on SSSO elements, We also need a way to record these action sequences. For this, an appropriate way to reference HTML elements is need that is resilient to page navigation, and element staleness.

Last but not least, the same problems that we tried to solve for SSSO buttons, need to be solved for login buttons, and other elements that we need to discover before we can discover SSSO buttons.

5.7.2 Observations

When an unauthenticated user visits the main page of a website, the web page usually contains one or multiple incitements to log in. Sometimes, a registration form or username-password form is presented immediately to the user, but more often than not, ways to authenticate are hidden behind a login button. When the user interacts with one of these login buttons, either a modal window or pop-up menu apears, or the browser is redirected to a dedicated login or registration page. In either case, interacting with these login buttons is an important step in our discovery of SSSO buttons.

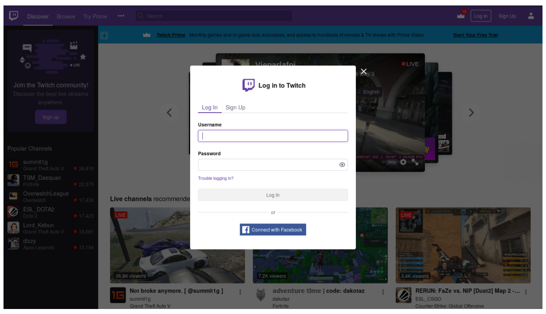


Figure 5.18: (Twitch) A modal window appears in front of the web page after clicking the "Log In" button.

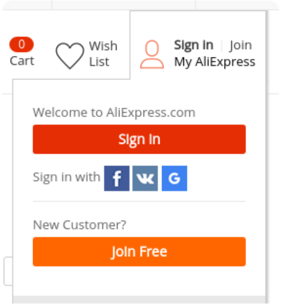


Figure 5.19: (AliExpress) A pop-up menu appears when the mouse cursor is placed over the sign-in button.

We found that in a large majority of cases, prior interactions involved clicking on a button, while in some cases, the user had to place the mouse cursor on top of an element, followed by clicking on an SSSO button as the final action. Sometimes, multiple interactions were necessary before SSSO buttons were revealed. As such, we are able to generalize our interactions with a web page as a sequence of clicks and hovers.

Hypothesis 5.9
Any sequence of interactions that results in clicking on an SSSO buttion is composed of the following interactions:

- click on an element
- hovering, or placing the mouse cursor on top of an element

We confirmed this hypothesis in experiment A.7, in which we observed 35 different sequences of actions that resulted in clicking on an SSSO button. During this experiment, we observed an interesting trend regarding the length of these sequences of interactions:

- prior interactions were almost always required on main pages
- no prior interactions were needed for unique sequences found on login/registration pages

Note that we did observe longer sequences on login and registration pages. However, in these cases, the same sequences were also found on the main page, as they were usually part of the website's template. Based on these findings, we think that an effective approach should only attempt to discover longer sequences of interactions on the main page.

page	**n=1**		**n=2**		**n=3**		**n=4**		**n=5**	
main	0	0.00%	10	28.57%	5	14.29%	0	0.00%	0	0.00%
login	12	34.29%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
registration	8	22.86%	0	0.00%	0	0.00%	0	0.00%	0	0.00%

Table 5.3: Amount of interactions (=n) needed to click on an SSSO button per type of page.

Within the context of programmatic browsers, these sequences of interactions are commonly referred to as action chains, and most programmatic browsers have built-in functionality for handling them. An action chain is an ordered sequence of actions, where each action is comprised of an event, such as clicking, and a reference to an HTML element in the loaded page, upon which the action is performed.

A first problem related to the use of programmatic browser is the limited scope of focus of programmatic browsers. A loaded web page in a tab typically consists of one or multiple frames: the main frame, and any embedded frames within the main frame, which in turn, can also contain frames. A frame is a browser context, and a programmatic browser can only operate within the browser context that it is currently focused upon. Therefore, if we want to access an element in another frame, then we should change the browser context that our browser is currently focused upon.

Although it is not very common, SSSO buttons and other elements of interest can sometimes be found in embedded frames. Therefore, we should access and search embedded frames as well, and thus our sequences of interactions, when using a programmatic browser, should also include actions related to switching the focus from one frame to another.

Another problem related to programmatic browsers and action chains is the referencing of HTML elements on which to perform actions. Most programmatic browsers use references to HTML elements that currently exist within a web page, which become stale, or invalid, upon any form of page navigation, such as the page being refreshed, or when the element is removed from the page. As such, we need to find a way to reference HTML elements that is resilient to becoming stale, and only fails when the element simply is not present on the page.

Example 5.2
An example of the persistent CSS selectors that we will use to reference elements.

```
html > body:nth-child(0) > div:nth-child(3) > p:nth-child(2)
```

Besides using a CSS selector to reference elements, we also identified the use of an element's position, or an image of an element, as a potential way to reference elements in a persistent way. Using an image of a button seems ideal with a visual approach as described in 5.6, but wasn't further explored due to taking a different approach. Using the position of an element to reference elements turned out to be a bad idea, due to the reliance on several factors: changing the viewport of a browser window can result in changing the whole layout of a page, and its elements. Another problem is the inaccuraccy of this method: in any given position, an element, its ancestors, and its descendants may all locate the same location. This approach would also not allow us to reference frame elements in a persistent way.

As such, we decided to reference HTML elements using CSS selectors. Using a CSS selector allows us to avoid most, if not all, of the disadvantages listed above, but will only suffice if carefully crafted. In addition, we identified the following properties as ideal:

- every element has a unique selector
- changes in other parts of the page do not invalidate our selector
- the selector does not match any other element if the element is not present

Not all elements have id or class attributes, and thus using CSS selectors based on these attributes cannot be used for every HTML element. In addition, the value of an element's class attribute can be quite volatile, and the HTML standard requires that the values of id attributes are unique, but this is not enforced in reality. The scope of these selectors also tends to be global, and thus may be invalidated by changes in other parts of page. In experiment A.6, we found that selectors using only IDs could only select elements of interest in 20% of all cases, while selectors using class attributes could be used to select around 85% of all cases.

Every HTML element has a tag name and a position relative to its sibling HTML elements, which we can use to uniquely identify the element among its siblings. We can also use specific selectors to limit the search scope among the child elements of an element. By combining these selectors, we can create a CSS selector for selecting the n'th child element with a given tag name:

- an nth-type selector can be used to select the nth element that matches a selector

  ```
  selector:nth-of-type(n)
  ```

- a child selector can be used to search among a parent's child elements:

  ```
  parent_selector > child_selector
  ```

- when combined, we can select the n'th child of a parent element:

  ```
  parent_selector > child_selector:nth-of-type(n)
  ```

This way, we can recursively build a unique CSS selector for any element in a page, by recursively selecting the next descendant in the web page's DOM until the element is reached, as described in algorithm 1. In experiment A.6, we were able to select all cases using these persistent CSS selector. An example of such a persistent CSS selector is shown in example 2.

We consider these persistent CSS selectors as ideal, because, every element has a unique selector, and due to the use of child selectors, the search scope is always kept as small as necessary. The selector could potentially match other siblings if the element is not present, but because the HTML elements of interest are usually in fairly static parts of a page, this approach still seems to hold up, as confirmed in experiment A.6.

---

algorithm 5.1
build_persistent_selector(e) builds a unique CSS selector for a given HTML element *e*.

A helper function build_child_selector(e) returns the next part of our CSS selector, which selects the n'th child element with a given tag name. If an HTML element *child* has tagname *TAGNAME*, and is the *N*'th child with that tagname among its siblings, then a function call will return the following string:

```
build_child_selector(child) -> "<TAGNAME>:nth-of-type(<N>)"
```

For example, if the *child* element is the second child element with tagname DIV, then the result will be the string "DIV:nth-of-type(2)".

The main algorithm recursively builds up the selector string, by visiting element *e* and every ancestor of *e*, starting from element *e*, target element, until the root element of the web page is reached.

1. First we will set up the algorithm, by initializing a variable *c* used to designate the current element, and a variable *s* used to store the CSS selector while we are building it.

   ```
   var c = e;
   var s = "";
   ```

2. We then build up most of the query selector by visiting element *e*, and every one of its ancestors, until we reach the document's root element. At every node, we prepend another piece of the final CSS selector.

   ```
   while (c.parentElement !== null) {
     s = " > " + build_child_selector(c) + s;
     c = c.parent;
   }
   ```

3. When the root element of the web page has been reached, we simple prepend the root element's tagname to the string.

   ```
   s = c.tagname + s;
   ```

4. Now variable *s* contains the query selector, and the algorithm is finished.

5.7.3 Conclusions

In this section, we have discussed how we can record the sequences of actions, often referred to as action chains, that lead up to clicking an SSSO button. Every step of an action chain is comprised of an action to perform (click, hover), and some way to select an element of a page. Action chains for approaches using programmatic browsers are complicated by the fact that one needs to handle the switching between different browser contexts. The selection method for the HTML element to perform an action upon needs to be accurate, and survive page navigation, for which a CSS selector build out child selectors seems to work the best.

5.8 Finding URLs of Relevant Web Pages

5.8.1 Problem Description

In section 5.5, we found that SSSO buttons are commonly found on login and registration pages, and thus our approach should include the discovery of these pages. We also found that if a website has a login page or a registration page, then references to these pages can typically be found on the main page. However, due to the nature of modern web applications, these references aren't always anchor elements with the page's URL as an href attribute.

No prior research has investigated the extraction of URLs from web pages, but Ghasemisharif et al. [12] tried guessing the URLs of login and registration pages. We also identified search engines as a potential source for URLs, and websites may provide other sources of information where one can possibly find references to these pages.

5.8.2 Observations

We have identified multiple sources of information where we can find references to login and registration pages. Although each method has its merits, it can be difficult to extract URLs in some cases, and some methods also tend to bring up some concerns about whether or not the URLs that have been found should be accessible to users.

In a lot of cases, the references to other web pages can be found in the DOM of the web page itself. We can typically find new URLs by checking certain attributes of the following HTML elements:

- the href attribute of an anchor elements and area element
- the action attribute of form elements
- the src attribute of iframe elements

Not only are these sources easy to find, and is it easy to extract URLs from them, but they also cover a lot of grounds. During experiment A.8, the URLs of login and registration pages were stored as attributes of an HTML in 69 out of 75 cases (92.00%), and we would've been able to successfully scrape 15 out of 20 websites (75.00%) without worrying about URLs stored in JavaScript.

However, a signification amount of modern web applications also use JavaScript to implement behaviour related to redirecting the user to authentication pages, in which case, the URLs cannot be extracted as easily. URLs of login and registration pages may be hidden in event listeners, or the scripts that are called by those event listeners. We could inspect the source code of a web page and included scripts to identify URLs, but it would be difficult to recognize any URLs besides absolute URLs. Instead, we think that it's better to rely on actually clicking on login buttons, and checking which page we are redirected to, as discussed in section 5.7.

Websites can also provide other sources for potentially finding URLs, such as sitemaps, and robots.txt files. Although sitemaps almost always contain the URLs of login and registration pages, only few websites actually include them, and thus sitemaps aren't a very common source that we can rely on. Similarly, robots.txt files are also not available for all websites, and they usually do not include full URLs, but instead only include parts of a URL's path.

Similarly to Ghasemisharif et al. [12], we can also try out well known URL patterns of login and registration pages, or use search engines to find the URLs of these pages. The URLs of login and registration pages tend to follow certain patterns such as `www.domain.com/login/` or `secure.domain.com/auth/login/`. The crawling strategy that is typically used by search engines aims to index every page, and thus is more thorough than ours. Both methods can be used to find URLs of pages that we would normally miss. However, we should be careful when guessing URLs or when using a search engine, as we might find URLs that are not meant for normal users, such as the login page for web administrators.

Based on the advantages and disadvantages of the aforementioned methods, and the context in which we will be using these methods, we decided on employing the following methods to find the URLs of the login and registration pages, in a way that mimicks a regular viewer:

- inspecting the attributes of anchor elements, form elements, and frame elements
- clicking on buttons, and inspecting the URLs of newly loaded pages after page navigation

We are not interested in every URL that we can extract from a web page, and thus we should filter out the URLs of pages where we are unlikely to find the SSSO buttons of a website. We are most interested in the URLs of login and registration pages, which typically contain several keywords that we can use to identify such URLs. Similarly to how we filtered and sorted candidates of SSSO buttons in section 5.6, we should filter and sort the URLs that we discover.

When we took a closer look at the data set published by Ghasemisharif et al. [12], we found that their web crawler regularly ended up scanning web pages belonging to unrelated websites. In order to avoid this problem, a page should only be scanned if the top private domain of its URL is a subdomain of the target domain that we are investigating, as demonstrated in table 5.4.

target domain	**URL**	**top private domain**	**accept?**
bbc.co.uk	https://www.bbc.co.uk/login/	bbc.co.uk	yes
bbc.co.uk	https://secure.bbc.co.uk/login/	bbc.co.uk	yes
bbc.co.uk	https://www.ikea.co.uk/login/	ikea.co.uk	no

Table 5.4: Examples of which URLs should be accepted and rejected.

This measure may fail when a federated identity manager is used by several separate websites, and is hosted under a different top private domain than the target domain. However, the use of federated identity managers is not very common, and thus this measure still helps us to drastically improve the quality of our results.

5.8.3 Conclusions

We have concluded that our approach should inspect the attributes of HTML elements for URLs of login and registration pages, and that we should attempt to click on login buttons to discover the URLs of login pages, in case that we are redirected. We should also avoid scanning unrelated web pages.

5.9 Searching for SSSO Buttons On A Page

5.9.1 Problem Description

The main research problem of this chapter is the discovery of entry points for SSSO authentication processes, and as we determined in section 5.3, SSSO buttons are the only entry points of SSSO authentication processes that are consistently available. We already explored several problems related to the discovery of SSSO buttons that needed to be addressed before we could shape a solution for this problem: where to look for SSSO buttons, how to recognize SSSO buttons, how to click on SSSO buttons, and how to find the URLs of relevant web pages. We now present a general method to discover SSSO buttons on a web page, before discussing how to handle searching a website in the next section.

5.9.2 Observations

To search a page with a given URL for SSSO buttons, we can use a strategy that repeatedly tries out action chains in order to determine whether or not the action chain results in clicking on an SSSO button, and if not, tries to discover new items of interest that might eventually lead us to clicking on an SSSO button.

---

algorithm 5.2
search_page(url, found_urls, ssso_action_chains) scans a web page with a given URL *url* for URLs of other web pages that need to be scanned, and action chains to click on SSSO buttons, which are stored in the parameters *found_urls* and *ssso_action_chains*, respectively.

We use a queue to store the new action chains that we discover while searching. We start out by queuing an empty action chain, and pop the next action chain from the queue, until the queue is empty. For each action chain, we do the following:

1. reload the web page
2. execute the action chain
3. process the results

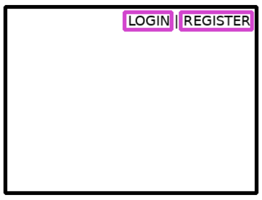
We reload the page before executing the action chain to ensure that the execution of previous action chains does not affect the results. Depending on what happens after executing the action chain, we do one of the following:

- **browser loaded page for authentication by SSSO provider**
  We can use hypothesis 5.2 to determine whether or not we clicked on an SSSO button, and if so, we add the current action chain to *ssso_action_chains*.
- **browser loaded another page**
  If the page that we were redirected to is a login or registration page, then the URL of the page should be stored in *found_urls*.
- **browser did not load a new page**
  We search new parts of the page for items of interest: SSSO buttons, login and register buttons, and URLs of login and registration pages. For each button that we discover, we extend the current action chain with an action for clicking on, or hovering over, that button, and queue the new action chain so that we can try it out later. URLs of login and registration pages are added to *found_urls*.

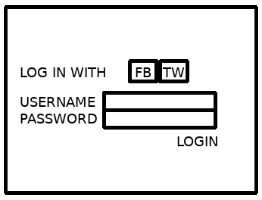
When the queue of action chains is empty, we have finished scanning the web page.

To further demonstrate the workings of algorithm 2, we provide some examples of typical situations. When we scan a website, we always start by scanning the main page of the website.

action chain: [ ]
When we first investigate a web page, the action chain that we are trying out is an empty action chain, and thus we can skip the execution of the action chain. We check the whole page, including all embedded frames, for any items of interest. In this situation, we typically discover login buttons and registration buttons, and the URLs of login pages and registration pages. We queue new action chains for clicking on the buttons that we have discovered, and add the URLs that we have discovered to *found_urls*, so that we can visit those web pages later.

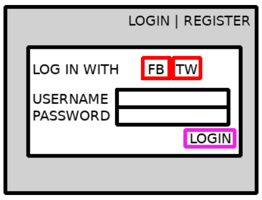


action chain: [**(click, login button)**]
After reloading the main page, we execute the action chain that results in clicking on the login button. The browser loads a new page, which is the website's login page, in which case, we do not investigate this page, but instead add the URL of the login page to *found_urls*, so that we can visit it later.

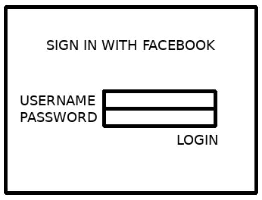


When we scanned the main page for the first time, we also added an action chain to the queue, which results in clicking on the register button. When we click it, the browser might load a registration page whose URL is stored for later scanning, just like when we clicked on the login button. Because we did not discover any new buttons to interact with, after we clicked on the login and registration buttons, no new action chains were added to the queue, and thus the queue is empty now, which indicates that we have finished scanning the web page. However, it is also possible that the browser didn't load a new page after clicking on the login and registration buttons, but that instead, a pop-up menu or a modal window appeared.

action chain: [**(click, login button)**]
After reloading the main page, we execute the action chain that results in clicking on the login button. A modal window has appeared in front of the rest of the page, in which case, we search the modal window for items of interest. In this situation, we typically discover SSSO buttons, and other authentication methods, such as username-password forms. We queue new action chains for clicking on the buttons that we have discovered, and add the URLs that we have discovered to *found_urls*, so that we can visit them later.



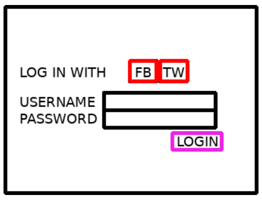
action chain: [**(click, login button), (click, FB button)**]
After reloading the main page, we execute the action chain that results in clicking on the FB button. The browser loads a new page, which is a page for authentication by the SSSO provider. In this case, we investigate the page for evidence to determine whether or not we were redirected to an SSSO provider's login page to use SSSO authentication. If so, then we know that the current action chain results in clicking on an SSSO button, and we add it to *ssso_action_chains*.



When we clicked on the login button and the browser loaded the website's login page, we stored the URL of the login page. The situation tends to be simpler for login pages. As we established in section 5.7, prior interactions are typically not needed to click on SSSO buttons that are located on login and registration pages, and thus the situation tends to be a lot simpler.

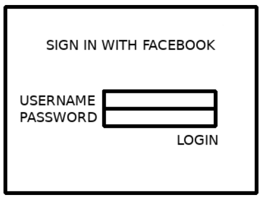
action chain: [ ]

When we first investigate a web page, the action chain that we are trying out is an empty action chain, and thus we can skip the execution of the action chain. We check the whole page, including all embedded frames, for any items of interest. In this case, we typically discover SSSO buttons, other authentication methods, such as username-password forms, and the URL of the registration page. We queue new actions chains for clicking on the buttons that we have discovered, and add the URL that we have discovered to *found_urls*, so that we can visit it later.



action chain: [(**click, FB button**)]

After reloading the main page, we execute the action chain that results in clicking on the FB button. The browser loads a new page, which is a page for authentication by the SSSO provider. In this case, we investigate the page for evidence to determine whether or not we were redirected to an SSSO provider's login page to use SSSO authentication. If so, then we know that the current action chain results in clicking on an SSSO button, and we add it to *ssso_action_chains*.



Although the presented algorithm is fairly simple in its general form, there are still several smaller problems, which have been omitted so far. When implementing this algorithm using a programmatic browser, the following problems need to be addressed:

- to determine which parts of the page need to be scanned after executing an action chain, the states of a page right before and after executing the last step of the action chain must be compared; any element that has become visible, or any element that was newly added to the page, should be investigated
- when we extend action chains to interact with a button inside of an embedded frame, the action chain should also include actions to change the focus of the browser

We may also encounter situations where the algorithm would take too long, or never finish. As such, the following measures should be taken:

- when searching for buttons to interact with, these should be filtered and sorted/prioritized, as discussed in section 5.6
- an action chain should contain at most 4 interactions with buttons, any longer action chain is unlikely to be valid, as demonstrated in section 5.7

5.9.3 Conclusions

We have presented a general method for scanning a web page for SSSO buttons and URLs of other web pages that we might be interested in. This can be done by repeatedly trying out action chains and processing the results to determine whether or not we clicked on an SSSO button.

5.10 Searching for SSSO Buttons On A Website

5.10.1 Problem Description

Now that we have established how to search a single web page for SSSO buttons by using the **search_page** algorithm, we present a general approach for scanning a whole website for SSSO buttons. To conduct this search in an efficient way, we need a way to establish which pages to scan, and in which order they should be scanned.

5.10.2 Observations

Our approach starts by scanning the main page for SSSO buttons using **search_page**, and then scans other pages, based on the URLs of login and registration pages that we were able to extract from the main page. We keep scanning pages to discover SSSO buttons and new URLs, until there are no more pages left to scan.

---

algorithm 5.3
search_website(url, ssso_buttons) searches a website for SSSO buttons. The website's main page has the URL *url*. Information about discovered SSSO buttons is stored in passed parameter *ssso_buttons* for access by the caller, after the algorithm is done.

We use a priority queue to keep track of the URLs that we still have to visit. The priority of a URL should be higher if it is more likely that SSSO buttons can be found on the its web page. We should also implement some form of memory, to remember which URLs and web pages have already been visited.

We start by adding the URL of the website's main page to the queue. As long as our priority queue of URLs is not empty, we repeat the following steps.

1. In any of the following cases, we should not scan the web page:
   - the top private domain of the URL is different from the target domain's
   - we have already visited a web page with this URL
2. Let the browser load the web page with the given URL. The browser may load a web page with a different URL after redirection.
3. In any of the following cases, we should not scan the web page:
   - the top private domain of the URL is different from the target domain's
   - we have already visited a web page with this URL
   - the contents of the loaded web page are too similar to the contents of a web page that we have already scanned
4. Scan the page using **search_page(url, found_urls, ssso_buttons)** algorithm.
5. After scanning the page, we process the results by:
   - add the URLs that were discovered to the URL queue
   - add information about discovered SSSO buttons to parameter *ssso_buttons*

We scan web pages until the priority queue is empty, in which case, we have finished scanning the whole website for SSSO buttons, and all information about the discovered SSSO buttons is stored in *ssso_buttons*.

We already established in section 5.5, that the maximum search depth of an effective strategy should be 2 links at most. The order in which we visit pages should reflect our findings from section 5.5. We are more likely to find SSSO buttons on the main page, and at a depth of 1, so a breadth-first search is deemed more appropriate. Among pages with the same depth, we can prioritize different URLs based on the likelihood of the web page containing an SSSO button.

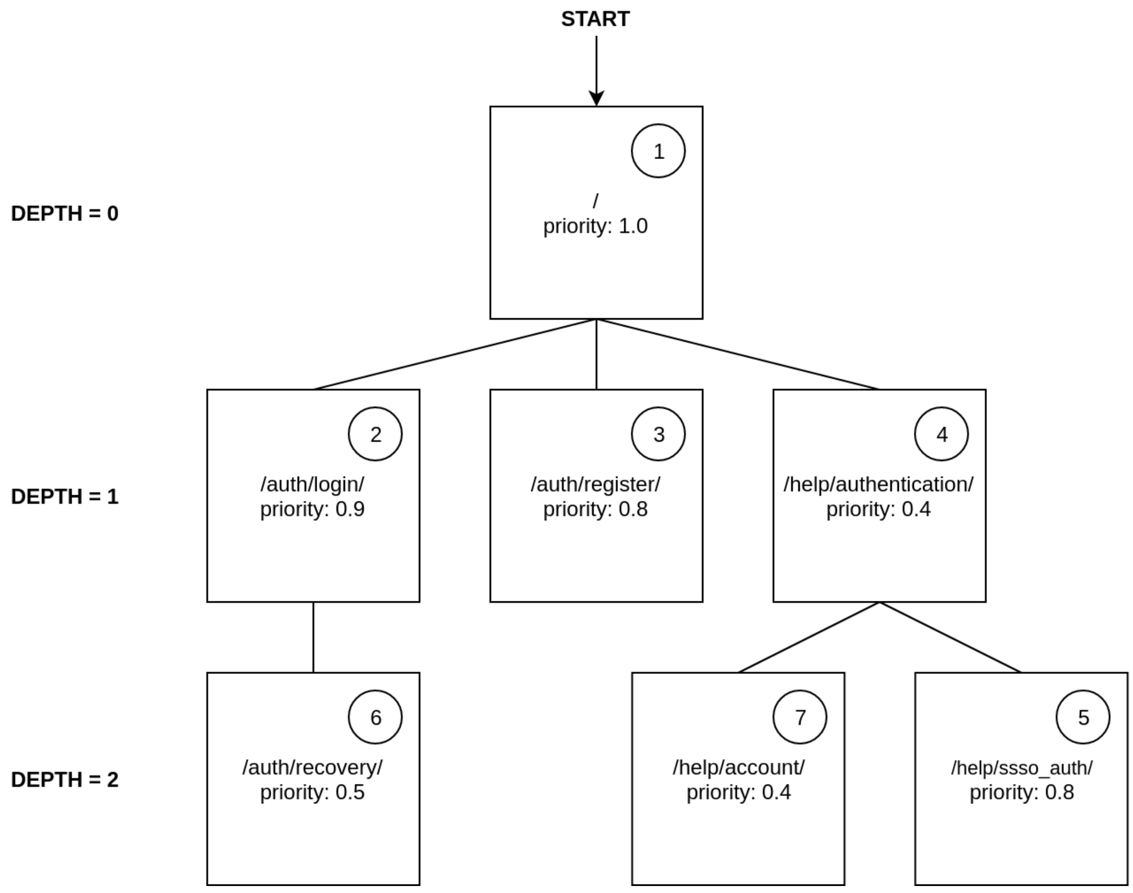


Figure 5.20: An example of our approach to scan a website. Every node is a different web page, whose priority is based on its URL. The order in which web pages are visited is noted in the circles in the top-right of every node.

We should avoid scanning pages that are the same as, or too similar to, the ones that we have already scanned. To enforce this measure, we need to use some form of memory that 'remembers' the pages that we have already visited. Using this memory, we can decide before and after loading the web page for the first time, whether or not we should scan the page.

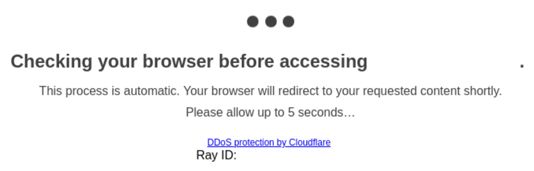
Two URLs can be considered different if their domains or paths are different. Vice versa, two URLs often result in the same document being served if their domains and paths are the same. However, due to the dynamic nature of web pages, the parameters of a URL might also completely change the page that is served to the user. In our situation, we are usually not interested in pages with dynamic content, but a small amount of websites map URLs to web pages using parameters (e.g. `www.example.com/?login`). This practice is not very common, and thus we deem two URLs equivalent if the domain and path match.

After we have loaded a web page for the first time, we can access the source of the web page. We can use simhashing to determine how similar two web pages are, as demonstrated by Manku, Jain, and Das Sarma [36]. This algorithm was developed at Google, and has been used by Google to detect duplicate web pages.

We presented our approach in an ideal situation where the landing page that we first visited was the main page. However, the landing page of a website might not be the main page, in which case, our approach might fail, due to one of the following reasons:

- some websites use a browser check to prevent bots from accessing their websites, which is presented to the user on their first visit to the website, as shown in figure 5.21
- some websites present the user with a landing page that lists all localized websites for different countries, as shown in figure 5.22
- some websites block the user from entering their websites, until the user confirms their age, agrees with the website's GDPR policies, or due to other legal issues

These situations are not common, and thus this problem has not been addressed in-depth. However, they might cause our approach to fail. An implementation of our approach should attempt to identify the main page first, and not simply start scanning a website from the landing page.



Checking your browser before accessing .

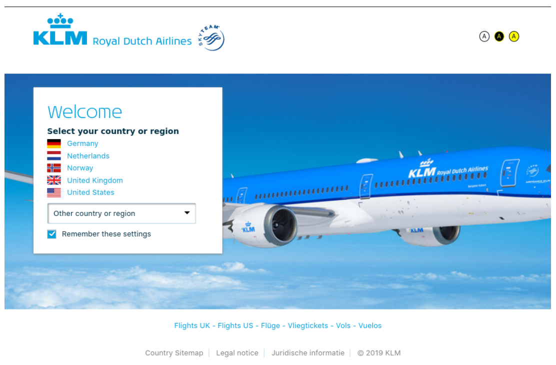
This process is automatic. Your browser will redirect to your requested content shortly.

Please allow up to 5 seconds…

DDoS protection by Cloudflare

Ray ID:

Figure 5.21: An example of a browser check being performed when a website is visited for the first time.



Welcome

Select your country or region

- Germany
- Netherlands
- Norway
- United Kingdom
- United States

Other country or region

✓ Remember these settings

Flights UK - Flights US - Flüge - Vliegtickets - Vols - Vuelos

Country Sitemap | Legal notice | Juridische informatie | © 2019 KLM

Figure 5.22: (KLM) The user needs to select their country or region from the menu, after which they are redirected to a localized website.

5.10.3 Conclusions

We have presented a general approach for discovering SSSO buttons on a website, that relies on the **search_page(url, found_urls, ssso_buttons)** method from section 5.9. During our search for SSSO buttons, we need to make several decisions about which web pages to scan, and in which order. One of the preconditions for our approach to work well, is that the scan needs to start on the main page. Not every landing page is the main page, and if so, our approach might fail.

5.11 Discussion and Final Conclusions

5.11.1 Evaluation

As a proof of concept, we added a module to the Shepherd framework that implements our approach for discovering SSSO buttons on a website, and used this module to scan multiple websites. Although the implementation of the presented approach generally works, its effectiveness is lessened due to several factors:

- errors due to instable nature of programmatic browsers and the Web platform
- the unstable waiting times for pages to load or change
- managing the browser's focus is difficult/unstable/error-prone

Although experiments were designed to investigate our hypotheses, the actual sample sizes of our experimental data are still too small to really form any conclusive evidence.

We did a small scan of 25 websites which provided SSSO authentication, and were able to detect SSSO buttons for 15 out of 25 websites (60%). Detection failed due to the following:

- a fatal exception occurred on 2 websites which ended the scan
- we did not find several steps due to the instability of having to wait for a page to load or change

5.11.2 Future Work

Besides resolving the factors that lessen the effectiveness of our implementation, and gathering more experimental evidence to support our findings, we identified several areas throughout this chapter where future work is required to further explore this problem:

- explore using visual recognition to recognize SSSO buttons
- find a better way to reference HTML elements in a persistent way
- investigate different strategies for pages to load
- investigate different strategies for pages to change
- further investigate methods to extract URLs from web pages
- improve the decision making for whether or not to scan a web page
- assigning priorities to discovered items, such as buttons or URLs
- handling landing pages, and identifying the main page

5.11.3 Conclusions

In this chapter, we approached the problem of identifying entry points of SSSO authentication processes. A general method for discovering SSSO buttons on any website. However, the problem is complex, due to the , and there are still problems left that need to be investigated.

Chapter 6

Automated Authentication via Social Single Sign-On

That it is possible to leverage Social Single Sign-On to automate the creation of the perspective of an authenticated user for a large number of web applications has been done in prior research by Zhou and Evans [11]. They identified the following natural steps: authenticating to the SSSO-provider, handling any possible post-login account creation and verification of the authenticated state. Their research was limited to the usage of Facebook as an identity provider. One of our goals is to automatically authenticate using any SSSO provider detected. In this chapter we will try to answer the following research question:

To what extent is it possible to automatically create the perspective of the authenticated user for any website or web application using any SSSO provider supported by the site in question?

Web applications are designed with humans in mind. Humans needed to authenticate themselves, humans need to interpret the questions asked, interpret the forms to fill and need to decide which buttons to click. This results in that this is much more difficult for machines than for humans. We investigated the authentication to an SSSO provider, the account creation procedures and to what extent it is possible and desirable to create accounts automatically and how to verify if the authenticated state has been created.

6.1 Overview

First we investigate the authentication process from the perspective of a human user and we will identify commonalities found within the authentication processes found for multiple SSSO providers. After identifying the commonalities we propose how we can apply these for many SSSO providers. And we will argue that the approach taken by Zhou and Evans [11] is reliable, relatively stable and adaptable to any SSSO provider. Next, we investigate the problem of additional account creation steps that are needed in a substantial amount of domains after authenticating to an SSSO provider to create the authenticated state. This problem was also identified by Zhou and Evans [11] and remained unsolved. Then we investigate how the authenticated state can be verified keeping in mind the restrictions from the Shepherd project and how the existing methods can be improved.

6.2 Commonalities and Differences in Authenticating to Different SSSO Identity Providers

6.2.1 Problem Description

In this section we will identify diversities and commonalities found in the authentication process used by multiple SSSO providers which should result in a general approach to this chapter's first sub-problem, automated authentication to any SSSO provider. Zhou and Evans [11] were able to automate the process of authenticating to the SSSO provider Facebook. Their approach was to follow the flow from the perspective of a human user. Our goal is to use any SSSO provider integrated by a web application following the same approach.

6.2.2 Observations

As initially observed, from a technical perspective the SSSO implementation can be very diverse. However from the users perspective there are commonalities between SSSO provideres and they and match the process as described in section 2.4. Example 6.1 describes the authentication process for Facebook and example 6.2 describes the authentication process for Google. The most conspicuous difference is the number of steps in which the user needs to enter the credentials.

The Authentication Process The similarities found in both the authentication flow for Facebook and Google makes us suspect that a general flow for authenticating with any SSSO provider exists. We conducted a small study on the 10 SSSO providers that Shepherd will support. We authenticated with each SSSO provider on 5 different domains to confirm the existence of such a generic authentication flow from the perspective of a human user. We found that:

- Facebook, VKontakte, Twitter, mail.ru, LinkedIn, ok.ru, Instagram and GitHub request the credentials in one step.
- Google and Yandex request the credentials in two steps.
- The user needs to explicitly grant access to the social media account when using Facebook, Google, VKontakte LinkedIn, ok.ru, Yandex, Instagram and GitHub SSSO when this is used for the first time for a web application. This is optional for Google.
- All SSSO providers redirect back to the web application upon successful authentication.

When the browser instance is already logged into the user's account for the SSSO provider, then it is not needed to supply the credentials for the SSSO account. The user only needs to grant explicit access to its social media account, figure 6.2 for VKontakte and figure 6.3 for LinkedIn. We found that this was the same for every SSSO provider.

Another scenario is that if the user granted access to the web application earlier and the current browser instance is logged into the user's account for the SSSO provider. The user only needs to click the SSSO button to authenticate and sign in.

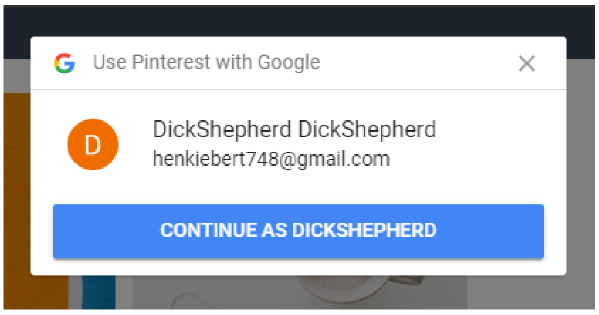
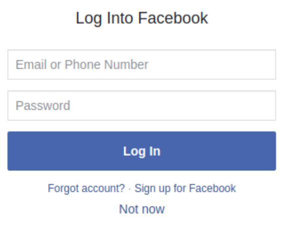
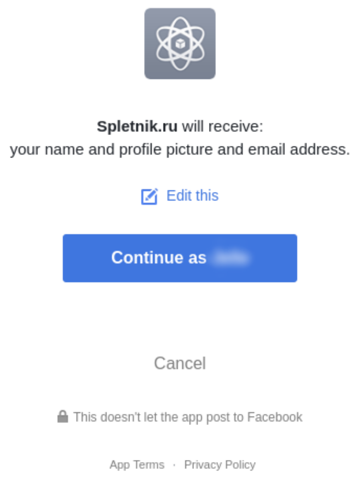


Figure 6.1: (Pinterest) A small pop-up shown when the user already has an account to the web application and is not logged in when Google SSSO is used.

Example 6.1
The flow in the authentication process for Facebook's SSSO is as follows when the current browser instance is not logged into the user's Facebook account.

The user visits Spletnik and clicks the Facebook SSSO button to start the authentication process.





Spletnik.ru will receive:
your name and profile picture and email address.

Edit this
Continue as [redacted]
Cancel
This doesn't let the app post to Facebook
App Terms · Privacy Policy

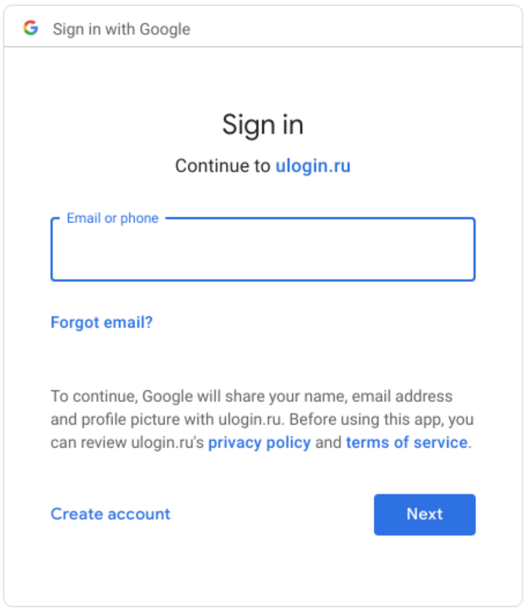
1. the browser is directed to Facebook's SSSO authentication page
2. the user supplies the email address or phone number and password and clicks a button to login
3. if this is the first usage of Facebook's SSSO for this web application, the user needs to grant explicit access to the user's Facebook account for the web application
4. the browser is redirected to the web application again if the all previous steps are successful

During the authentication process the URL starts with:

```
https://www.facebook.com/login.php?...
```

Example 6.2
The flow in the authentication process for Google's SSSO is as follows when the current browser instance is not logged into the user's Google account.

The user visits Stopgame and clicks the Google SSSO button start the authentication process.



Sign in with Google

Sign in

Continue to ulogin.ru

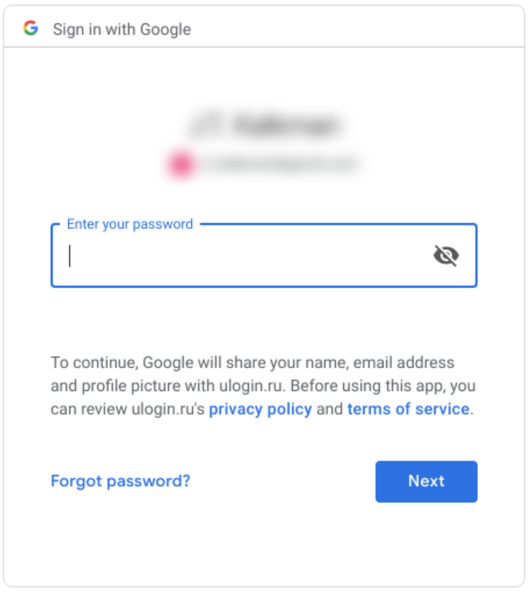
Email or phone

Forgot email?

To continue, Google will share your name, email address and profile picture with ulogin.ru. Before using this app, you can review ulogin.ru's privacy policy and terms of service.

Create account

Next



Sign in with Google

Enter your password

To continue, Google will share your name, email address and profile picture with ulogin.ru. Before using this app, you can review ulogin.ru's privacy policy and terms of service.

Forgot password?

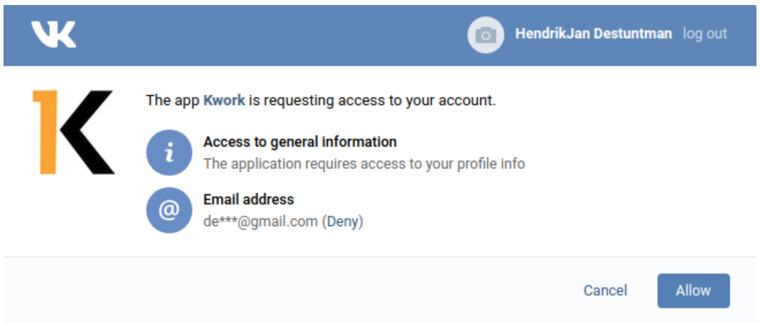
Next

1. the browser is directed to Google's SSSO authentication page
2. the user supplies the email address or phone number and clicks a button for the next step
3. the user supplies the password and clicks a button to login
4. optionally, if this is the first usage of Google's SSSO for this web application, the user needs to grant explicit access to the user's Google account for the web application
5. the browser is redirected to the web application again if the all previous steps are successful

During the authentication process the URL starts with:

```
https://accounts.google.com/signin/...
```

Some SSSO providers, among Google, show a small pop-up with a button to click to authenticate with the SSSO provider such as in figure 6.1. This only seems to be the case when the user already used this SSSO provider for this web application and is currently logged out of the web application. When this button is clicked, the user is authenticated to the web application without any further interaction needed.



HendrikJan Destuntman log out

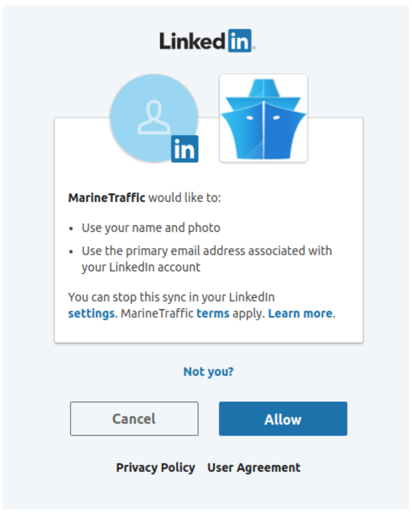
The app Kwork is requesting access to your account.

Access to general information
The application requires access to your profile info

Email address
de***@gmail.com (Deny)

Cancel Allow

Figure 6.2: Granting explicit access to the web application with VKontakte SSSO.



MarineTraffic would like to:

- Use your name and photo
- Use the primary email address associated with your LinkedIn account

You can stop this sync in your LinkedIn settings. MarineTraffic terms apply. Learn more.

Not you?

Cancel Allow

Privacy Policy User Agreement

Figure 6.3: Granting explicit access to the web application with LinkedIn SSSO.

Interaction with the SSSO Provider In modern web applications everything can be made clickable, buttons are not declared in the HTML with regular button tags, CSS is used for complex styling and JavaScript can trigger all kinds of events to change the state of the application. This is confirmed by inspecting the elements used for the interaction with the authentication pages or iframes used by SSSO providers. For example, Google uses a form with regular inputs for the email address, however for the submit buttons div-elements are used. Example 6.3 shows the complexity of an element that needs to be clicked. In this particular case identification is easy because of the unique id-attribute. We suspect that each SSSO provider has it own design and that no generic selectors can be found. Luckily this covers all instances for the usage of a SSSO provider.

To confirm this we investigated all elements for which it is needed to interact with when using SSSO for each SSSO provider that will be supported by Shepherd. We found that:

- Regular input elements are used to enter data.
- The type element of an input element is not always correctly used. For example an input with type email that can also accept a telephone number. Or an input that requires an email address that has the type text.
- Clickable elements can be button, input or div elements.
- Buttons mostly have the correct type attribute.
- Class attributes are not reliable for selecting the elements. These can be used for other elements too and could result in selecting the wrong elements. Modern CSS mostly uses reusable classes for multiple components.
- The ID attribute can be a good candidate for selecting a specific element, however if generated IDs are used they can change. A good indicator is that when numbers are used

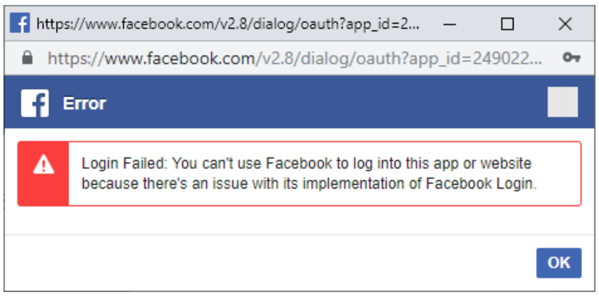
Example 6.3
An HTML div element used as a button to submit the user name in the authentication process for Google SSSO.

```
<div role="button" id="identifierNext" class="U26fgb O0WRkf zZhnYe e3Duub
  C0oVfc nDKKZc DL0QTb M9Bg4d" jscontroller="VXdfxd" jsaction="click:cOuC
  gd; mousedown:UX7yZ; mouseup:lbsD7e; mouseenter:tfO1Yc; mouseleave:Jyw
  Gue; focus:AHmuwe; blur:O22p3e; contextmenu:mg9Pef;touchstart:p6p2H;
  touchmove:FwuNnf; touchend:yfqBxc(preventMouseEvents=true|preventDefaul
  t=true); touchcancel:JMtRjd;" jsshadow="" jsname="Njthtb" aria-disabled
  ="false" tabindex="0">
  <div class="Vwe4Vb MbhUzd" jsname="ksKsZd"></div>
  <div class="ZFr60d CeoRYc"></div>
  <content jsslot="" class="CwaK9">
    <span class="RveJvd snByac">Volgende</span>
  </content>
</div>
```

or the IDs are made up of random characters, these could be generated and could change with every request and are thus not reliable as CSS-selectors.

- Language based selection for keywords is not reliable, for example Google adjusts the language presumably on IP-range or browser or OS preferences.

Based on our findings we think it is best to create CSS selectors for each HTML element that is needed to interact with and for each SSSO provider individually. It is recommended to validate the found selectors when using Shepherd because the implementation by the SSSO provider is presumably under active development.


https://www.facebook.com/v2.8/dialog/oauth?app_id=2...

https://www.facebook.com/v2.8/dialog/oauth?app_id=249022...

Error

Login Failed: You can't use Facebook to log into this app or website because there's an issue with its implementation of Facebook Login.

OK

Figure 6.4: (SlideShare) An example of a broken Facebook SSSO implementation

Broken SSSO Implementation We encountered multiple cases of broken Facebook SSSO implementations, figure 6.4. The browser is redirected to the correct URL of the login page or iframe, however the correct elements that are needed to interact with could not be found and a message is shown that the implementation by the web application is incorrect. There is also no automated redirection back to the web application.

We did not experience any other broken SSSO providers besides Facebook, we conclude for now that when the elements that are needed to interact with are absent or an error message is displayed, the SSSO implementation is flawed.

6.2.3 Conclusions

In this section, we investigated authenticating to a SSSO provider from the perspective of a human user. We found that from a human perspective a general flow can be described for multiple SSSO providers, that it is needed to precisely identify the elements to interact with during the authentication process and that broken SSSO implementations can be detected when authenticating to an SSSO provider.

6.3 Automating Social Single Sign-On

In the previous section we investigated authentication to an SSSO provider from a human perspective. In this section we will discuss how we realized our findings in a design and implementation that resulted in a module for Shepherd to authenticate using SSSO. To prove that these methods work we tested this module for each supported SSSO provider on 10 domains.

6.3.1 The SSSO Login Module

In the previous section we identified two possible approaches for the authentication flow. First is to authenticate to an SSSO provider and then visit the domain under investigation, the browser instance is already logged in. Second is to authenticate to the SSSO provider when using SSSO to authenticate to the web application, the browser instance is not logged in yet.

Design Decisions The first approach can be realized by authenticating to each SSSO provider first and store all cookies. When an SSSO provider is searched on a domain, then at least the authentication cookies for that SSSO provider should be stored in the browser. With this approach we suspect the following difficulties:

- It is possible that the authentication cookies for the SSSO provider expire and thus expiring the session.
- Additional tests need to be added to verify if the session with the SSSO provider has not expired.
- Presumably more cookies are stored which could affect security research on cookie properties and the results could be less pure caused by unnecessary cookies.

The second approach always requires to authenticate to the SSSO provider for each domain. This results in more interaction when authenticating, however compared to the detection of SSSO this is a minimal penalty in scanning speed. Other considerations are:

- Only the required cookies are stored in the browser.
- This way we make sure the session for the SSSO provider has not expired.
- This makes the implementation less complex, it is not needed to determine what flow to use or take multiple scenarios into account.

We also decided to start each attempt to authenticate to a domain using SSSO in a new browser instance. The result of a login attempt can be a set of cookies stored by the browser. When a new browser instance is used for every attempt, the set of cookies will not be cluttered with cookies from previous attempts. The current implementation authenticates to all SSSO providers that are detected, this behavior can be easily changed. After the last attempt, the browser instance is kept alive because the subsequent module tries to handle any account creation procedures and the browser should be left in an untouched state in order not to abort these procedures.

Module Results The SSSO login module uses the results of the SSSO detection module. The SSSO detection module stores clickable elements in an action chain that trigger the entry point for the SSSO authentication to show. This action chain is used as the entry point for this module to start the authentication to an SSSO provider and also illustrates how modules can pass information to each other and build on top of each other. The SSSO login module stores the entire set of cookies for each SSSO provider if it successfully authenticated. This cookie set can then be used by a subsequent module to recreate the authenticated state.

Implementation This module relies heavily on functionality implemented in the core classes of Shepherd which results in lean classes with a clear responsibility and a minimum amount of code. The generic authentication flow maps easily to a corresponding architecture. An SSSO login base class implements the authentication flow and provides reusable code for subclasses for each SSSO provider. This provides the option to easily extend the Shepherd framework for other SSSO providers or to make changes to them and existing classes could be used as a template. A simplified UML class diagram of this module is provided within the appendix.

6.3.2 Testing the SSSO Login Module

To have any metrics on the performance and reliability of the SSSO login module and to find weaknesses, we tested this module for each SSSO provider supported by Shepherd on ten domains. We used a random sample from the top 10,000 from the Alexa ranking and continued until we had enough samples for each SSSO provider. If a domain supports multiple SSSO, these were all taken into account if needed. We used Shepherd to scan for domains supporting SSSO and to authenticate to the detected SSSO providers. We manually verified the results using the screenshots and if needed we authenticated manually and compared the results, for example when the screenshots were inconclusive.

Results We managed to find 10 different domains for 9 of the 10 SSSO providers. We experienced difficulty to find usage of Instagram SSSO for which we only managed to find 4 domains in the given time. We found that:

- In all cases an SSSO button was found, Shepherd tried to login.
- Incorrect setup of SSSO was found for Facebook, Google and LinkedIn, each in one instance and for ok.ru in two instances.
- In two cases wrong SSSO buttons for Yandex were identified. These links were presumably to share content or used in clickable ads. However Shepherd tried to login into the account and this resulted in an error message, which we found erratic behavior.
- LinkedIn and mail.ru detected suspicious behavior. Mail.ru uses CAPTCHAs in this case and for LinkedIn it was needed to provide a verification code sent to an email address. Both these cases were experienced only once we could login in subsequent domains for these SSSO providers.
- We did not encountered any blocked accounts, however we can give no assurance when the amount of domains in a scan increases.
- If we do not take falsely identified SSSO buttons, that is another module, broken implementations and detection of suspicious behavior into account, this module managed successfully login in 85 cases.

The full results of this test can be found in the appendix.

6.3.3 Conclusions

We are convinced that this module provides a good basis for use in the Shepherd project to automate the authentication using SSSO. This module is easily extensible for other SSSO providers. However we see room for improvements and more features, such as:

- Adding detection for errors or wrong configuration during the authentication process. Depending on the SSSO provider this could be done on patterns in the URL or searching for known errors in the HTML DOM.
- Detection of CAPTCHA usage or other countermeasures for suspicious behavior.
- Extending the modules with the option to first authenticate to the SSSO provider, followed by deduction of the cookies used to authenticate to the SSSO provider and adding detection of and switching to the flow when this flow for SSSO is followed. However this makes the SSSO login module more complex.

6.4 Account Creation

6.4.1 Problem Description

The second sub-problem in this chapter is handling additional account creation actions, such as filling in registration forms or simply that the user needs to agree to the terms and conditions. For humans this is an easy task however for machines this is entirely different. First of all, these processes are not designed to be easy solvable for machines and secondly, counter measures are taken in web application to prevent bots from creating accounts easily that could be used for malicious purposes.

In a substantial amount of cases signing in with the SSSO provider and granting the web application access to user's account is sufficient to create a new account. Zhou and Evans [11] found that in about 44 % of the domains they investigated additional actions were needed to actually create an account, and also the authenticated perspective.

Zhou and Evans [11] also mentioned the complexity of the post-login account creation procedures. Their solution was to only process forms whereby all pre-populated fields were left untouched and to process fields in a specific order. For fields that required input they used heuristics, however no further information was given on this. Their solution seems to be an underestimation of the problem of processing forms, which can be a meticulous job, however they managed to create accounts automatically in 66 % of the cases where additional account creation was needed. Their research focused on the top 20,000 in the U.S. and as such we suspect that they did not took multiple, or at least not much, other languages into account. They investigated why they failed in the account creation procedures however they did not tried to find solutions. Presumably these were deemed unsolvable which is not explicitly stated.

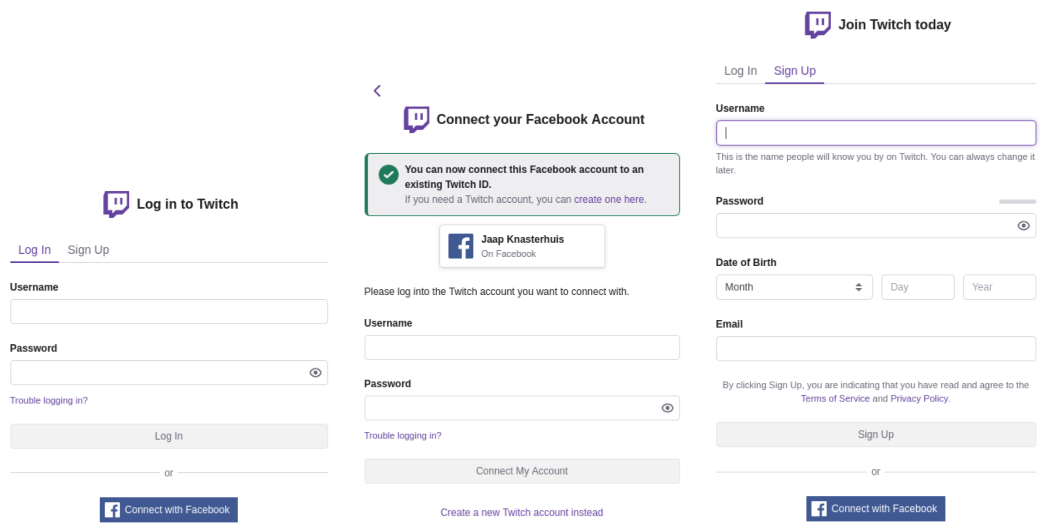
In this section, we will verify this percentage, investigate the account creation procedures, what automated account creation countermeasures can be expected and the prevalence.

6.4.2 Observations

The research from Zhou and Evans [11] dates back to 2014 and therefore we should at least verify and possibly update the percentage of domains that require additional account creation handling.

Account Creation Needed To investigate if their observations are still valid we conducted a small study on 225 domains that integrate SSSO and were picked randomly from the top 10,000 from the Alexa ranking. We authenticated manually and performed all steps needed to

Example 6.4
The case that no account is linked to the SSSO account on Twitch.


Log in to Twitch

Log In Sign Up

Username

Password

Trouble logging in?

Log In

or

Connect with Facebook

Connect your Facebook Account

You can now connect this Facebook account to an existing Twitch ID.
If you need a Twitch account, you can create one here.

Jaap Knasterhuis
On Facebook

Please log into the Twitch account you want to connect with.

Username

Password

Trouble logging in?

Connect My Account

Create a new Twitch account instead

Join Twitch today

Log In Sign Up

Username

This is the name people will know you by on Twitch. You can always change it later.

Password

Date of Birth

Month Day Year

Email

By clicking Sign Up, you are indicating that you have read and agree to the Terms of Service and Privacy Policy.

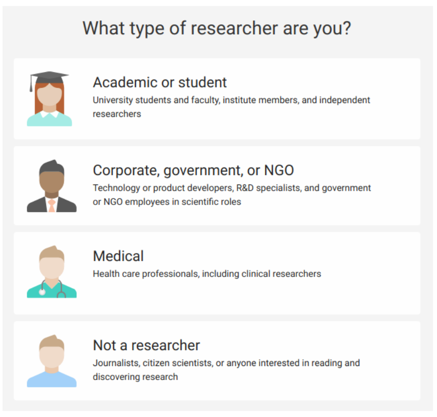
Sign Up

or

Connect with Facebook

Directly after redirection back to the web application the user is shown that no account is linked to the credentials provided by the SSSO provider and the user needs to link an account or create a new account.

Example 6.5
Account creation steps without the use of HTML forms on ResearchGate and Blablacar. The user is supposed to select one element by clicking it.

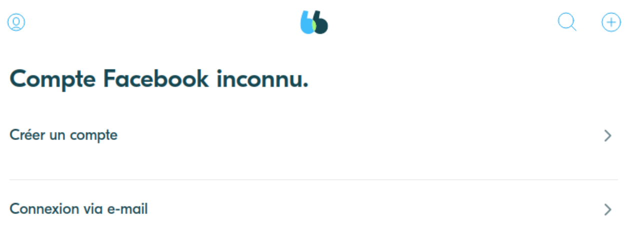

What type of researcher are you?

Academic or student
University students and faculty, institute members, and independent researchers

Corporate, government, or NGO
Technology or product developers, R&D specialists, and government or NGO employees in scientific roles

Medical
Health care professionals, including clinical researchers

Not a researcher
Journalists, citizen scientists, or anyone interested in reading and discovering research


Compte Facebook inconnu.

Créer un compte

Connexion via e-mail

stay logged in. For 100 out of 225 domains it was required to perform additional actions to create the authenticated state. This confirms the findings of Zhou and Evans [11], 44 %.

Account Creation Procedures We conducted a small study on the 100 domains requiring additional account creation which we identified earlier. For this we authenticated manually using SSSO and created accounts for all domains. We encountered the following scenarios:

- When the user is redirected back to web-application a message is shown that it is needed to link an account or create a new account, thus following the regular account creation procedures such as in example 6.4. In 28 out of the 100 domains we experienced this scenario.
- In one or multiple steps in the account creation procedures HTML forms are not used. The user needs to click a specific element, which can be anything, to proceed, as in example 6.5. We experienced these design solutions in 8 out of 100 domains.
- 24 out of the 100 domains use CAPTCHAs and Google's reCAPTCHA is mostly used, example 6.6. This CAPTCHA system seems to adjust its behavior when it is used often in a short amount of time by one computer.
- Verification of the supplied email address by clicking a link in an email or entering a code and verification by SMS is experienced both in 7 out of 100 cases, example 6.7. This number seems a bit low to us, however we did not verified if an email was sent to the email addresses that belong to the SSSO account, presumably temporary accounts are created in a number of cases.
- Some web-applications require specific data such as a telephone number. This poses a barrier because the telephone numbers could be country specific, example 6.8. Supplying random phone numbers could result in an unknown person being called or send text messages by an unknown organization. This is something which conflicts with our ethical standards. The same holds true for credit card numbers or bank account. In some cases it is required that the user already needs to have an existing account or must be known by the organization, example 6.8. In an initial observation we noticed the request for a social security number and this is also something we do not wish to supply or guess. We found that for 3 domains a telephone number was required, in 6 cases institutional credentials were needed and in one case a bank account or credit card number was required.
- For one domain it was needed to register via an app first.
- Other observations are that whatever we tried to register, it just seems to fail without any clear reason or the registration process was broken, it was unclear what data to enter even when the website was translated. We experienced this in 8 cases.

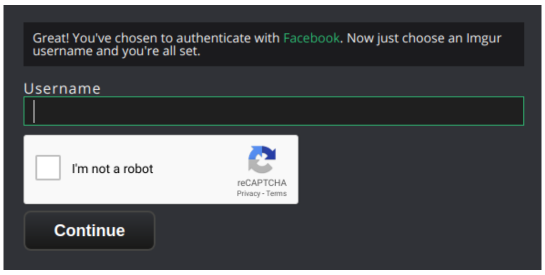
The full results of this short survey can be found in the appendix.

6.4.3 Registration Forms

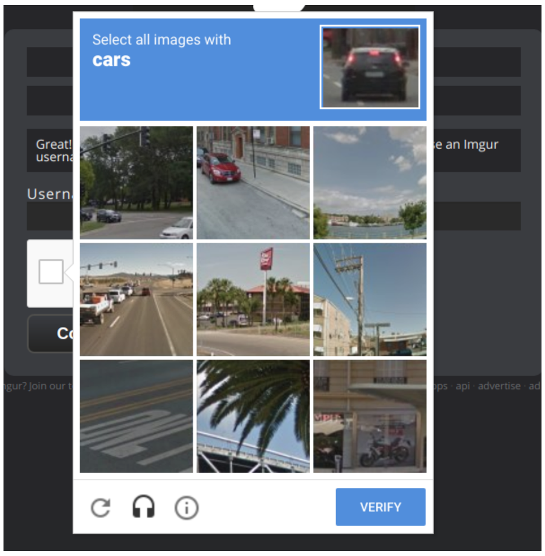
Forms seem to be the only possible steps that seem suitable for automated processing at a first glance. However, when we performed the quick survey on account creation procedures in 6.4.2, we observed the following issues which could hinder automated processing of forms:

- Not all forms have a submit button or not a button that is logically the submit button of a form when it is inspected with the developer tools in a browser. In those cases the forms behavior is presumably controlled by JavaScript.

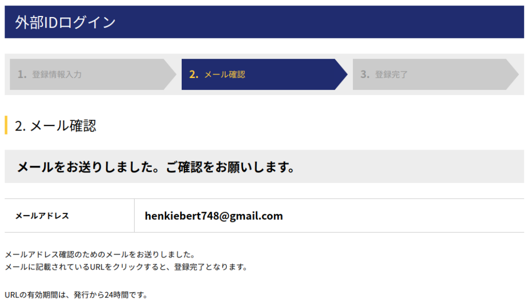
Example 6.6
Google's reCAPTCHA when suspicious behaviour was detected on Imgur


Great! You've chosen to authenticate with Facebook. Now just choose an Imgur username and you're all set.

Username


Select all images with
cars

Example 6.7
Email verification on Goo and verification by SMS on Carousell.


外部IDログイン

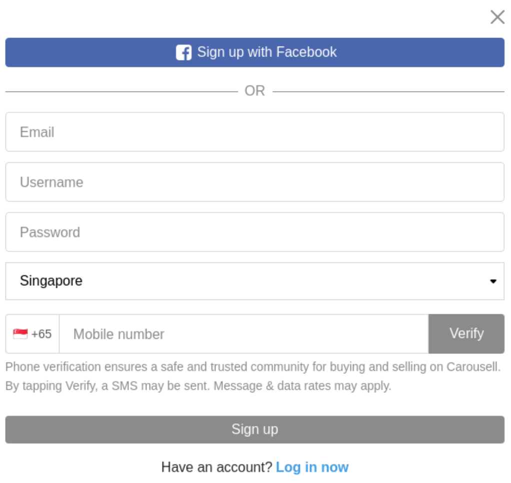
2. メール確認

メールをお送りしました。ご確認をお願いします。

メールアドレス	henkiebert748@gmail.com

メールアドレス確認のためのメールをお送りしました。
メールに記載されているURLをクリックすると、登録完了となります。

URLの有効期間は、発行から24時間です。

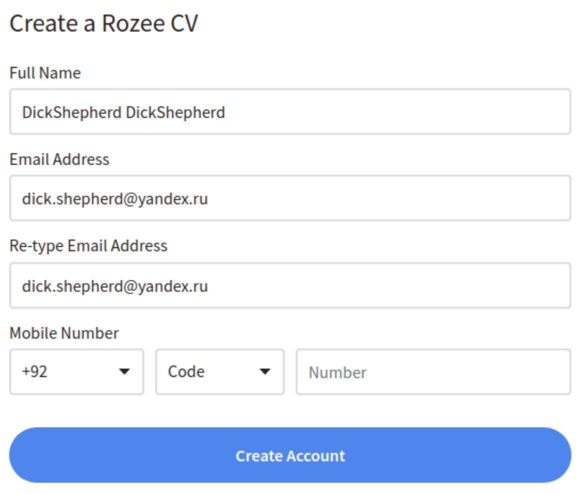


Phone verification ensures a safe and trusted community for buying and selling on Carousell. By tapping Verify, a SMS may be sent. Message & data rates may apply.

Have an account? Log in now

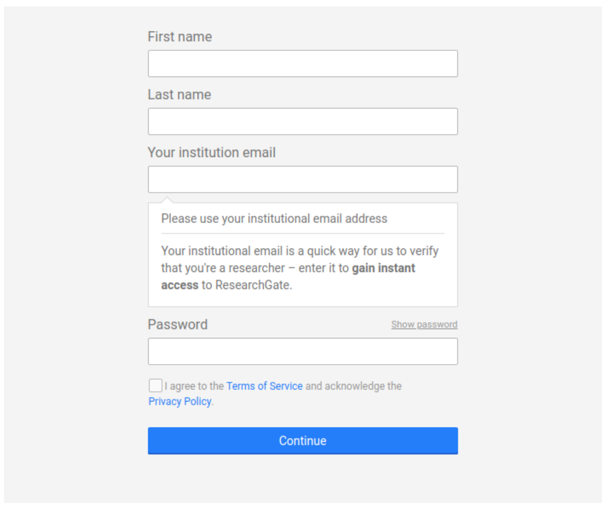
Example 6.8

A form which requires a telephone number in a specific format on Rozee and a form whereby valid institutional credentials are needed on ResearchGate.



Create a Rozee CV

Full Name	DickShepherd DickShepherd
Email Address	dick.shepherd@yandex.ru
Re-type Email Address	dick.shepherd@yandex.ru
Mobile Number	+92 / Code / Number



First name

Last name

Your institution email

Please use your institutional email address

Your institutional email is a quick way for us to verify that you're a researcher – enter it to **gain instant access** to ResearchGate.

Password

I agree to the Terms of Service and acknowledge the Privacy Policy.

- Forms might be controlled by JavaScript to show, hide or create new input elements when the form is filled and data might be loaded asynchronously.
- Input elements are used outside form elements.
- Input elements have no label attached with the for attribute of a label.
- Forms can make use of HTML5 form validation or custom solution using JavaScript and CSS classes to visualize the errors for humans.
- Regular expression can be used in a pattern attribute of an input for validation.
- Front-end frameworks can provide complex styling and behavior and make use of pseudo-elements to create elements for visual purposes only, especially for selects, radio buttons and checkboxes.
- Form inputs can be without any type attribute or an incorrect type attribute. For example the type attribute has the value text, however the form validation requires an email address.
- Name attributes are used, not used or could be empty in input elements.
- The Web Content Accessibility Guidelines (WCAG) do not seem to be followed and the WAI-ARIA attributes do not seem to be widely used in forms.

For most humans these issues do not pose any barrier however our observations could hinder interpretation of a form by a machine. We devoted section 6.5 almost entirely to the problem of form processing.

6.4.4 Conclusions

In this section we investigated the account creation procedures that could be required to authenticate to a web applications after authenticating to an SSSO provider. 41 out of the 100 domains we investigated are not hindered by the usage of CAPTCHAs, SMS or email verification, the need for specific data or complex or difficult interaction for the account creation procedures. CAPTCHAs bypassing methods exist [37, 38, 39] and might be usable within the Shepherd project. However, should it not be respected that automated account creation in those cases is not desired? We also inspected forms to decide whether or not these might be suitable for automated processing by a machine, we suspect that this is only possible in a limited number of cases. Overall we are convinced that automated account creation on a large number of unknown web applications will be difficult to solve.

6.5 Processing Forms with Formasaurus

We decided to only handle forms that are used in the account creation processes based on our findings in 6.4. We also suspect that classifying forms and form fields and identifying the correct form to fill in any given language is a too big problem to solve in the given time. We opted to find and use an existing tool to at least experiment with automated form processing. The only candidate tool we could find was the Python library Formasaurus.

6.5.1 Formasaurus

Formasaurus uses machine learning to classify forms and form fields and the developers state that they successfully classify 88.9 % of the forms and that fields are classified correctly for 76 %. Formasaurus is trained on at least 1000 forms, mostly randomly selected from the Alexa top 1 million ranking. How Formasaurus classifies forms and fields is explained on the project page. Formasaurus' developers mention that usage of CSS and JavaScript to change behavior in forms limit its capacities which is to be suspected because the detection and classification is based on HTML source. Usage of machine learning instead of keywords or other heuristics could help to overcome the language barrier for processing forms that can be used with any language.

6.5.2 Shepherd's Account Creation Module

In this section we shall explain our choices made for the integration of Formasaurus in the Shepherd project. A simplified UML class diagram for the account creation module is provided in the appendix.

Scanning for Forms In Shepherds account creation module we use Formasaurus to detect and classify forms and their fields. In the survey in section 6.4 we observed that in most cases no more than 3 steps are performed in the account creation steps. However it is also needed to take form fields that appear after interaction with the form into account. We decided that the number of scans for forms should be configurable and that 10 scans is an appropriate number to start with.

Processing a Form Each form that is detected in a scan is classified by Formasaurus. A configuration file provides control over the form classification that are processed. A Formasaurus form object only provides information about the form, it does not provide the actual form element. It is essential that the correct form element is found in the HTML, this is done by trying to create an as specific as possible CSS selector based on the information in a Formasaurus object. This selector is also used to provide an explicit boundary for the selection of form fields, these should not be outside the form to prevent processing the wrong elements. When a form element can be found, this module tries to find all form fields and tries to determine the submit element.

Processing a Form Field First it is needed to construct an as specific as possible selector for each form field. If an element can be found this module tries to collect as much information from the Formasaurus object and the HTML source for correct processing of an input, such as:

- The type of the tag.
- The type attributes if it is an input tag, for example radio or password.
- The value of the name attribute.
- The value of the ID attribute.
- If a form field is an input, has it a value set?
- The action that should be performed, clicking, sending text or selecting a value.
- If this element is classified as a submit element by Formasaurus.

To control which classes of form fields should be processed a configuration file is used. This configuration file also contains the data to enter for each class of form field. Just as Zhou and Evans [11] we leave pre populated fields untouched.

Submitting a Form Formasaurus tries to classify an element as a submit element, however there is no guarantee that it can find such an element. We decided that if no submit element is found by Formasaurus, to search for a submit element based on keywords. Difficulty with this is that this needs a survey on a large number of registration forms in a large number of languages. This also means that the chance of clicking the wrong element increases, especially when multiple forms are present. However this is just a fallback and a best effort attempt.

Module Results This module requires that the state of the browser instance is untouched directly after successful authentication to the SSSO provider in order not to abort or disturb the account creation process. The account creation module does not store any information for other modules. For each scan for forms this module logs what form classifications are detected, which could be useful for testing purposes.

6.5.3 Testing and Tuning the Account Creation

We saved 100 registration forms picked from domains requiring additional account creation handling that we identified earlier. This could be more extensive forms or just simple forms where the user only needs to agree with the terms and conditions. We used this module to detect and process these forms. We observed if the form are correctly detected, what classification is assigned by Formasaurus and visually observed if the forms and the fields are correctly processed. Goals of this analysis are:

- Determine how difficult this problem is.
- Finding the most suitable configuration for the form classes that should be processed. This is also to prevent wrong forms being processed, search forms should not be processed for example.
- Clearly find limitations in our approach.
- Finding any weaknesses or bugs in this module.

Form Classification Results How Formasaurus classifies registration forms can be of use on how to configure this module. Processing less form classifications reduces the risk of processing the wrong form, however this also increases the risk of missing forms. Only 39 out of the 100 forms are correctly identified as registration form, a number that does not surprises us given the diversity in the forms. This module also logs if Formasaurus detected CAPTCHAs, however this failed in this in all cases.

Classification	Number of Forms
registration	39
Form not detected or not sure which form	17
other	14
login	8
join mailing list	6
contact/comment	7
password/login recovery	7
order/add to cart	1
search	1

Table 6.1: Results from the classification of forms.

An improvement in the form processing algorithm might be processing all forms, whereby forms with the highest likelihood of being a registration form first, thus following the order from table 6.1.

Form Processing Results Our findings from the visual observation are:

- Check boxes that are checked should not be unchecked.
- When multiple forms are found, the wrong form could be processed. A much more advanced method is needed to correctly identify the form that needs processing. A possible solution could be finding the element that actually has focus using JavaScript.
- Custom form elements from front-end frameworks are difficult to assess if these are correctly processed due to CSS and pseudo elements.
- A substantial amount of fields are misclassified, which strengthen our conclusions from 6.4.
- The method to find the correct submit element for a form when Formasaurus failed to find one needs more refinement.

6.5.4 Conclusions

This module provides only a proof of concept solution for automated form processing and our findings confirm our suspicions from section 6.4, automated form processing is a difficult problem to solve and a study on its own. Experimenting and trying to improve this module might provide mediocre results, however we leave this as future work. This also makes us wonder how Zhou and Evans [11] reached a success rate of 66.4 %. Their research was done in 2014 and we have no conclusive data to provide an answer. Maybe then more regular forms and less automated account creation countermeasures were used.

6.6 Verification of the Authenticated State

6.6.1 Problem Description

In this section we investigate the options to verify the authenticated state in order to decide on the best approach for solving this problem or to improve current methods.

Currently Shepherd uses the following methods to verify the logged in state:

- Searching for the existence of certain keywords, such as user credentials on the landing page and the current page.
- Searching for the existence of a logout element on the landing page and the current page.
- Searching for password fields.

The success rate of this implementation is around 60 %. Any improvements in verifying the authenticated state greatly increases the amount of domains that can be assessed on security.

There are some aspects of the Shepherd project that need to be taken into account when deciding on the verification methods:

- The verification methods need to be repeatable, as these will also be used to verify if the user is authenticated when Shepherd tries to detect the set of authorization cookies.
- This repeatability implies that the methods also needs to be fast.
- The verification methods should not depend on clicking any elements that could trigger navigation or other unknown side effects that could alter the, logged in, state of the web application.

The same approach to verify the authenticated state was taken by Zhou and Evans [11]. They additionally searched the set of cookies in the browser for hints if the user is logged in, such as usernames, email addresses or profile images.

6.6.2 Observations

To find improvements for the verification methods we manually investigated 100 domains randomly selected from the top 10,000 from the Alexa ranking that at least integrate one of the SSSO providers supported by Shepherd. We authenticated to all these domains using SSSO and, if needed, followed the account creation procedures. Then we examined each domain for suitable properties to verify the logged in state.

Cookies Searching the cookie set for any indication of the authenticated state could be a good candidate. One limitation is that Shepherd also uses the verification methods to deduce the set of cookies that create the session. In that process cookies are loaded and removed in the browser instance to calculate this set of authentication cookies.

We found that 20 out of the 100 web applications store information in a cookie that could indicate that the user is authenticated. To make sure this is an appropriate method we also checked if the information could not be found after we logged out of the application, this seems to be correct. In example 6.9 is the user's email address clearly visible.

We also noticed that the value for a cookie attribute can be stored using percent-encoding, such as in example 6.10. When searching for any hints on the authenticated state this must be taken into account. More information on percent-encoding is provided by The Internet Society.

Example 6.9
The attributes of a cookie stored by a browser after authenticating to MarineTraffic with Google SSSO. The email address indicates the authenticated state.

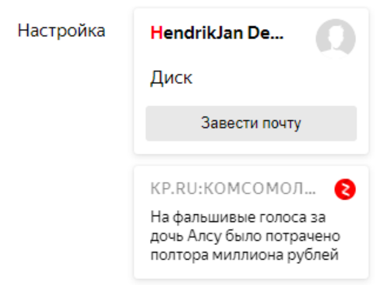
```
name: AUTH
value: EMAIL=destuntmanhendrikjan@gmail.com&CHALLENGE=CPQvn8l1gT8LrCxP...
domain: www.marinetraffic.com
expires: session
path: /
httpOnly: true
secure: false
sameSite: unset
```

Example 6.10
Example of an email address stored with percent encoding in a cookie value.

```
user: %7B%22display_name%22%3A%22HendrikJan%20DeStuntman%22%2C%22...
```

The HTML DOM Web applications often show the users name somewhere to create a more personalized user experience. This can be permanently visible such as in figure 6.5, or after clicking on or hovering over an element such as in figure 6.6. The users credentials can also be found in the HTML source and not be visible to the user as in figure 6.7. The credentials can also be placed in attributes of HTML elements, for example the data attribute used in a button to pass data to JavaScript when it is clicked or the value attribute of an input element.

It is good practice that all the users data is removed from the DOM when the user is logged out of the web application and thus it is possible to use this behavior to determine if the browser instance is in an authenticated state or not.



На фальшивые голоса за дочь Алсу было потрачено полтора миллиона рублей

Figure 6.5: Visible user name in Yandex.

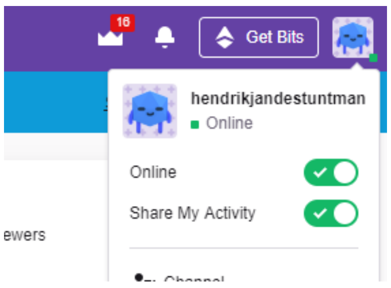
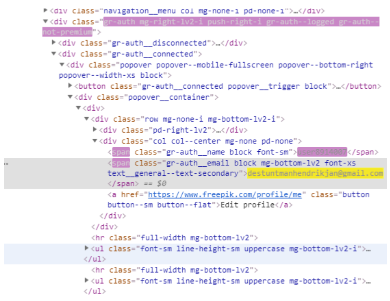


Figure 6.6: The username only visible after user interaction on Twitch.

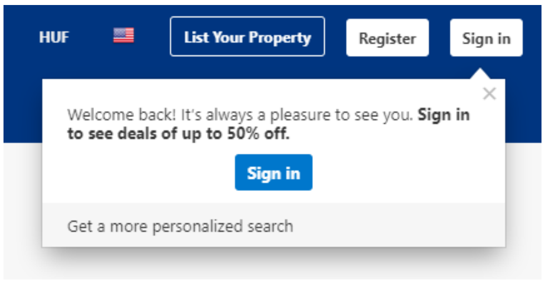


```
▸<div class="navigation__menu col mg-none-i pd-none-i">…</div>
▾<div class="gr-auth mg-right-lv2-i push-right-i gr-auth--logged gr-auth--
not-premium">
  ▸<div class="gr-auth__disconnected">…</div>
  ▾<div class="gr-auth__connected">
    ▾<div class="popover popover--mobile-fullscreen popover--bottom-right
    popover--width-xs block">
      ▸<button class="gr-auth__connected popover__trigger block">…</button>
      ▾<div class="popover__container">
        ▾<div>
          ▾<div class="row mg-none-i mg-bottom-lv2-i">
            ▸<div class="pd-right-lv2">…</div>
            ▾<div class="col col--center mg-none pd-none">
                <span class="gr-auth__name block font-sm">user8914007</span>
                <span class="gr-auth__email block mg-bottom-lv2 font-xs
                text__general--text-secondary">destuntmanhendrikjan@gmail.com
                </span> == $0
                <a href="https://www.freepik.com/profile/me" class="button
                button--sm button--flat">Edit profile</a>
              </div>
            </div>
            <hr class="full-width mg-bottom-lv2">
          ▸<ul class="font-sm line-height-sm uppercase mg-bottom-lv2-i">…
          </ul>
            <hr class="full-width mg-bottom-lv2">
          ▸<ul class="font-sm line-height-sm uppercase mg-bottom-lv2-i">…
          </ul>
```

Figure 6.7: The username only visible in the HTML DOM on Freepik.

When searching the HTML DOM for user data, script and style elements should also be taken into account. In that cases node.textContent instead of node.outerHTML or node.innerHTML should be used in JavaScript. An explanation on this is provided by The Mozilla Foundation. All attributes of an element also need to be taken into account, this is easily covered with node.outerHTML.

We found an email address in the DOM on 13 domains, the surname and given name in 55 and 70 domains respectively and a combination of surname and given name on 52 domains. In theory 75 out of the 100 domains can be verified using this method.



Welcome back! It's always a pleasure to see you. **Sign in to see deals of up to 50% off.**

Figure 6.8: (Booking) A log in element.

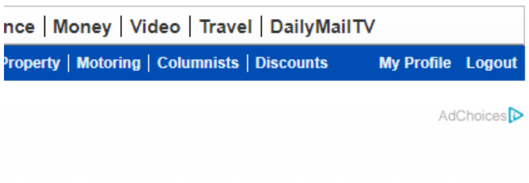

nce | Money | Video | Travel | DailyMailTV
roperty | Motoring | Columnists | Discounts
My Profile
Logout
AdChoices

Figure 6.9: (Daily Mail) A logout element.

Login/Log-out Elements Web application often show a login element on the landing page when the user is not logged in, figure 6.8, and a logout element when logged in, figure 6.9. When using specific elements to draw a conclusion on the authenticated state, the visibility should also be taken into account. A common approach in web development is to have a login element always present in the DOM, when logged in it is present in the DOM however not visible to the user. The visibility should be taken into account in this case. We found that a login element is shown on 72 domains and a logout element on 11 domains. In theory 73 out of the 100 domains can be verified using this method.

Automated Redirection Some web applications immediately redirect the user to a specific account page when the landing page is requested and the user is logged in. This behavior can be found for example on for example on Doodle, figures 6.10 and 6.11. We observed this behavior in 6 out of the 100 domains.

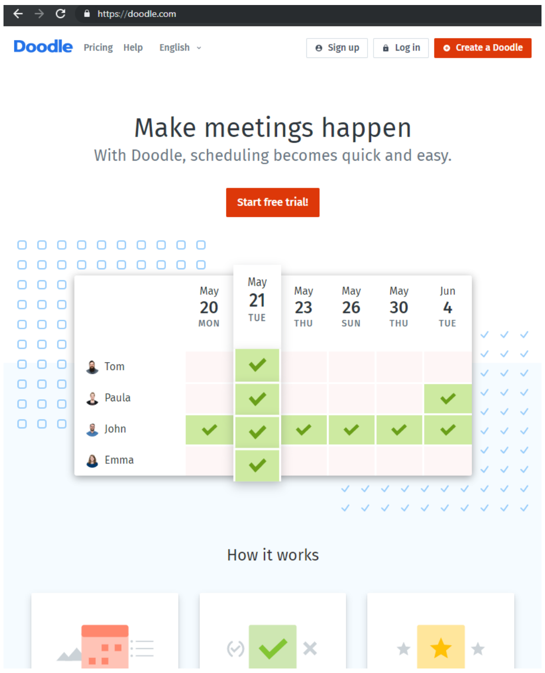


Figure 6.10: (Doodle) The logged out landing page.

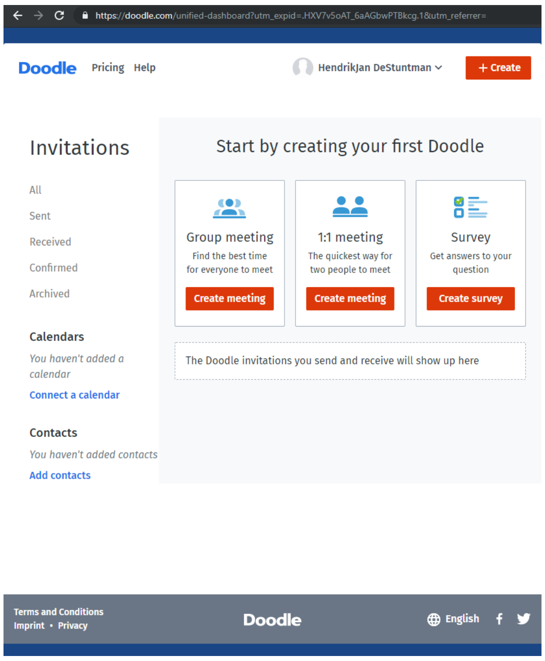

https://doodle.com/unified-dashboard?utm_expid=.HXV7v5oAT_6aAGbwPTBkcg.1&utm_referrer=
Doodle
Pricing
Help
HendrikJan DeStuntman
+ Create
Invitations
All
Sent
Received
Confirmed
Archived
Calendars
You haven't added a calendar
Connect a calendar
Contacts
You haven't added contacts
Add contacts
Start by creating your first Doodle
Group meeting
Find the best time for everyone to meet
Create meeting
1:1 meeting
The quickest way for two people to meet
Create meeting
Survey
Get answers to your question
Create survey
The Doodle invitations you send and receive will show up here
Terms and Conditions
Imprint • Privacy
Doodle
English

Figure 6.11: (Doodle) The automatically redirected page when the landing pages is requested.

The methods discussed so far cover 95 out of the 100 domains theoretically, and can be easily extended with patterns to detect login forms or elements such as account buttons. Another option is to calculate the side by side difference of the HTML from the logged in and logged out state. However we do think that we should give a meaning to the difference between the HTML files, the difference could be caused by changing content or different ads that are shown. Therefore we did not investigated this method any further.

6.6.3 Conclusions

The methods used by Zhou and Evans [11], detection of automated redirection, and in the Shepherd project together should provide enough coverage to verify the authenticated state for a substantial percentage of domains.

6.7 Automating the Verification

In the previous section we described methods how we could verify the authenticated state that cover a substantial percentage of domains. In this section we discuss how we used these methods in the SSSO verification module. To prove that these methods work we tested this module on the domains we investigated in the previous section.

6.7.1 Shepherds SSSO Verification Module

The authenticated state could leave a fingerprint in a web application in a manner of speaking, we described these in section 6.6. For the verification module we decided to create a class, Fingerprint, that searches the HTML DOM and the set of cookies for hints on the authenticated state or unauthenticated state. This search is performed twice, with and without the set of cookies that was stored by the SSSO login module. In this process also the URLs are stored. Afterwards the results of both the searches and the URLs are compared and a conclusion is made whether or not the set of cookies creates the authenticated state. A simplified UML diagram for this module can be found in the appendix.

Design Decisions We decided not to use any confidence factor for a specific method. The reason is that we could not decide how to apply these. The user's credentials in the HTML DOM and the absence of these when logged out is a good method, but why should it have higher confidence over automated redirection? Therefore we decided to treat all methods as equal.

Implementation Decisions We also opted to use as much JavaScript as possible for all searches in the HTML DOM. This is known to be faster than Python over Selenium because this needs extra communication between Python, Selenium and the browser. When testing we experienced that this module did not slowed us down and one search is mostly done in just a couple of seconds. Speed is important for this module because the scans might be repeated a large number of times to deduce the set of authorization cookies in the security scan.

To decide if a web application redirects when authenticated we decided to simply compare the URLs, if any difference is found, than some redirection is done. Some web application use the same URL for the logged out and authenticated state, however HTTPS is used when authenticated. With this simple method this is taken into account.

Module Results This module depends on the results of the SSSO login module, only for successful SSSO login attempts the verification is performed. This module stores for which SSSO provider it could verify the authenticated state for use in subsequent modules and this is also written in the logs for debugging purposes.

Example 6.11
The user's credentials in an attribute of an image tag.

```
<img src="https://cdn.igromania.ru/-Engine-/SiteTemplates/igromania/images/
misc/default_avatar.png" alt="HendrikJan DeStuntman [FB]"
title="HendrikJan DeStuntman [FB]" class="uprofile_av">
```

6.7.2 Testing Shepherds Verification Module

To provide metrics on the performance and reliability on the methods for the verification of the authenticated state, we tested Shepherd's verification module on all domains used in the analysis in section 6.6. We manually authenticated to each web application using SSSO and ran the verification module directly afterwards. During this process we also did a visual inspection.

Results The verification failed in 6 cases that are not related to this module.

- The framework failed in 2 cases.
- A cache loading time-out was experienced in 1 case.
- In 2 cases the authentication to the web application failed.
- In one case we could not load the page because of bot detection.

From the remaining 94 cases we observer that the session was not properly restored after loading the cookies and refreshing the page. In some cases this could be fixed by a forced cache refresh, pressing ctrl + f5, however this was only tried to find a cause and is not included in the results. We experienced this in 11 cases and we are not sure if this can be fixed at all and this needs further investigation.

This leaves 83 domains in which the verification methods failed in 10 cases, approximately a success rate of 84 %. In 6 cases none of our current methods work and in 4 cases the credentials could not be found in the HTML DOM which we could find by manual inspection. In three cases the user's credentials were found in an attribute of an image tag, example 11. A cause for the failed detection of the credentials in an image tag could be that it has not been loaded yet. Lazy loading or creation by JavaScript could also be the cause that the credentials in a data attribute, example 12, could not be found. For this we do not have a solution.

Conclusions This module performs reasonably well, however there is room for improvement. We included a pattern for detection of login forms and sign-up elements. Adding more patterns is relatively straight forward however can be time-consuming. Our main concern is the fact that 11 out of the 100 domains did not restored the authenticated state correctly after loading the cookies, resolving this issue could increase the performance of this module.

6.7.3 Conclusions

We are convinced that the combination of the methods already in Shepherd and those used by Zhou and Evans [11] bundled together with the URL comparison in this module provide simple and easy extensible methods to verify the authenticated state. The methods have shown to be reliable and cover most cases.

We do have an extension for this module in mind. Some web applications show the user an account page with the users credentials directly after authentication. If this URL is stored

Example 6.12
The user's credentials in a data attribute of a body tag.

```
<body data-user-guest="false" data-user="{&quot;firstName&quot;:null,&quot;
lastName&quot;:null,&quot;email&quot;:&quot;destuntmanhendrikjan@gmail.com
&quot;,&quot;id&quot;:&quot;78998420&quot;,&quot;phone&quot;:null}"
class="_frontend">
    ...
</body>
```

directly after the authentication has been done, and optionally after processing registration forms, this could be used when verification on the landing page failed. However this is future work.

6.8 Results and Discussion

In this chapter we investigated to what extent it is possible to automatically create the perspective of the authenticated user for any website or web application using any SSSO provider supported by the site in question.

Automated authentication to a SSSO provider can be done in almost all cases the element that triggers the entry point for the authentication is correctly identified. Handling the account creation procedures seems to be a major challenge which can not be solved easily. Verification of the authenticated state, which is important, has a success rate of approximately 84 % if we are able to recreate the authenticated state.

56 % of the domains do not require any additional account creation. When we take broken implementations and bot detection into account, Shepherd will not be able to authenticate anyway, and apply the success rate of 84 % from the verification, Shepherd is able to create the authenticated state in approximately 47 % of all domains that integrate SSSO if this is correctly detected and the SSSO provider is supported by Shepherd. However a study on a larger number of domains will provide more reliable numbers.

Chapter 7

Conclusions and Future Work

Our study enables session security research from the perspective of an authenticated user on a large number of domains by leveraging SSSO. We preferred to have more statistical evidence on this however this was hindered by external factors which limited our study. This could be the subject of a supplementary study or future work, for example a study on at least 10.000 domains from the top of the Alexa ranking or another suitable top list ranking.

We thoroughly explored the problem of detecting entry points of SSSO authentication processes for automation. We were only able to detect SSSO buttons on 15 out of 25 websites with SSSO detection in a small test. This low number can mostly be attributed to the amount of related problems that still need to be addressed, and the stability of the underlying technologies need to improve in order to actually improve our results.

Currently Shepherd does not support any Chinese SSSO provider. We managed to create a Chinese social media account on Sina Weibo, however we failed to use it for any domain, we did not investigate this any further. Adding support for Asian SSSO providers could increase the reach of Shepherd.

A limitation with the biggest impact is that we could not automate the handling of additional account creation procedures. This is required in 44 % of the domains, however we suspect that the success rate will stay quite low and any effort is better spent elsewhere.

During testing we experienced bot detection by internet service providers such as CloudFlare and Quantcast which results in not loading the page at all if a Google reCAPTCHA is not processed correctly or that it is needed to wait. Although we experienced this in only a small number of cases we suspect that this could pose a problem when scaling up. Finding methods to circumvent this detection could be an assurance that Shepherd will not be blocked on (a large number) domains.

Bibliography

[1] Maurits Martijn. *Maybe Better If You Don't Read This Story on Public WiFi.* 2014. URL: https://medium.com/matter/heres-why-public-wifi-is-a-public-health-hazard-dd5b8dcb55e6 (visited on 11/16/2018).

[2] HackerOne. *HackerOne: Hacktivity Dashboard.* URL: https://hackerone.com/hacktivity (visited on 03/23/2019).

[3] Frank Piessens. "A Taxonomy of Causes of Software Vulnerabilities in Internet Software". In: Citeseer. 2002.

[4] Graeme Burton. *Security experts Hit Out At Google Over Refusal To Patch Android Security Flaw Exploited By Ransomware.* 2017. URL: https://www.computing.co.uk/ctg/news/3010407/security-experts-hit-out-at-google-over-refusal-to-patch-android-security-flaw-exploited-by-ransomware (visited on 02/28/2019).

[5] Charlie Osborne. *Second time lucky: Cisco pushes fix for failed Webex vulnerability patch.* 2018. URL: https://www.zdnet.com/article/cisco-pushes-new-patch-to-fix-failed-fix-for-severe-webex-vulnerability/ (visited on 02/28/2019).

[6] Antonio Nappa et al. "The attack of the clones: A study of the impact of shared code on vulnerability patching". In: *2015 IEEE symposium on security and privacy.* IEEE. 2015, pp. 692–708.

[7] Dan Goodin. *Unsafe cookies leave WordPress accounts open to hijacking, 2-factor bypass.* 2014. URL: https://arstechnica.com/information-technology/2014/05/unsafe-cookies-leave-wordpress-accounts-open-to-hijacking-2-factor-bypass/ (visited on 11/10/2018).

[8] Raphael Satter. *Yahoo issues another warning in fallout from hacking attacks.* 2017. URL: https://apnews.com/cb14057fa1b24569bed73ae369d9132b/apnewsbreak-yahoo-issues-new-security-warning-users (visited on 06/06/2018).

[9] Robert McMillan. *Yahoo Triples Estimate of Breached Accounts to 3 Billion.* URL: https://www.wsj.com/articles/yahoo-triples-estimate-of-breached-accounts-to-3-billion-1507062804 (visited on 04/13/2019).

[10] Julia Carriew. *Facebook says nearly 50m users compromised in huge security breach.* 2018. URL: https://www.theguardian.com/technology/2018/sep/28/facebook-50-million-user-accounts-security-berach (visited on 11/10/2018).

[11] Yuchen Zhou and David Evans. "SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities." In: *USENIX Security Symposium.* 2014, pp. 495–510.

[12] Mohammad Ghasemisharif et al. "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, 2018, pp. 1475–1492. ISBN: 978-1-931971-46-1. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/ghasemisharif.

[13] Yogesh Mundada, Nick Feamster, and Balachander Krishnamurthy. "Half-baked cookies: Hardening cookie-based authentication for the modern web". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM. 2016, pp. 675–685.

[14] Yusuke Takamatsu, Yuji Kosuga, and Kenji Kono. "Automated detection of session management vulnerabilities in web applications". In: *2012 Tenth Annual International Conference on Privacy, Security and Trust* (2012), pp. 112–119.

[15] Hugo Jonker et al. "Shepherd: Enabling Automatic and Large-Scale Login Security Studies". In: *arXiv preprint arXiv:1808.00840* (2018).

[16] OWASP. *Session Management Cheat Sheet*. 2017. URL: `https://www.owasp.org/index.php/Session_Management_Cheat_Sheet` (visited on 11/16/2018).

[17] The Internet Society. *Hypertext Transfer Protocol – HTTP/1.1*. 1999. URL: `https://tools.ietf.org/html/rfc2616` (visited on 12/05/2018).

[18] The Internet Society. *HTTP State Management Mechanism*. 2011. URL: `https://tools.ietf.org/html/rfc6265` (visited on 04/13/2019).

[19] OWASP. *Session hijacking attack - OWASP*. URL: `https://www.owasp.org/index.php/Session_hijacking_attack` (visited on 06/29/2019).

[20] Adam Barth. *HTTP state management mechanism*. Tech. rep. 2011.

[21] Rorot. *Session Hijacking Cheat Sheet*. 2015. URL: `https://resources.infosecinstitute.com/session-hijacking-cheat-sheet/` (visited on 12/10/2018).

[22] Dafydd Stuttard and Marcus Pinto. *The web application hacker's handbook: Finding and exploiting security flaws*. John Wiley & Sons, 2011.

[23] Andrew Whitaker and Daniel P Newman. *Penetration Testing and Network Defense: Penetration Testing _1*. Cisco Press, 2005.

[24] OWASP. *Cross-site Scripting (XSS) - OWASP Cross-site Scripting (XSS)*. 2018. URL: `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)` (visited on 11/10/2018).

[25] OWASP. *Session hijacking attack - OWASP Session hijacking attack*. 2014. URL: `https://www.owasp.org/index.php/Session_hijacking_attack` (visited on 11/10/2018).

[26] OWASP. *Session fixation*. 2018. URL: `https://www.owasp.org/index.php/Session_fixation` (visited on 11/10/2018).

[27] Michael Olson. *Research Study: Consumer Perceptions of Online Registration and Social Sign-in*. 2011. URL: `https://www.janrain.com/blog/research-study-consumer-perceptions-online-registration-and-social-sign` (visited on 05/10/2018).

[28] Kelsey Goings and Paul Abel. *The Value of Social Login*. 2013. URL: `http://www1.janrain.com/rs/janrain/images/Industry-Research-Value-of-Social-Login-2013.pdf` (visited on 01/09/2019).

[29] Dinei Florencio and Cormac Herley. "A large-scale study of web password habits". In: *Proceedings of the 16th international conference on World Wide Web*. ACM. 2007, pp. 657–666.

[30] Pavol Sovis, Florian Kohlar, and Jörg Schwenk. "Security analysis of openid". In: *Sicherheit 2010. Sicherheit, Schutz und Zuverlässigkeit* (2010).

[31] Christian Mainka et al. "Automatic recognition, processing and attacking of single sign-on protocols with burp suite". In: *Open Identity Summit 2015* (2015).

[32] Rui Wang, Shuo Chen, and XiaoFeng Wang. “Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services”. In: *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE. 2012, pp. 365–379.

[33] Wanpeng Li and Chris J Mitchell. “Security issues in OAuth 2.0 SSO implementations”. In: *International Conference on Information Security*. Springer. 2014, pp. 529–541.

[34] Wanpeng Li and Chris J Mitchell. “Analysing the security of Google’s implementation of OpenID Connect”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, pp. 357–376.

[35] Quirin Scheitle et al. “A long way to the top: significance, structure, and stability of internet top lists”. In: *Proceedings of the Internet Measurement Conference 2018*. ACM. 2018, pp. 478–493.

[36] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. “Detecting near-duplicates for web crawling”. In: *Proceedings of the 16th international conference on World Wide Web*. ACM. 2007, pp. 141–150.

[37] Ismail Akrout, Amal Feriani, and Mohamed Akrout. “Hacking Google reCAPTCHA v3 using Reinforcement Learning”. In: *CoRR* abs/1903.01003 (2019). arXiv: `1903.01003`. URL: `http://arxiv.org/abs/1903.01003`.

[38] Ye Wang and Mi Lu. “An optimized system to solve text-based CAPTCHA”. In: *CoRR* abs/1806.07202 (2018). arXiv: `1806.07202`. URL: `http://arxiv.org/abs/1806.07202`.

[39] Ahmad B. A. Hassanat. “Bypassing Captcha By Machine A Proof For Passing The Turing Test”. In: *CoRR* abs/1409.0925 (2014). arXiv: `1409.0925`. URL: `http://arxiv.org/abs/1409.0925`.

Appendices

Appendix A

SSSO Detection Experiments

A.1 Prevalence of SSSO On The Web

The following experiment was conducted to validate the results of an experiment done by Ghasemisharif et al. [12] regarding the prevalence of SSSO on the web. We manually investigated websites hosted by 2000 web domains on July 4 2019, which were randomly sampled from the Alexa Global Top 1M list of July 3 2019. We also extended the original experiment by further investigating the authentication systems of these websites.

A.1.1 Sampling

We used a list of 2000 web domains as our sample, which was generated in the following way:

- we downloaded the Alexa Global Top 1M list of July 3 2019
- we shuffled the list in a random way, and took the 2000 first entries

A.1.2 Process

During the experiment, we made sure that we had not been authenticated yet by any SSSO provider or website. The following steps were executed for every domain:

1. If the domain was a subdomain of a known service that hosted websites that cannot have their own authentication system (e.g. *username*.tumblr.com), the website was not deemed a target, and skipped.
2. To determine whether a website was hosted on this domain, we executed the following steps until a page was loaded by the browser:
    - type domain name in address bar of web browser
    - type domain name with "www." prepended in address bar of web browser
    - use a search engine or DNS recon tool to find if a website is hosted on this domain
    - otherwise the website is deemed unavailable.
3. If a website was discovered in the previous step, we searched the website for any evidence of an authentication system, resulting in one of the following outcomes:
    (a) none: no evidence of an authentication system was found
    (b) closed: a user could log in, but there was no way for a user to register an account
    (c) public: a user could log in, and there was a way for a user to register an account
4. If any evidence of SSSO authentication was discovered in the previous step, we identified which SSSO providers were supported by the website.

Note that one should be careful when using a VPN while conducting this experiment, because some websites or countries may block any attempts to access a website when using a VPN.

ranking	domains	not a target		unavailable		none		closed		public	
1-100k	192	0	0.00%	10	5.21%	91	47.40%	11	5.73%	80	41.67%
100k-200k	209	3	1.44%	8	3.83%	117	55.98%	12	5.74%	69	33.01%
200k-300k	183	4	2.19%	1	0.55%	102	55.74%	8	4.37%	68	37.16%
300k-400k	214	5	2.34%	3	1.40%	122	57.01%	16	7.48%	68	31.78%
400k-500k	212	12	5.66%	9	4.25%	123	58.02%	9	4.25%	59	27.83%
500k-600k	192	5	2.60%	5	2.60%	111	57.81%	7	3.65%	64	33.33%
600k-700k	210	11	5.24%	4	1.90%	135	64.29%	8	3.81%	52	24.76%
700k-800k	199	11	5.53%	4	2.01%	132	66.33%	11	5.53%	41	20.60%
800k-900k	184	11	5.98%	5	2.72%	123	66.85%	3	1.63%	42	22.83%
900k-1M	205	7	3.41%	4	1.95%	137	66.83%	7	3.41%	50	24.39%
1-1M	**2000**	**69**	**3.45%**	**53**	**2.65%**	**1193**	**59.65%**	**92**	**4.60%**	**593**	**29.65%**

Table A.1: Results of our investigation of authentication systems in experiment A.1. All percentages indicate the prevalence among domains with a rank within the stated range.

ranking	SSSO	domains	SSSO%	public	SSSO%
1-100k	33	192	17.19%	80	41.25%
100k-200k	20	209	9.57%	69	28.99%
200k-300k	19	183	10.38%	68	27.94%
300k-400k	12	214	5.61%	68	17.65%
400k-500k	14	212	6.60%	59	23.73%
500k-600k	9	192	4.69%	64	14.06%
600k-700k	11	210	5.24%	52	21.15%
700k-800k	8	199	4.02%	41	19.51%
800k-900k	6	184	3.26%	42	14.29%
900k-1M	7	205	3.41%	50	14.00%
1-1M	**139**	**2000**	**6.95%**	**593**	**23.44%**

Table A.2: Prevalence of SSSO authentication in experiment A.1. All percentages indicate the prevalence of SSSO for a certain range of rankings within a group of results.

A.1.3 Analysis

We found that 139 domains hosted a website with SSSO authentication (6.95% ± 1.11%, CI 95%), which is similar to the result of 6.3%, reported by Ghasemisharif et al. [12]. We also found that 593 domains hosted a website with user registration (29.65% ± 2.00%, 95% CI), which means that we expect to be able to use SSSO authentication for 23.57% (± 3.41%, CI 95%) of websites with user registration.

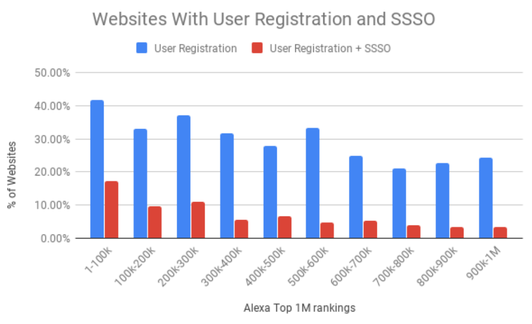


Figure A.1: Percentages of websites with user registration and SSSO.

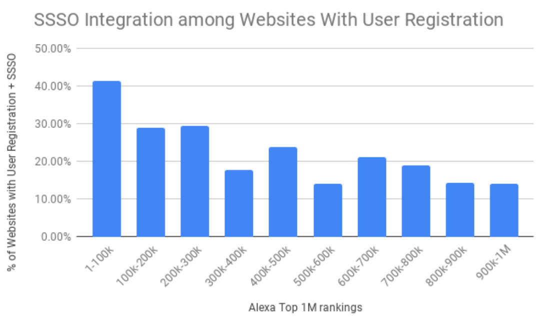


Figure A.2: Percentages of websites with SSSO among websites with user registration.

As shown in table A.3, we observed several strong correlations between the ranking of a website and the prevalence of authentication systems, that are statistically significant, including the trend observed by Ghasemisharif et al. [12] that a website with a better ranking is more likely to support SSSO authentication. We can argue that this trend is driven by the following related trends:

- a website with a better ranking is more likely to have an authentication system
- a website with a better ranking and user registration is more likely to support SSSO

Y	corr(ranking, Y)	p-value
none/domains	0.96	<0.001
auth/domains	-0.92	<0.001
closed/domains	-0.75	0.011
public/domains	-0.85	0.002
ssso/domains	-0.95	<0.001
ssso/public	-0.83	0.003

Table A.3: Correlation analysis of results of experiment A.1 using Spearman rank correlation.

We also confirmed the dominance of Facebook as an SSSO provider among websites listed in the Alexa Global Top 1M list, followed by Google, Twitter, and a long tail of other SSSO providers, as shown in table A.4, and figure A.3.

provider	count	websites w\ SSSO	all websites
facebook	107	76.98%	5.35%
google	66	47.48%	3.30%
twitter	27	19.42%	1.35%
vk	11	7.91%	0.55%
linkedin	6	4.32%	0.30%
amazon	5	3.60%	0.25%
microsoft	5	3.60%	0.25%
wechat	5	3.60%	0.25%
ok.ru	4	2.88%	0.20%
steam	4	2.88%	0.20%
yahoo	4	2.88%	0.20%
qq	4	2.88%	0.20%
instagram	3	2.16%	0.15%
mail.ru	3	2.16%	0.15%
yandex	3	2.16%	0.15%
weibo	3	2.16%	0.15%

Table A.4: List of SSSO providers most commonly found on websites during experiment.

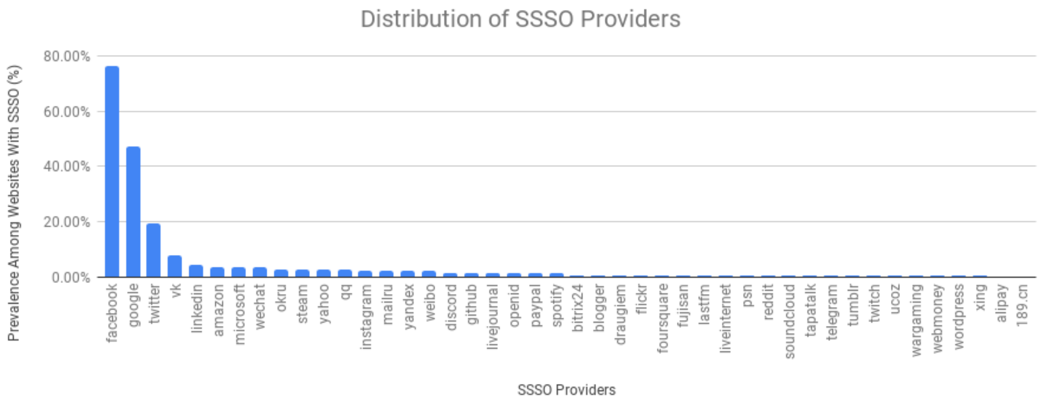


Figure A.3: Integration of different SSSO providers by websites among experiment's results.

A.1.4 Conclusions

We have validated the findings of Ghasemisharif et al. [12], and also discovered some new trends correlating the ranking of a website and authentication systems.

A.2 Clicking SSSO Buttons To Initiate Authentication Process

The following experiment was conducted to confirm hypothesis 5.1, which states that a user only needs to click an SSSO button to initiate an SSSO authentication process.

A.2.1 Sampling

The sample used for this experiment was a list of 139 domains that hosted a website with SSSO authentication, taken from experiment A.1. The conditions of randomness were not violated because the sample used in A.1 was generated in a random way.

A.2.2 Process

During the experiment, we made sure that we had not been authenticated yet by any SSSO provider or website. The following steps were executed for every website from the sample:

1. We identified all entry points for SSSO authentication on a website.
2. We executed the following steps for every entry point:
   (a) is SSSO authentication process part of website's own authentication system?
   (b) check hypothesis:
      - positive: user needs to click an SSSO button to initiate SSSO authentication
      - negative: otherwise

Note that one should be careful when using a VPN while conducting this experiment, because some websites or countries may block any attempts to access a website when using a VPN.

A.2.3 Analysis

We found 396 different entry points for SSSO authentication processes on 139 websites. We found more entry points among websites with a better ranking. This can be attributed to the trend that popular websites are more likely to let users log in with SSSO, which we observed in experiment A.1. We discovered an average of 2.85 entry points per website with SSSO authentication. No statistically significant correlation was found between the rank of a website with SSSO authentication, and the amount of entry points discovered on that website.

ranking	1-100k	100k-200k	200k-300k	300k-400k	400k-500k	500k-600k	600k-700k	700k-800k	800k-900k	900k-1M	**1-1M**
domains w/ SSSO	33	20	19	12	14	9	11	8	6	7	**139**
entry points	87	47	79	23	41	25	48	15	16	15	**396**
avg. per domain	2.64	2.35	4.16	1.92	2.93	2.78	4.36	1.88	2.67	2.14	**2.85**

Table A.5: Distribution of entry points among domains with SSSO authentication.

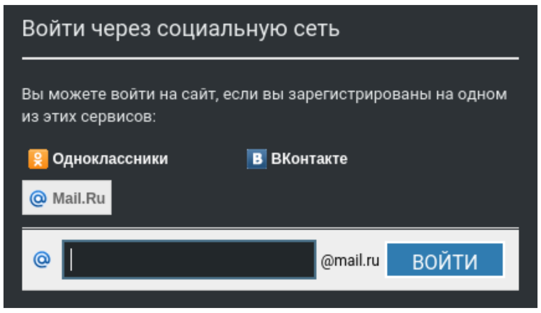
We made the following observations:

- hypothesis 5.1 was true for 382 out of 396 entry points (96.46% ± 1.82%, 95% CI)
- no entry points of an unrelated third party service were found during this experiment

ranking	1-100k	100k-200k	200k-300k	300k-400k	400k-500k	500k-600k	600k-700k	700k-800k	800k-900k	900k-1M	**1-1M**
entry points	87	47	79	23	41	25	48	15	16	15	**396**
H5.1 = true	$\underline{\underline{85}}$	47	79	$\underline{\underline{22}}$	$\underline{\underline{33}}$	25	$\underline{\underline{45}}$	15	16	15	**382**

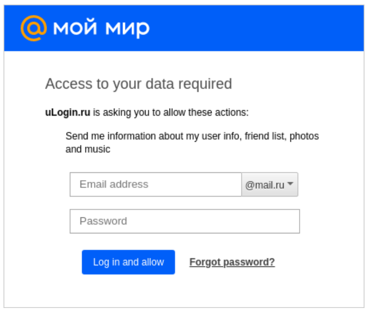
Table A.6: Results of experiment A.2 investigating hypothesis 5.1.

We discovered 14 entry points of SSSO authentication processes on 5 different websites for which hypothesis 5.1 was false. In all but one of these cases, users were expected to identity themselves beforehand, as demonstrated in figure A.4, which may happen when SSSO authentication is implemented by using protocols such as Mozilla BrowserID, OpenId, or OpenId Connect. This way of handling SSSO authentication is only used by a small amount of unpopular SSSO providers, and these SSSO providers may also implement SSSO authentication in a way that conforms with our hypothesis, as is shown in figure A.5.


Войти через социальную сеть

Вы можете войти на сайт, если вы зарегистрированы на одном из этих сервисов:

Figure A.4: (darxton.ru) Users need to identify themselves beforehand by providing their *mail.ru* email address.



Access to your data required

uLogin.ru is asking you to allow these actions:

Send me information about my user info, friend list, photos and music

Figure A.5: (emigrantforum.ru) Users are redirected to web page hosted by *mail.ru* after clicking on SSSO button.

In the other case where our hypothesis failed, a Chinese website presented users with a QR code that was embedded on its own website. Users need to scan this QR code with a mobile device to initiate SSSO authentication. The use of QR codes is more prominent in Chinese culture, and the most popular Chinese SSSO providers support the use of QR codes for handling SSSO authentication.

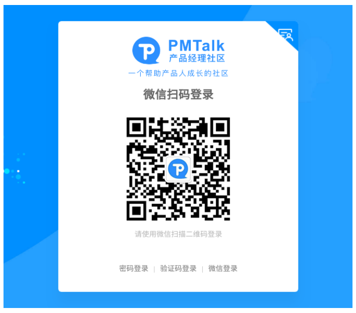


Figure A.6: (ipmtalk.com) Example of a QR code being used to log in with WeChat.

We also would like to note that we are aware of other situations where hypothesis 5.1 does not hold up, but such cases were not observed during this experiment. In one case, users needed to agree with a 'Terms of Use' before being able to register a new account using SSSO authentication. In another case, users were required to choose a username before they were able to register a new account.

A.2.4 Conclusion

We conclude that hypothesis 5.1 is true in almost all situations. Situations where the hypothesis does not hold up are not common or significant enough to invalidate its usefulness.

A.3 Redirection of Browser After Clicking SSSO Buttons

The following experiment was conducted to confirm hypothesis 5.2, which states that if a user has not been authenticated yet by the SSSO provider, and clicks on an SSSO button, then the browser is redirected to a web page for authentication by the SSSO provider. Due to our interest in avoiding several pitfalls related to SSSO detection, we also investigated these web pages for evidence that hints at which application we are trying to use SSSO authentication for.

A.3.1 Sampling

The sample used for this experiment was a list of 382 entry points of authentication processes which we discovered during A.2. All of these entry points were SSSO buttons for which hypothesis 5.1 was found to be true.

A.3.2 Process

The following steps were executed for every entry point in our sample:

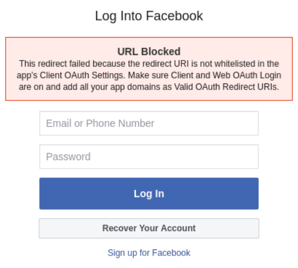
1. click the SSSO button
2. check hypothesis:
   - positive: browser opens web page for authentication by SSSO provider
   - negative: otherwise
   - did an error occur, and if so, why?
3. any evidence hinting at the service that we are trying to use SSSO authentication for?

A.3.3 Analysis

We found that hypothesis 5.2 was true for only 332 out of 382 entry points (86.91% $\pm$ 3.38%, 95% CI), which can be attributed to a large amount of broken SSSO authentication processes. After clicking an SSSO button, an error message or behaviour indicative of a faulty implementation was observed in 50 out of 382 cases (13.09% $\pm$ 3.38%, 95% CI):

- in 34 cases, SSSO authentication was badly integrated by the service provider's website
- in 16 cases, SSSO authentication was misconfigured on the SSSO provider's end

We found no correlation between the ranking of a website, and the likelihood of an SSSO authentication process being broken, as can be seen in figure A.8.


Log Into Facebook

URL Blocked
This redirect failed because the redirect URI is not whitelisted in the app's Client OAuth Settings. Make sure Client and Web OAuth Login are on and add all your app domains as Valid OAuth Redirect URIs.

Figure A.7: A warning message is shown to notify users that SSSO authentication is misconfigured on the SSSO provider's end.

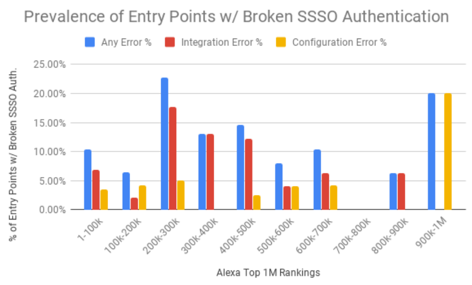
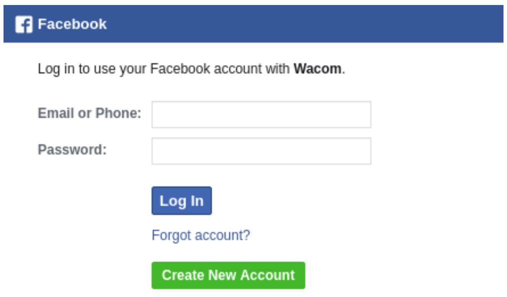


Figure A.8: Chart displays the rate of errors among websites belonging to different rankings.

At least one SSSO authentication process was broken for 27 out of 139 websites (19.42% ± 6.58%, 95% CI), and all SSSO authentication processes were broken for 8 out of 139 websites (5.76% ± 3.87%, 95% CI). Actual error rates might be higher than reported, and thus present a lower bound, because of the following reasons:

- the amount of configuration errors might be higher, because we could not identify these if the browser wasn't redirected to a web page for authentication by the SSSO provider due to an integration error
- the amount of integration errors might be higher, because we only clicked on SSSO buttons, and did not follow through with the whole SSSO authentication process

Hypothesis 5.2 held true in all other cases. This implies that we can reliably validate SSSO buttons, on condition that SSSO authentication is initiated by clicking on the SSSO button, and that SSSO authentication is integrated and configurated correctly. As such, we can also use this method to identify the entry points of SSSO authentication processes that we can automate.


Facebook

Log in to use your Facebook account with **Wacom**.

Email or Phone:

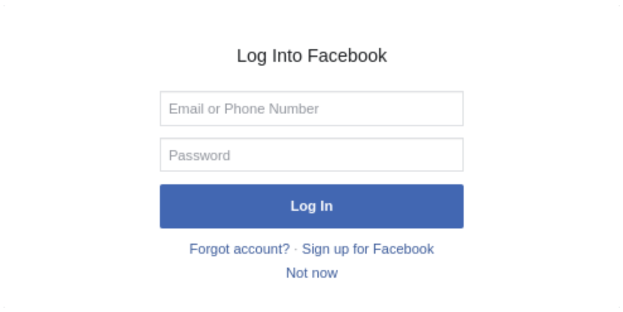
Password:

Log In

Forgot account?

Create New Account

Figure A.9: (wacom.com) Display name of application is shown when being authenticated by SSSO provider.


Log Into Facebook

Email or Phone Number

Password

Log In

Forgot account? · Sign up for Facebook

Not now

Figure A.10: (scake.com.tw) No visual information to identify application is shown when being authenticated by SSSO provider.

We also expressed our concern about several potential pitfalls related to SSSO detection. The web page of the SSSO provider that we are redirected to after clicking an SSSO button usually contains evidence of which service we are trying to use SSSO authentication for. In 323 out of 332 cases (97.29% ± 1.75%, 95% CI), we found evidence that could be used to identify the application, or the third party service that was used by that application to integrate SSSO authentication:

- in 303 cases (91.27% ± 3.04%, 95% CI), evidence was found in the URL
- in 190 cases (57.23% ± 5.32%, 95% CI), evidence was found within the page itself

In the remaining 9 cases where we could not find any evidence, specific SSSO providers were used, such as Amazon Pay, and thus we can anticipate for which SSSO providers we won't find any evidence.

We can use this information to fingerprint common third party services that are embedded on other websites to avoid confusing SSSO buttons of other services with SSSO buttons of the targeted website.

A.3.4 Conclusions

Based on these findings, we conclude that hypothesis 5.2 can be used to reliably validate SSSO buttons when SSSO authentication is integrated correctly. In addition, we can examine the web page that we are redirected to, to avoid the pitfalls related to SSSO detection listed in section 5.4.

A.4 Locations of Components Related To Authentication

In this experiment, we investigated the locations of elements related to authentication on a website, in order to confirm hypothesis 5.3, which states that we can expect to find SSSO buttons in locations where we also expect other components related to authentication to appear.

A.4.1 Sampling

Our sample was a list of 139 domains that were found to be hosting websites that had integrated SSSO during experiment A.1.

A.4.2 Process

The following steps were executed for every website from the sample:

1. identify all pages of the website
2. for each page, determine the following:
   (a) at which search depth did we find the page?
   (b) which locations of the page contain which components related to authentication?

A.4.3 Analysis

First of all, we observed that the search depth for login pages and registration pages is typically a depth of 1. For some websites, the registration page was found at a search depth of 2, in which case, the registration page was referenced from the login page. However, a significant amount of websites did not have a login page or registration page.

- 66.00% ± 13.13% (confidence level 95%) of websites are had a login page
- 64.00% ± 13.30% (confidence level 95%) of websites are had a registration page

page	n	depth=0		depth=1		depth=2		depth=3	
login	33	1	3.03%	32	96.97%	0	0.00%	0	0.00%
registration	32	0	0.00%	27	84.38%	5	15.63%	0	0.00%

Table A.7: Contains data about the amount of websites with dedicated login and/or registrationpages, and how these are distributed among different search depths.

We found that any component related to authentication could be in almost any location. However, we noticed a trend that all 16 SSSO buttons that we found were near a username-password form, or a registration form. This observation led to hypothesis 5.4.

A.4.4 Conclusions

Besides identifying the typical locations of where we can find components related to authentication, we also found that login pages and registration pages are usually referenced from the main page. Not all websites with user authentication have login pages and registrations, in which case, the user has to log in, or register, from the main page.

A.5 Recognition of SSSO Buttons Using Keywords

This experiment was done to confirm hypothesis 5.5, which states that we can use keywords to recognize SSSO buttons, as described in section 5.6. In addition, we further investigated the collected data to confirm the helpfulness of heuristics listed by Zhou and Evans [11], and to identify new ones, which could help us with filtering and sorting potential candidates of SSSO buttons.

A.5.1 Sampling

Our sample was a list of 50 domains that were found to be hosting websites that had integrated SSSO during experiment A.1.

A.5.2 Process

The following steps were executed for every domain in the used sample:

1. We identified each SSSO button on the website.
2. For each SSSO button that we identified in the previous step, we determined the following:
    - does the element's source contain a reference to its SSSO provider? (hypothesis 5.5)
    - which keywords related to its SSSO provider do parts of the element contain?
        - visible text
        - id/class attributes
        - outer HTML
    - which other keywords related to authentication can we find?

A.5.3 Analysis

We investigated a total of 126 different SSSO buttons on 50 different websites. Hypothesis 5.5 held true for 122 out of 126 SSSO buttons (96.83% ± 3.06%, 95% CI). We could not use keywords to identify SSSO buttons on 3 different websites, because they were disguised as a normal sign-in button, with not even a visual clue that they were SSSO buttons. These buttons would be confused with a more traditional SSSO buttons.

We compared the effectiveness of 3 different methods of searching elements for keywords.:

- by only observing visual text, we would be able to recognize 63 out of 122 recognizable SSSO buttons (51.64% ± 8.87%, 95% CI)
- by only searching the ID and/or class attributes of an element, we would be able to recognize 80 out of 122 recognizable SSSO buttons
- by searching the full outer HTML of an element, we were able to recognize all 122 recognizable SSSO buttons

In addition, searching the complete outer HTML of an element has the benefit of finding more keyword matches, which we believe can help us differentiate between incidental candidates, and great candidates. The most common keywords for recognizing SSSO buttons were the names of SSSO provider's, such as *facebook*, *twitter*, and *google*. In addition, common abbreviations of these names should be used to cover more SSSO buttons.

A.5.4 Conclusions

We can use keywords to recognize keywords, as stated by hypothesis 5.5, however, for it to be effective, we need to search an element's full outer HTML. Another precondition for our search method to be effective, is using a wide range of different keywords that cover all common variations.

A.6 Persistent CSS Selectors

We conducted this experiment to investigate the effectiveness and the ability of different CSS selectors to reference elements in a persistent way.

A.6.1 Sampling

The sample used for this experiment was a list of 10 domains that hosted a website with SSSO authentication, taken from experiment A.1. The conditions of randomness were not violated because the sample used in A.1 was generated in a random way.

A.6.2 Process

The following steps were executed for every domain in the used sample:

1. we identified every element of interest on the website, which includes the following:
   - SSSO buttons
   - login buttons
   - registration buttons
2. For each element of interest that we discovered in the previous step, we would determine whether or not the following selectors would work:
   - id-based selectors
   - class-based selectors
   - persistent selectors, as described in section 5.7

A.6.3 Analysis

We identified 51 different elements of interest on 10 websites, and found that:

- id-based selectors could be used in 10 out of 51 cases (19.61%)
- class-based selectors could be used in 43 out of 51 cases (84.31%)
- persistent selectors could be used in all cases

A.6.4 Conclusions

We have determined that persistent selectors, as described in section 5.7, can be used to reliably reference HTML elements in a persistent way.

A.7 Action Sequences For Clicking on SSSO Buttons

During this experiment, we investigated the sequences of interactions that were needed to click on SSSO buttons and other elements of interest.

A.7.1 Sampling

For this experiment, we used a sample of 50 different domains with an authentication system and user registration, which we found during experiment A.1. The conditions of randomness were not violated because the sample used in A.1 was generated in a random way.

A.7.2 Process

The following steps were executed for every domain in the used sample:

1. We identified every element of interest on the website, which includes the following:
   - SSSO buttons
   - login buttons
   - registration buttons
2. For each element of interest that we discovered in the previous step, we would determine the following:
   - what's the type of the element: login button, registration button, SSSO button?
   - which page was the element found on: main, login, registration?
3. We then determined which steps it would take to interact with the button (e.g. "click, click", or "hover, click, click").

A.7.3 Analysis

We found 232 different action chains on 50 different websites, of which the distribution is described in table A.8. Almost all action chains consisted of clicks. We only found a single element where the cursor had to placed on top of it.

action chain	**n**
click	173
click, click	42
click, click, click	12
hover, click	3
hover	1

Table A.8: Distribution of different action chains.

We were able to find all types of each button on each page. However, we observed the trend that only action chains on the main page consisted of more than one step, and that we could always directly click any elements of interest on login and registration pages, as shown in A.9.

page	**n=1**		**n=2**		**n=3**		**n=4**		**n=5**	
main	73	31.60%	45	19.48%	12	5.19%	0	0.00%	0	0.00%
login	61	26.41%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
registration	46	19.91%	0	0.00%	0	0.00%	0	0.00%	0	0.00%

Table A.9: Distribution of chain lengths found on different pages.

When we only looked at the data for SSSO buttons, we noticed that we found no action chains with a single step on the main page, while we only found action chains with a single step on the login and registration page.

page	**n=1**		**n=2**		**n=3**		**n=4**		**n=5**	
main	0	0.00%	10	28.57%	5	14.29%	0	0.00%	0	0.00%
login	12	34.29%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
registration	8	22.86%	0	0.00%	0	0.00%	0	0.00%	0	0.00%

Table A.10: Distribution of chain lengths found on different pages (SSSO buttons only).

We also found that our action chain had to access the contents of an iframe only once.

A.7.4 Conclusions

We have determined that most action chains tend to be shorter, and that the page plays an immense role when it comes to the length of an action chain. Action chains found on login and registration pages tend to consist of a single step. Interactions where we need to place the cursor on top of an element are not very common.

A.8 Finding URLS of Relevant Pages

In experiment A.4, we identified the main page, login page, and registration page as pages where we should search for SSSO buttons. To find the URLs of these pages, there are several methods that we can employ. In this experiment, we determined whether or not we could extract references to login pages

A.8.1 Sampling

For this experiment, we used a sample of 20 different domains with an authentication system and user registration, which we found during experiment A.1. The conditions of randomness were not violated because the sample used in A.1 was generated in a random way.

A.8.2 Process

The following steps were executed for every domain in the used sample:

1. We identified each login button, registration button, and SSSO button, that would result in a new page being loaded, when clicked upon.
2. For each button that we discovered in the previous step, we would determine the following:
    - what's the type of the element: login button, registration button, SSSO button?
    - what kind of page was loaded: login page, registration page, ssso provider's page?
    - where can we find URL of page: HTML attribute, JavaScript?
    - what is the URL?

A.8.3 Analysis

We found 75 different buttons on 20 websites, that would result in a new page being loaded when clicked upon. We could extract the URL in the following cases:

- in 69 out of 75 cases (92.00%), the URL was the value of an HTML attribute
- in 6 out of 75 cases (8.00%), the redirection was implemented using JavaScript

A.8.4 Conclusions

We can extract most URLs for login and registration pages by checking the HTML attributes of a login or registration button. When the URL is not stored as an HTML attribute, we can click the button.

Appendix B

SSSO Login Experiments

domain	fb	go	vk	tw	ok	li	ya	ma	in	gh
letyshops.com	OK	OK								
pons.com	OK									
eodev.com	OK									
uploadboy.com	OK	OK								
neoseeker.com	OK	OK								
pptcloud.ru	1	OK	OK		1					
mangahere.cc	OK									
tonkosti.ru	OK		OK	OK						
eurogamer.net	OK									
bsnl.co.in	OK									
cdkeys.com		OK								
netbarg.com		1								
toggl.com		OK								
lazada.co.th		OK								
goodreads.com		OK								
allaboutcircuits.com		OK								
obrazovaka.ru			OK							
kommersant.ru			OK		OK					
stranamam.ru			OK		OK		OK	OK		
onedio.ru			OK							
drive2.ru			OK							
yaklass.ru			OK		OK					
superjob.ru			OK		OK		OK	OK		
english-films.com			OK							
disqus.com				OK						
altema.jp				OK						
lonelyplanet.com				OK						
sidereel.com				OK						
thairath.co.th				OK						
genius.com				OK						
start.me				OK						
elperiodico.com				OK						
hinative.com				OK						
trud.com					1			OK		
sibnet.ru					OK		OK	OK		

domain	fb	go	vk	tw	ok	li	ya	ma	in	gh
spcs.me					OK			OK		
fotostrana.ru					OK			OK		
gisher.org					OK		OK	OK		
leetcode-cn.com						OK				OK
csdn.net										OK
openstreetmap.org										OK
hostinger.com										OK
codeproject.com										OK
onesignal.com										OK
os.tc										OK
qiita.com										OK
codecademy.com						OK				OK
juejin.im										OK
nasdaq.com						OK				
codeprojecte.com						OK				
mediaexpert.pl						1				
justia.com						OK				
lynda.com						OK				
cadenaser.com						OK				
jusbrasil.com.br						OK				
c-sharpcorner.com						4				
rutube.ru								3, 4		
tcrain.ru								OK		
hh.ru								OK		
apteka.ru							2			
mysku.ru							OK			
gdeposylka.ru							OK			
baby.ru							OK			
rulit.me							OK			
coop-land.ru							2			
yaklass.ru									OK	
promodj.com									OK	
ask.fm									OK	
twitcasting.tv									OK	

Table B.1: All domains used for testing Shepherds SSSO login module and all results.

fb = Facebook, go = Google, vk = VKontakte, tw = Twitter, ok = ok.ru,
li = LinkedIn, ya = Yandex, ma = mail.ru, in = Instagram, gh = GitHub

1. Successfully authenticated, however SSSO not set up correctly or broken.
2. SSSO button wrongfully identified.
3. Elements needed to interact with not detected for mail.ru. This is corrected.
4. Suspicious behavior detected.

Appendix C

SSSO Account Creation Experiments

Alexa Rank	Domain	Observations
30	netflix.com	1
175	ebay.de	
318	mi.com	
336	box.com	1, 6
496	znanija.com	3
537	goo.ne.jp	1, 3, 4
545	discogs.com	
613	viva.co.id	1, 3, 8
620	snapdeal.com	5
673	bild.de	
729	kizlarsoruyor.com	
746	sputniknews.com	1, 4
890	news.com.au	
1058	abc.net.au	
1131	online-convert.com	
1220	aimer.tmall.com	4
1289	freelancer.com	2
1526	mvideo.ru	3, 4, 5
1554	thedailybeast.com	2
1576	gitlab.com	
1667	metacritic.com	
1700	livestrong.com	
1735	lavanguardia.com	
1742	shopee.com.my	
1828	jalopnik.com	
1885	islcollective.com	
1934	lining.tmall.com	4
1957	icook.tw	
2127	gainreel.tmall.com	4
2191	shopee.co.th	
2359	globaltestmarket.com	3, 7
2468	earthmusic.tmall.com	

Alexa Rank	Domain	Observations
2656	maniform.tmall.com	4
2989	qoo10.sg	1
3045	ticketmonster.co.kr	1
3077	qatarairways.com	1, 3
3078	yandex.ua	3
3168	kicker.de	1, 9
3220	jackjones.tmall.com	4
3247	marinetraffic.com	
3320	alexa.com	1, 7
3387	3bmeteo.com	1
3664	citethisforme.com	1
3708	newgrounds.com	3
3719	careerbuilder.com	
3878	alternativeto.net	
4020	tes.com	2
4054	myspace.com	1, 3
4336	interpals.net	
4344	591.com.tw	1, 5
4391	freecharge.in	5
4432	thenounproject.com	
4520	plataformaarquitectura.cl	
4796	byu.edu	1, 2, 7
4806	residentadvisor.net	
4937	pcloud.com	
4977	wgmods.net	
5082	healthunlocked.com	2
5089	ebay.com.hk	3
5222	gktoday.in	1
5496	zalo.me	4
5720	tvtime.com	1, 10
5859	publico.es	
5941	jav101.com	
6029	society6.com	3
6067	gbatemp.net	
6162	lepoint.fr	1
6212	bizoninvest.com	1, 3, 4
6242	main.jp	1, 7
6318	pinkbike.com	1, 3
6435	teamwork.com	2
6581	aptoide.com	3
6730	asiae.co.kr	
7078	giallozafferano.it	1, 3, 9
7561	b17.ru	2, 9
7610	avon.com	
7682	qobuz.com	3
7853	typeform.com	3, 4
8018	asos.fr	

Alexa Rank	Domain	Observations
8092	research.net	1
8124	reg.ru	1
8178	geocaching.com	
8197	surfearner.com	1
8326	geogebra.org	1, 3
8610	audiofanzine.com	3
8699	postype.com	1
8854	mysku.ru	3
8872	pinkoi.com	3, 4
9100	vktarget.ru	1, 7
9243	womantalk.com	
9297	toolbox.com	9
9560	dobreprogramy.pl	
9606	free3d.com	
9608	antena3.com	2, 8
9625	kinja.com	
9640	wacom.com	3, 4
9685	listal.com	3
9897	kwork.ru	
9945	laravel-china.org	
9998	tekstowo.pl	

Table C.1: Domains used for investigation of the account creation procedures and the results.

1. Account creation is required, clicking a specific element is the entry point to enter this procedure.
2. Non-HTML form usage.
3. CAPTCHA usage.
4. Email/SMS verification.
5. Telephone number is required.
6. Credit card or bank account number is required.
7. Existing account needed which can not be created.
8. Authentication failed with unclear reason.
9. Broken implementation.
10. Accounts need to be created in an app.

Domain	Classification	Observations
juejin.im	detection failed	
elPeriodico.com	login	1
codeproject.com	other	2
openstreetmap.org	registration	
coop-land.ru	registration	3
openedu.ru	join mailing list	4
hh.ru	registration	
jusbrasil.com.br	registration	
ask.fm	registration	5
gisher.org	password/login recovery	
fotostrana.ru	detection failed	
qiita.com	registration	5
cadenaser.com	other	6
Lynda.com	login	2
mediaexpert.pl	registration	1
mysku.ru	contact/comment	
wikium.ru	join mailing list	8
hinative.com	contact/comment	
dmm.com	login	7
ImmobilienScout24.de	other	
Miami Herald.com	contact/comment	
pagesjaunes.fr	join mailing list	7
osreinos.com	other	
carousell.com	registration	
stopgame.ru	contact/comment	
walkerland.com.tw	contact/comment	1, 5
mlive.com	other	5, 7
reg.ru	join mailing list	
toster.ru	registration	
bigbasket.com	other	7
orcid.org	other	7
philips.com	registration	5
dream-singles.com	order/add to cart	
buquebus.com	detection failed	
cssbuy.com	registration	5
imgur.com	contact/comment	
wikia.com	registration	6
nicovideo.jp	registration	
alibaba.com	other	5
vinted.fr	join mailing list	6
corriere.it	registration	1, 5
pearltrees.com	registration	
techrepublic.com	registration	
mintmanga.com	registration	
windguru.cz	login	
uefa.com	registration	9
larazon.es	registration	9

Domain	Classification	Observations
onesignal.com	registration	5
rozee.pk	registration	
hackerearth.com	other	
fossil.com	password/login recovery	5
gusto.com	registration	5
momomall.com.tw	login	
resumegenius.com	detection failed	
podbean.com	registration	
urcosme.com	contact/comment	
bananamall.co.kr	registration	5
boulanger.com	registration	
morhipo.com	registration	9
jarir.com	registration	2
yesstyle.com	join mailing list	
buenastareas.com	password/login recovery	
kimovil.com	detection failed	
adorama.com	password/login recovery	
ucoz.net	registration	
mos.ru	registration	
foxford.ru	detection failed	
mvideo.ru	detection failed	
proprofs.com	password/login recovery	
neowin.net	registration	
lostfilm.tv	detection failed	
tving.com	detection failed	
hatenablog.com	registration	
creativecow.net	password/login recovery	4
realtor.ca	detection failed	
fiverr.com	registration	
irecommend.ru	registration	
tripadvisor.fr	other	
codechef.com	registration	
game.co.uk	password/login recovery	5
agame.com	detection failed	
fifa.com	detection failed	
vide-greniers.org	other	
expedia.ca	other	
smutty.com	registration	
mackolik.com	Can determine what form is filled	
allaboutcircuits.com	detection failed	
halloweencostumes.com	registration	
myshared.ru	detection failed	
accorhotels.com	other	7
giantbomb.com	registration	
career.ru	registration	
stoloto.ru	detection failed	

Domain	Classification	Observations
pons.com	other	
eodev.com	login	
mangahere.cc	login	
geizhals.de	search	5
thestar.com	registration	5
yonhapnews.co.kr	detection failed	

Table C.2: Domains for the forms used for testing form processing.

1. Submit failed.
2. Unchecks checkboxes.
3. Eternal loop on radio buttons.
4. Wrong submit button.
5. Miss classified input.
6. Can not determine if custom checkboxes are processed correctly or failed.
7. Wrong form processed.
8. Crash on CSS selector, caused on too many spaces in the markup in the markup.
9. Field(s) not processed or detected.

Appendix D

SSSO Verification Experiments

rank	domain	verification options
109	booking.com	DOM, visible sign up, sign up
175	ebay.de	DOM, cookies
318	mi.com	1
392	olx.ua	DOM, visible logout, logout
639	uzone.id	DOM, visible login, login, visible logout, logout
659	yandex.com.tr	DOM, visible login, login
667	fb.ru	redirected
673	bild.de	DOM, visible logout, logout
683	trustpilot.com	DOM
755	codepen.io	2
843	v2ex.com	visible logout, logout
883	mynet.com	DOM, cookies
1029	britannica.com	visible logout, logout, visible sign up
1033	doodle.com	DOM, visible login, login, redirected
1038	prnt.sc	DOM, visible login
1058	abc.net.au	cookies
1131	online-convert.com	DOM, visible login, login, visible logout, logout, visible sign up, sign up
1227	retailmenot.com	DOM, logout, visible sign up, cookies
1294	mangakakalot.com	3
1735	lavanguardia.com	4
1809	vemale.com	5
1867	nymag.com	6
1869	tripadvisor.co.uk	DOM
1922	oantagonista.com	6
2117	gumtree.co.za	DOM, logout
2202	k2s.cc	visible logout, logout
2451	hamgardi.com	DOM
2693	tunein.com	visible account, account, visible sign up, sign up
2942	gogoanime.in	visible login, login, visible logout, logout, visible sign up, sign up
3009	uploadboy.com	visible logout, logout, visible account, account, visible sign up, sign up
3093	manualslib.com	DOM, visible login, logout, cookies
3219	pensador.com	DOM, visible login, login, visible logout, logout, login form, cookies
3247	marinetraffic.com	DOM, visible logout, logout, visible account, account, cookies
3387	3bmeteo.com	6

rank	domain	verification options
3438	indeed.co.in	DOM, visible logout, logout, visible account, account
3443	lightinthebox.com	visible login, login, visible logout, logout, login form
3499	top1health.com	visible login, login, visible logout, logout
3505	hola.org	DOM, visible logout, logout, visible account, account, visible sign up, sign up, cookies
3658	themuse.com	DOM, logout, visible sign up, sign up
3664	citethisforme.com	logout
3747	topuniversities.com	DOM, logout
3878	alternativeto.net	visible logout, logout
3988	esporte.uol.com.br	DOM, visible logout, logout, login form, cookies
4391	freecharge.in	7
4665	daftsex.com	visible logout, logout
4674	bible.com	DOM, visible logout, logout, visible sign up, sign up, redirected
4739	digit.in	login, logout
4806	residentadvisor.net	DOM, logout, visible sign up
4962	svyaznoy.ru	5
4975	bitrix24.net	visible login, login, logout, login form
4977	wgmods.net	DOM, visible login, logout
5089	ebay.com.hk	DOM, visible login, login, cookies
5151	podio.com	DOM, visible logout, logout, visible account, account, redirected
5282	elsiglodetorreon.com.mx	visible login, visible logout, logout
5295	heureka.sk	DOM
5340	ntv.ru	visible login, login
5633	rome2rio.com	visible login, login, sign up, login form
5659	klix.ba	1
5736	citethisforme.com	logout
5969	2ememain.be	DOM, visible login, login, visible logout, logout
6057	ad.nl	6
6329	homeaway.com	DOM, visible login, logout, account
6337	rusplt.ru	1
6567	forumfree.it	DOM, visible login, visible logout, logout
6572	telegraaf.nl	6
6589	rabota.ua	DOM, visible login, login, logout, sign up
6661	hunker.com	visible login, login
6712	lastminute.com	DOM
6730	asiae.co.kr	cookies
6784	seedr.cc	cookies
6858	zoon.ru	1
6971	olx.ph	7
7002	gordonua.com	DOM, cookies
7610	avon.com	DOM
7679	longdo.com	DOM visible account, account, login form
7739	roosterteeth.com	visible login, login, logout, visible sign up, sign up
7853	hackerearth.com	DOM, visible login, login, login form, cookies, redirected
7855	teepublic.com	8
7869	dochub.com	6
7903	buenastareas.com	DOM, logout, cookies

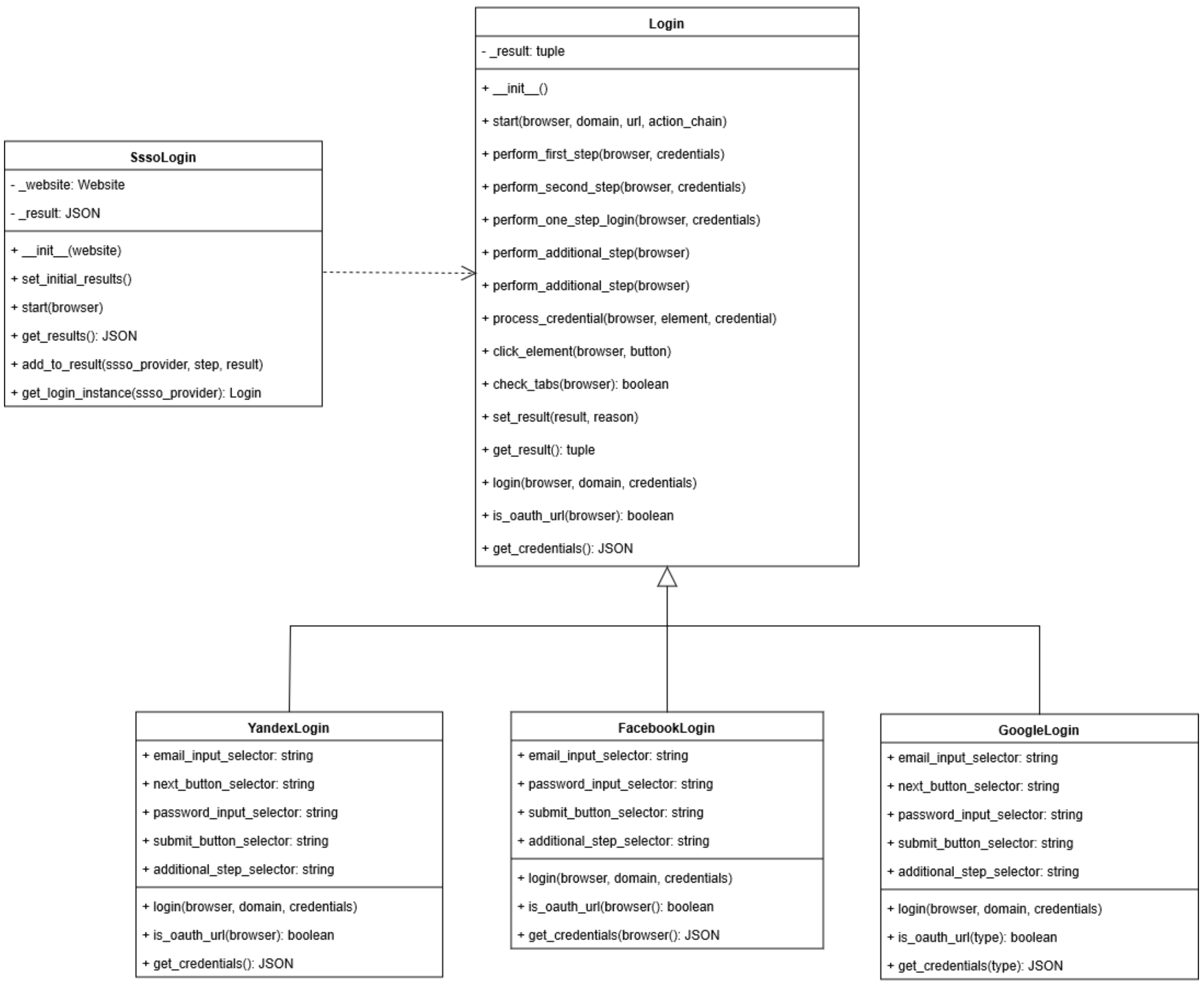
rank	domain	verification options
8018	asos.fr	6
8096	goodhouse.ru	visible login, cookies
8124	reg.ru	6
8240	travelask.ru	DOM, visible login, login, logout
8277	hdlava.me	visible login, login, visible logout, logout
8347	wikihow.it	DOM, visible login, login, cookies
8591	dokumen.tips	visible login, login, visible logout, logout, visible sign up, sign up
8869	gingersoftware.com	DOM, visible login, visible logout
8872	pinkoi.com	DOM, visible login, logout, account, visible sign up, sign up
8935	igromania.ru	5
9098	myprotein.com	DOM, visible logout, logout
9560	dobreprogramy.pl	DOM, login
9608	antena3.com	6
9614	womany.net	DOM, visible login, login
9625	kinja.com	1
9634	philstar.com	6
9761	brilliant.org	visible login, logout, account, visible sign up, sign up, redirected
9840	beyazperde.com	5
9897	kwork.ru	DOM, visible login, login, logout, visible sign up, sign up
9998	tekstowo.pl	6

Table D.1: Domains used for testing Shepherd's verification module.

1. Can not verify, none of our methods worked.
2. Error in Shepherds framework.
3. Failed due to caching issue.
4. Failed to detect log in element. (Might be due to language.)
5. Failed to detect credentials in DOM. (Often image elements.)
6. The session is not restored after loading the cookies and refreshing the browser.
7. The authentication process failed.
8. Could not load page, detected as bot.

Appendix E

UML Class Diagrams



SssoLogin
- _website: Website
- _result: JSON
+ __init__(website)
+ set_initial_results()
+ start(browser)
+ get_results(): JSON
+ add_to_result(ssso_provider, step, result)
+ get_login_instance(ssso_provider): Login
Login
- _result: tuple
+ __init__()
+ start(browser, domain, url, action_chain)
+ perform_first_step(browser, credentials)
+ perform_second_step(browser, credentials)
+ perform_one_step_login(browser, credentials)
+ perform_additional_step(browser)
+ perform_additional_step(browser)
+ process_credential(browser, element, credential)
+ click_element(browser, button)
+ check_tabs(browser): boolean
+ set_result(result, reason)
+ get_result(): tuple
+ login(browser, domain, credentials)
+ is_oauth_url(browser): boolean
+ get_credentials(): JSON
YandexLogin
+ email_input_selector: string
+ next_button_selector: string
+ password_input_selector: string
+ submit_button_selector: string
+ additional_step_selector: string
+ login(browser, domain, credentials)
+ is_oauth_url(browser): boolean
+ get_credentials(): JSON
FacebookLogin
+ email_input_selector: string
+ password_input_selector: string
+ submit_button_selector: string
+ additional_step_selector: string
+ login(browser, domain, credentials)
+ is_oauth_url(browser(): boolean
+ get_credentials(browser(): JSON
GoogleLogin
+ email_input_selector: string
+ next_button_selector: string
+ password_input_selector: string
+ submit_button_selector: string
+ additional_step_selector: string
+ login(browser, domain, credentials)
+ is_oauth_url(type): boolean
+ get_credentials(type): JSON

Figure E.1: Simplified UML class diagram for the SSSO login module in Shepherd only showing a subset of all login subclasses.

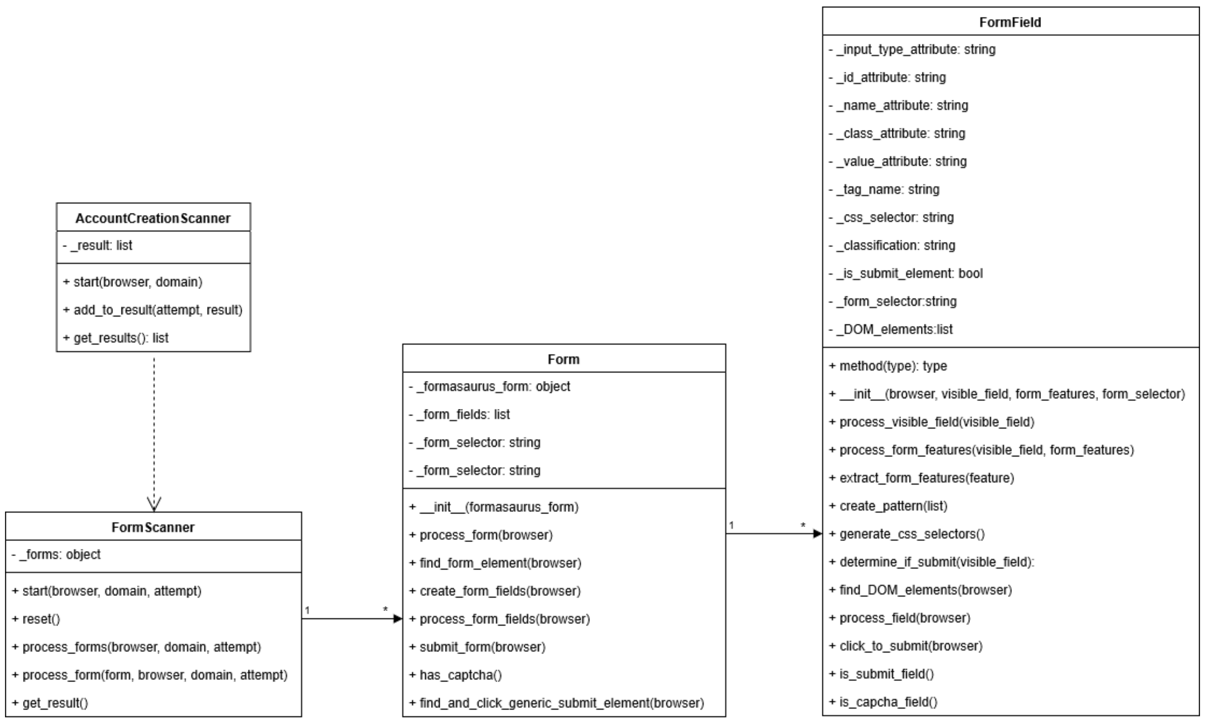

AccountCreationScanner
- _result: list
+ start(browser, domain)
+ add_to_result(attempt, result)
+ get_results(): list
FormScanner
- _forms: object
+ start(browser, domain, attempt)
+ reset()
+ process_forms(browser, domain, attempt)
+ process_form(form, browser, domain, attempt)
+ get_result()
1
*
Form
- _formasaurus_form: object
- _form_fields: list
- _form_selector: string
- _form_selector: string
+ __init__(formasaurus_form)
+ process_form(browser)
+ find_form_element(browser)
+ create_form_fields(browser)
+ process_form_fields(browser)
+ submit_form(browser)
+ has_captcha()
+ find_and_click_generic_submit_element(browser)
1
*
FormField
- _input_type_attribute: string
- _id_attribute: string
- _name_attribute: string
- _class_attribute: string
- _value_attribute: string
- _tag_name: string
- _css_selector: string
- _classification: string
- _is_submit_element: bool
- _form_selector:string
- _DOM_elements:list
+ method(type): type
+ __init__(browser, visible_field, form_features, form_selector)
+ process_visible_field(visible_field)
+ process_form_features(visible_field, form_features)
+ extract_form_features(feature)
+ create_pattern(list)
+ generate_css_selectors()
+ determine_if_submit(visible_field):
+ find_DOM_elements(browser)
+ process_field(browser)
+ click_to_submit(browser)
+ is_submit_field()
+ is_capcha_field()

Figure E.2: Simplified UML class diagram for the account creation module in Shepherd

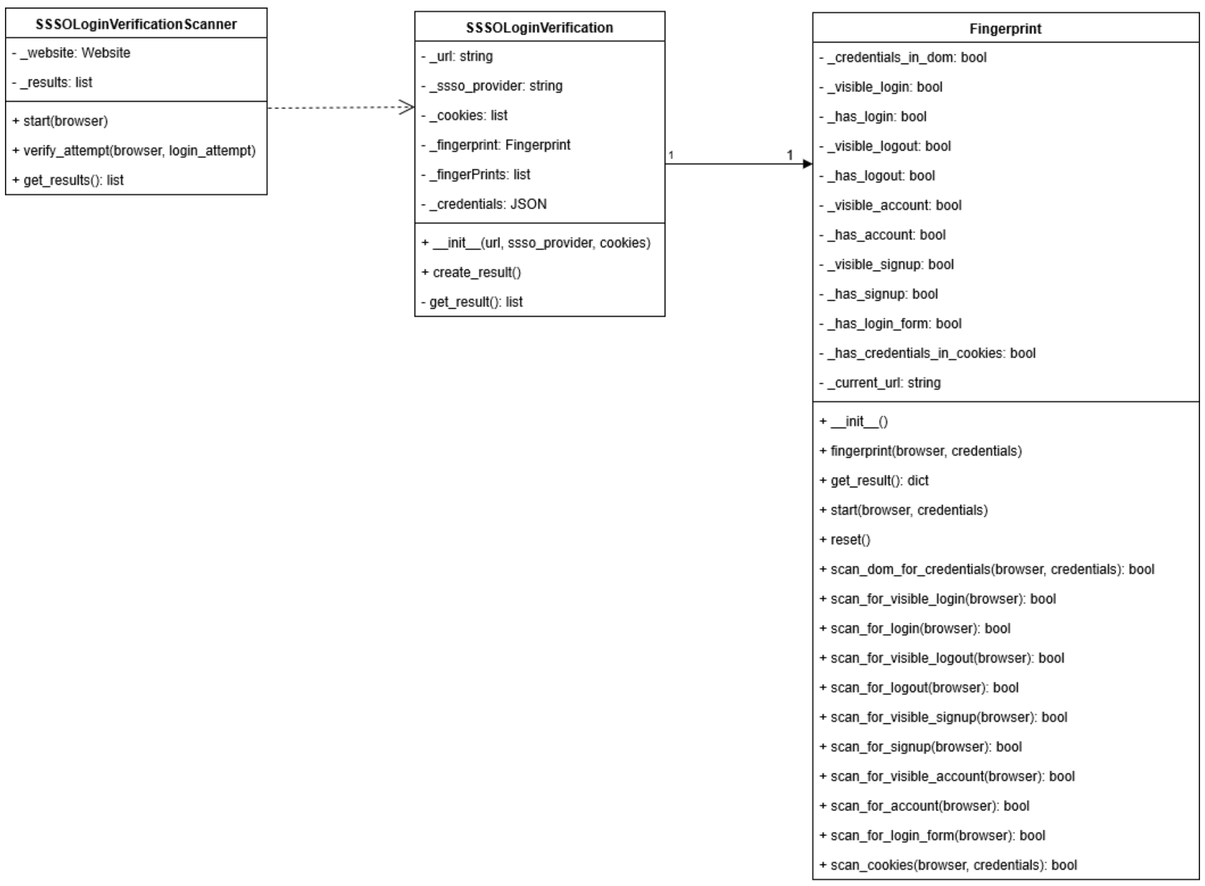

SSSOLoginVerificationScanner
- _website: Website
- _results: list
+ start(browser)
+ verify_attempt(browser, login_attempt)
+ get_results(): list
SSSOLoginVerification
- _url: string
- _ssso_provider: string
- _cookies: list
- _fingerprint: Fingerprint
- _fingerPrints: list
- _credentials: JSON
+ __init__(url, ssso_provider, cookies)
+ create_result()
- get_result(): list
1
1
Fingerprint
- _credentials_in_dom: bool
- _visible_login: bool
- _has_login: bool
- _visible_logout: bool
- _has_logout: bool
- _visible_account: bool
- _has_account: bool
- _visible_signup: bool
- _has_signup: bool
- _has_login_form: bool
- _has_credentials_in_cookies: bool
- _current_url: string
+ __init__()
+ fingerprint(browser, credentials)
+ get_result(): dict
+ start(browser, credentials)
+ reset()
+ scan_dom_for_credentials(browser, credentials): bool
+ scan_for_visible_login(browser): bool
+ scan_for_login(browser): bool
+ scan_for_visible_logout(browser): bool
+ scan_for_logout(browser): bool
+ scan_for_visible_signup(browser): bool
+ scan_for_signup(browser): bool
+ scan_for_visible_account(browser): bool
+ scan_for_account(browser): bool
+ scan_for_login_form(browser): bool
+ scan_cookies(browser, credentials): bool

Figure E.3: Simplified UML class diagram for the verification module in Shepherd

Appendix F

Looking back on the project

F.1 Jelle

My biggest challenge within this project was to decide what my contributions should be, that is, what my research should be. The Shepherd project has numerous aspects. The initial assignment was to extend Shepherd with a module to use Single Sign-On to authenticate to websites.

Without any clear decision made Alan focused on detection of SSSO and contribution to the framework. I decided to study all three main key aspects of automated authentication with SSSO, namely: The actual authentication to an SSSO provider. To what extent it is possible to solve the problem of automated account creation in the context of the Shepherd project. How the verification methods of the authenticated state could be improved. Deliver working modules for these three key aspects. These problems tend to be more pragmatic than scientific in my opinion, the scientific component is more a large scale study what we failed to accomplish.

The difficulties within the team resulted that I got quite unhappy in the project and at times frustrated. In my opinion this were mostly caused by the slow progress we made and at the end of the project still missing a large scale study that would have made our work a much more substantial to the Shepherd project. This has left me unsatisfied.

From this I learned that I should address problems within the team at an earlier stage to protect the project and myself.

I am still prepared to do any supplementary work to the thesis or project to make our contributions more substantial. For example: Still try to perform a large scale scan. Improvements to the thesis. Improvements to the Shepherds SSSO login modules. This should be accompanied by strict agreements and deadlines.

This project was my first encounter with the field of web scraping and Python. Web scraping is an interesting field and touches my current field of interest in web development. I gained some first hand experience with Python and I liked the language.

From personal experience from working in small teams I tend to prefer an agile approach. I have had good experience with this in small teams, 2- 10 members. By using small feature branches that are reviewed and done within a couple of weeks, fast progress is made early and other developers could use each other code soon and get more involved. In this project, for example, a lot of framework code was pushed to the repository after 10 months, and then it was available to start building on top of this. Reviewing each other's code is also something I highly recommend in this type of project to keep everyone up to date, this might be especially true for students. In my experience progress is faster and wrong decisions in these types of projects are earlier detected and solved.

In my opinion, Alan is a capable Python developer with a good sense for well thought over implementation and architectural solutions, and his experience with web scraping was a good contribution to this project. His reliability is his main shortcoming what might be caused by little job experience or his burnout. This is one of the causes of slow progress in this project and is something that Alan must improve in order to function well within a team. In my experience his current reliability is not accepted in a job situation.

F.2 Alan

Overall, this thesis project has been a very bittersweet experience. There was a lot of frustration, too much time and effort was spend on this project, but at the same time, I learned a lot, and it really got me interested in research.

I think a lot of the experienced problems are inherent to the subject matter, and its complexity. Generic web scraping, dealing with the Web and browser technologies, and the heterogeneity of websites are all difficult things to handle. When we decided to take on this project, we couldn't have anticipated this unavoidable obstacle of complexity.

Another source of problems was the different technologies that were used during this project. Selenium, webdrivers, and browsers are all complex technologies and sadly, not without flaws. Throughout this project, we constantly had to struggle with the instability and bugginess of these technologies. We had to extend browsers with extensions to achieve some of the wanted behaviour, and find workarounds to circumvent the limitations of these technologies.

Last but not least, our thesis project lacked a real sense of direction since the beginning. For too long, my focus wasn't on doing science, but on building a software product. Not because I wanted to, but because it was a necessary step to implement a proper software solution to tackle our research problem. It wasn't until after more than a year that I was able to really pinpoint a subject to focus upon. In the mean time, I had worked on so many different problems, that it feels like I worked on 3 or 4 different theses.

I was very ambitious when we started this project, but these problems made it difficult for me to consistently keep working on this project. 6 months or so into the project, this eventually led to a heavy burnout, where I could not work on the thesis project as much as I should have. Although I did keep working on the project most of the time. I did not stay in contact with Jelle and the rest of the team as much as I should have. A lot of the time, there was not much to say, but there definitely were moments where I should've been available, but wasn't. I feel sorry for Jelle, as I think that my actions might've resulted in this project taking longer, postponing his graduation.

A book called 'Writing for Computer Science' by Justin Zobel really helped me make heads or tails of everything. It helped me understand how to approach my research problem from a scientific perspective: formulate hypotheses, collect evidence, and bind everything together with proper arguments. It was the first time that I finally knew what I was supposed to do. It's a shame that I only started reading this book after a year had already passed. Perhaps a course that really explored doing research, designing and analyzing experiments, etc. would've better prepared me.

There wasn't enough time left to bring this project to a conclusion that I'm satisfied with, but I'm also relieved that I can leave this behind me.