OAuth 2.0 Redirect URI Validation Falls Short, Literally

Tommaso Innocenti
Northeastern University
Boston, MA, USA

Matteo Golinelli
University of Trento
Trento, Italy

Kaan Onarlioglu
Akamai Technologies
and Northeastern University*
Cambridge, MA, USA

Ali Mirheidari
Independent Researcher
Austin, TX, USA

Bruno Crispo
University of Trento
Trento, Italy

Engin Kirda
Northeastern University
Boston, MA, USA

ABSTRACT

OAuth 2.0 requires a complex redirection trail between websites and Identity Providers (IdPs). In particular, the "redirect URI" parameter included in the popular Authorization Grant Code flow governs the callback endpoint that users are routed to, together with their security tokens. The protocol specification, therefore, includes guidelines on protecting the integrity of the redirect URI.

In this work, we analyze the OAuth 2.0 specification in light of modern systems-centric attacks and reveal that the prescribed redirect URI validation guidance exposes IdPs to path confusion and parameter pollution attacks. Based on this observation, we propose novel attack techniques and experiment with 16 popular IdPs, empirically verifying that the OAuth 2.0 security guidance is under-specified. We finally present end-to-end attack scenarios that combine our attack techniques with common web application vulnerabilities, ultimately resulting in a complete compromise of the secure delegated access that OAuth 2.0 promises.

KEYWORDS

OAuth 2.0, redirect URI, path confusion, parameter pollution, account takeover

ACM Reference Format:
Tommaso Innocenti, Matteo Golinelli, Kaan Onarlioglu, Ali Mirheidari, Bruno Crispo, and Engin Kirda. 2023. **OAuth 2.0 Redirect URI Validation Falls Short, Literally. In *Annual Computer Security Applications Conference (ACSAC '23), December 04–08, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3627106.3627140**

1 INTRODUCTION

OAuth 2.0 is an industry-standard delegated access protocol allowing Internet users to grant a web application access to their data hosted on a third-party server. The most widely-used mechanism provided by OAuth 2.0, the *Authorization Code Grant* flow, involves multiple interactions between a *Client* application requesting access to external data and an *Identity Provider (IdP)*[1], where sensitive parameters need to be securely transferred and processed by each party. As a result, security analysis of OAuth 2.0 flows is an active research area, with a steady stream of practical vulnerabilities being discovered and mitigated (e.g., [14, 22, 27, 33]).

Notably, after the Client forwards a user's browser to the IdP and the user authorizes the data access, the IdP must redirect the browser back to a callback endpoint on the Client site. The Client communicates this endpoint to the IdP via the `redirect URI` parameter defined in the protocol. The request sent to this callback endpoint contains security tokens, so ensuring the integrity of `redirect URI` is paramount. Consequently, Clients must register their callback endpoint with the IdP during their setup. IdPs must validate during each OAuth 2.0 flow that the supplied redirect URI matches that registered endpoint. Unsurprisingly, exploiting OAuth 2.0 flows by abusing the `redirect URI` parameter has been heavily explored, and security guidelines integrated into the protocol specification [16, 17].

In this paper, we revisit `redirect URI` abuse in light of the lessons learned from emerging systems-centric web attacks, where vulnerabilities stem from the discrepancies between how different system components parse the same URI (e.g., [1, 18]). In particular, we observe that the RFC guidance available for Clients and IdPs narrowly focuses on protecting the integrity of the domain name included in `redirect URI` alone, but not the entire URI. We hypothesize that the RFCs' URI validation guidance is hazardously under-specified. We then explore novel mechanisms to attack OAuth 2.0 flows by abusing `redirect URI` path components and query string arguments.

Our experiments with 16 major IdPs show that they expose vulnerabilities due to insufficient validation of `redirect URI`, even under the charitable assumption that they follow the relevant RFCs flawlessly. Specifically, 6 IdPs are vulnerable to path confusion, and 10 are vulnerable to parameter pollution attacks. Using these vulnerabilities as novel exploit building blocks and combining them with other Client and IdP vulnerabilities, we show that sensitive OAuth 2.0 parameter leakage leading to complete account takeover attacks is viable. Ultimately, we confirm that the existing security guidance is insufficient and that a passing score from compliance

*The work described in this paper was performed solely at Northeastern University.

[1]We note that Identity Provider is not strictly OAuth 2.0 terminology, roughly replacing the components Authorization Server and Resource Server defined in the respective RFC. Nevertheless, the term IdP is often used in literature to simplify the discussion and better capture the common model where delegated authorization and identity services are combined in a single provider service. In this paper, we also use this simplified terminology for brevity.

ACSAC '23, December 04–08, 2023, Austin, TX, USA

ACM ISBN 979-8-4007-0886-2/23/12…$15.00
https://doi.org/10.1145/3627106.3627140

check frameworks (e.g., the recently published OAuch [25]) does not necessarily reflect good security.

Following a coordinated disclosure process, we have shared our findings with the impacted parties. We have also identified the parts of the OAuth 2.0 specification where `redirect URI` validation requirements are under-specified, leading to the vulnerabilities we have discovered and made recommendations to the OAuth Working Group for improvements to the protocol specification.

We summarize the contributions of this research below.

- We explore path confusion and parameter pollution in the context of OAuth 2.0.
- We run experiments with 16 IdPs, confirming that insufficient `redirect URI` validation issues impact them.
- We discuss practical attack scenarios and empirically demonstrate how `redirect URI` validation issues can be exploited for account takeover attacks.
- We demonstrate that the existing OAuth 2.0 security guidance is insufficient, and make concrete recommendations to improve the security of OAuth 2.0 Clients and their users.

Availability. We make the tools described in this work publicly available[2].

Ethics. We have conducted all experiments, exploit proofs-of-concepts, and disclosure of our findings in an ethical manner. For details, please see Section 7.

2 BACKGROUND

2.1 OAuth 2.0

OAuth 2.0 is a secure delegated access framework that enables *Resource Owners* to grant a *Client* access to their data hosted on a third-party *Resource Server*. The authorization is granted via interactions with an *Authorization Server* in lieu of sharing the Resource Owner's credentials with the Client. OAuth 2.0 defines four grant types, *Authorization Code Grant* being a common one suitable for environments where the Client can interact with the Resource Owner's user agent [12]. This grant flow enables the common web application deployment model where Internet users (i.e., Resource Owner) can enable web applications (i.e., Client) access to their external data by authenticating to an Identity Provider (i.e., often a combination of federated authentication services, Authorization Server, and Resource Server).

The Client must first establish a trust relationship with the Identity Provider (IdP) by registering its application. This process includes setting up a callback endpoint called `redirect URI`. In turn, the IdP issues a unique `client ID` and `client secret` to the Client. We summarize the rest of the authorization code grant flow in Figure 1 and describe each step below.

Authorization Process. (1) The flow starts when the user visiting the Client site asks to authenticate with a specific IdP, and (2) the Client redirects the user's browser to the IdP login endpoint. (3) This request to the IdP is called the *Authorization Request* and it commonly includes the following parameters: i) `response type = code`, specifying the authorization code grant type, ii) the previously issued `client ID`, a public Client identifier, iii) `state`, used as a Cross-Site Request Forgery (CSRF) defense, iv) `redirect URI`, used to redirect the browser back to the Client application after the user has granted or denied authorization.

(4) Once the browser is redirected to the IdP, the user authenticates on the IdP using their credentials and authorizes the Client to access their data. During this step, the IdP validates the parameters included in the Authorization Request. In particular, `redirect URI` is validated against the one Client provided during their registration. (5) If the validation succeeds, the IdP redirects the browser back to the Client endpoint specified in `redirect URI`. (6) The resulting *Authorization Response* includes a fresh authorization code (i.e., `code`) and the earlier `state`. The Client validates the `state` bound to the user's session, ensuring there is no CSRF attack.

Redeem Process. The `code` does not directly grant access to the user's resources. (7) The Client instead uses it to redeem an `access token` by making an *Access Token Request* to the IdP. This request includes the following parameters: i) `client ID`, ii) `grant type = authorization_code`, iii) `client secret`, iv) the `code` received in the Authorization Response, and v) the same `redirect URI` used in the Authorization Request. Upon receiving this, the IdP authenticates the Client using `client secret`, verifies that `code` was issued to this Client and was not used before, and checks that `redirect URI` is identical to the one included in the Authorization Request. (8) If all checks succeed, the IdP issues an `access token` to the Client. Notably, the same `code` cannot be used again.

Data Access. Finally, (9) the Client can access the user's protected resources with `access token`, where the IdP must verify that the token has not expired.

2.2 Related Work

OAuth 2.0 comes at the cost of a complex redirection trail between all parties involved in the protocol. The data flows must be secured in flight, and sensitive parameters validated at each endpoint.

Researchers began investigating the protocol from the early days using formal methods [6, 24]. This research culminated in the work of Fett et al., which identified multiple protocol-level vulnerabilities such as *IdP Mix-Up* and *307 Redirect* [7].

`redirect URI` is a natural target for abuse, and researchers have explored ways to redirect users to malicious domains [22]. Consequently, in 2017, the first draft of the OAuth 2.0 Security Best Current Practice formally addressed `redirect URI` validation requirements [16]. However, as future work demonstrated, this validation is insufficient, and abusing the discrepancies in URI parsers still makes it possible to hijack OAuth 2.0 flows [33]. Recently, OAuch presented a framework to verify the implementation correctness of IdPs, including validating `redirect URI` [25]. Only 34% of IdPs were shown to perform a correct validation.

With OAuth 2.0's sustained adoption, researchers have also discovered a flood of Client-side implementation flaws [9, 20, 31, 34]. In particular, Clients' mishandling of `state` has led to widespread Login CSRF vulnerabilities [4, 29]. Even when IdPs provided the Client developers with SDKs, implicit security assumptions and poor documentation resulted in continued implementation issues [32]. Similarly, recent research demonstrated that the complexity of supporting both SSO login protocols and traditional authentication methods

[2] https://github.com/innotommy/OAuthpaper-code

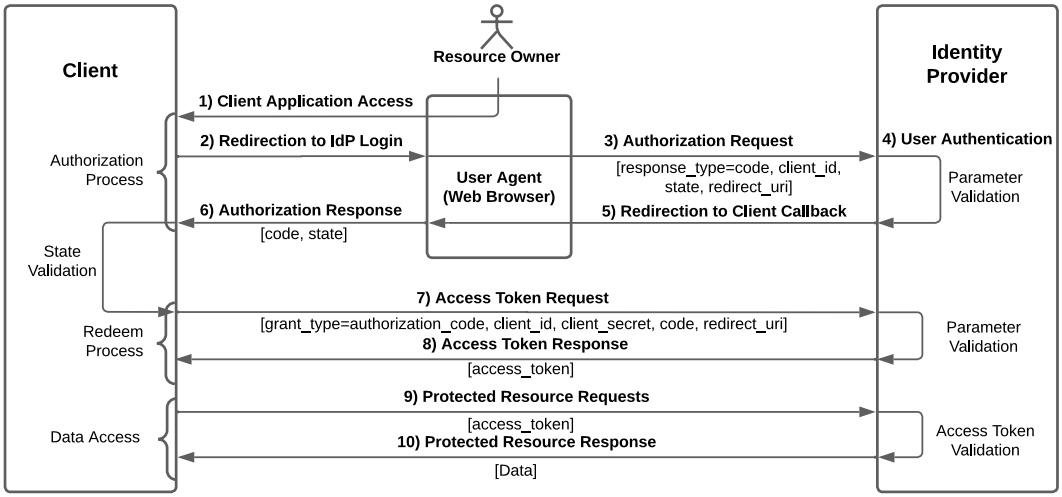

Figure 1: OAuth 2.0 Authorization Code Grant Flow.

in a Client, with intermingled paths, can lead to new classes of attacks where an attacker can *pre-hijack* a victim's account before the victim interacts with the Client [8, 30].

A further OAuth 2.0 integration challenge is the security of the Client endpoint. As the RFC spells out, including untrusted third-party scripts in Client endpoints that have access to sensitive OAuth 2.0 tokens is dangerous [12]. As demonstrated by Frans Rosén and selected as the top hacking technique in 2023 by PortSwigger, attacks abusing such token leaks are viable [3, 27]. However, this attack vector has largely been ignored by the academic research community so far.

Finally, research has looked at ways to address OAuth 2.0 vulnerabilities on the browser side, for example, by using browser extensions to upgrade network connections to HTTPS [5, 15].

We present novel techniques to abuse `redirect URI`, beyond what is covered in previous work, and describe how attackers can escalate those to complex yet practical end-to-end attacks when combined with common vulnerabilities on Client sites and IdPs. Our contributions are due to fundamental gaps in the OAuth 2.0 specification, undetected by cutting-edge tools like OAuch.

3 RESEARCH STATEMENT

3.1 Motivation

As evidenced by the OAuth 2.0 literature we covered, `redirect URI` has long been recognized as a lucrative abuse target by researchers and miscreants alike. Presumably anticipating these security issues, the authors of the OAuth 2.0 protocol specification and threat model RFCs have also extensively covered `redirect URI` attacks and explicitly called out the necessity to validate that a supplied `redirect URI` matches the callback endpoint that was registered during Client setup [11, 12, 17]. Quoting the relevant sections:

> ***RFC 6749 Section 3.1.2.3*** *The authorization server MUST compare the two URIs using simple string comparison as defined in RFC 3986 Section 6.2.1.*

> ***RFC 3986 Section 6.2.1*** *Testing strings for equality is normally based on pair comparison of the characters that make up the strings, starting from the first and proceeding until both strings are exhausted, and all characters are found to be equal, until a pair of characters compares unequal, or until one of the strings is exhausted before the other.*

This `redirect URI` validation strategy describes three stopping conditions; however, it does not mandate a validation success or failure outcome for these conditions. In particular, the final condition where two URIs may have a matching prefix, but overall different lengths, is **not** expressly disallowed. Should IdPs interpret this ambiguity as an intentional flexibility granted to them (e.g., to support dynamic path components or query parameters in `redirect URI`) or otherwise inadvertently allow a non-exact string match, there are significant security implications: While this validation scheme prevents tampering with the host or domain name included in a `redirect URI`, it falls short of detecting potentially malicious additions to, deletions from, and modifications to the path components and query string that follow. The security community has recently seen a surge of attacks that utilize such *path confusion* techniques, i.e., tricks that abuse URI parsing discrepancies within complex system interactions (e.g., [1, 18]). We hypothesize that `redirect URI` can too be abused by *path confusion* due to insufficient validation.

We next observe that RFC 6749 allows query strings in `redirect URI` and further prescribes that they be retained during the protocol flow. The RFC acknowledges that malicious injections into `redirect URI` parameters are a threat and recommends that endpoints perform validation and/or sanitization on sensitive values. Quoting the relevant sections:

> ***RFC 6749 Section 3.1*** *The endpoint URI MAY include an "application/x-www-form-urlencoded" formatted (per Appendix B) query component (RFC 3986 Section 3.4), which MUST be retained when adding additional query parameters.*

> ***RFC 6749 Section 10.14*** *A code injection attack occurs when an input or otherwise external variable is used by an application unsanitized and causes modification to the application logic. This may allow an attacker to access the application device or its data, cause a denial of service, or introduce a wide range of malicious side-effects. The authorization server and Client MUST sanitize (and validate when possible) any value received–in particular, the value of the "state" and "redirect_uri" parameters.*

While this language calls out a potential attack vector via abuse of query strings, it lacks prescriptive instructions on the appropriate input validation or attack prevention steps. When combined with the requirement (i.e., MUST) that additional parameters be retained, the RFC leaves `redirect URI` open to *parameter pollution* attacks, where an attacker injects duplicates of security-sensitive parameters in a query string to, once again, abuse parsing discrepancies between different system components that process the same URI [2]. Therefore, we hypothesize that OAuth 2.0 flows can be attacked via *parameter pollution* in `redirect URI`. A quick survey indicates that we are not alone in this second observation; in fact, two security researchers Lauritz Holtmann and Youssef Sammouda independently found specific evidence of parameter pollution in OAuth 2.0, which further warrants a systematic exploration of this issue [13, 28].

We stress that both of our hypotheses are valid under the idealized assumption that Clients and IdPs follow and implement the OAuth 2.0 RFCs *correctly*. We do not rely on implementation bugs but under-specified requirements.

3.2 Research Goals

In this work, we set out to experiment with popular IdPs and test the two hypotheses mentioned earlier. We ultimately aim to answer the following research questions.

- (Q1) Is OAuth 2.0 vulnerable to path confusion attacks?
- (Q2) Is OAuth 2.0 vulnerable to parameter pollution in security-sensitive tokens?
- (Q3) How can attackers use these techniques to enable end-to-end attacks on real-life applications?
- (Q4) How can we improve the OAuth 2.0 specification to address these issues?

We tackle these questions in the rest of this work.

3.3 Threat Model

The threat model we assume in this work is that of a typical web attacker, targeting a web application.

The Client is any web application that serves Internet users and uses identity and access management services offered by an IdP via OAuth 2.0. Internet users access the Client with user agents (e.g., a web browser) installed on any networked device. All networked communications between these entities run over a secure channel, such as a modern version of TLS, which guarantees cryptographic confidentiality and integrity.

The attacker has identical privileges to regular Internet users. They can access the Client web application with their legitimately created authentication and authorization credentials. They can, therefore, also interact with the IdP via OAuth 2.0 normally.

The attacker does not have man-in-the-middle capabilities or the ability to interfere with secure communication channels. They can, however, participate in OAuth 2.0 and maliciously interact with protocol flows on their user agents, receiving messages and responding to them with any data, just like any Resource Owner could on their device. We further assume that the attacker can utilize social engineering techniques to make their victim click on malicious links.

All attacks involving unauthorized access to a victim's data are in the scope of our threat model. This includes tricking the victim into accessing an attacker-controlled resource and leaking sensitive data (e.g., a Login CSRF attack), or a more straightforward takeover of the victim's account by the attacker.

We stress that the novel abuse vectors we present in this paper are building blocks for attacks, but they are not end-to-end exploits on their own. Therefore, our threat model assumes that the targeted Clients and IdPs may include other well-known web application vulnerabilities. An attacker can then combine our new findings with existing vulnerabilities to achieve severely damaging effects, such as a complete account takeover that would otherwise not be possible. We discuss these specific preconditions where relevant in the rest of this paper.

4 BAD VALIDATION PART I: PATH CONFUSION

To test our hypothesis that the OAuth 2.0 `redirect URI` validation guidelines are insufficient and subsequently answer our research question (Q1) (see Section 3.2), we design an experiment that exercises popular IdPs with `redirect URI` parameters containing path confusion payloads. We present our methodology and results below.

4.1 Path Confusion Primer

Path confusion refers to a collection of techniques that involve appending maliciously crafted path components to a URL. This serves to confuse modern URL parsers designed to accommodate complex URL rewriting and routing mechanisms, or otherwise to induce discrepancies between multiple parsers in a complex system. Path confusion has recently been used in various attack contexts such as *Web Cache Deception* and *Relative Path Overwrite* successfully, and the research community has been developing a steady stream of new confusion techniques [1, 18, 19].

In this experiment, we aim to replace the legitimate `redirect URI` parameter in OAuth 2.0 flows with path confusion payloads, and subsequently determine which IdPs fail to detect this malicious modification through validation and proceed with the protocol. The impact of a successful attack is that the IdP redirects the victim's user agent to an unintended endpoint on the Client site. **We will explain how this capability translates to a practical attack in the rest of the paper; in this experiment, however, our immediate goal is to detect vulnerable IdPs and verify that path confusion in OAuth 2.0 is possible.**

We test each IdP with 20 distinct path confusion payloads compiled from the cited literature, shown in Figure 2. These variations

```
Client.com/callback/FAKEPATH
Client.com/callback%2FFAKEPATH
Client.com/callback/..%2FFAKEPATH
Client.com/callback/%2e%2e%2FFAKEPATH
Client.com/callback/..%252FFAKEPATH
Client.com/callback/%252e%252e%252FFAKEPATH

Client.com/callback/FAKEPATH/..
Client.com/callback%2FFAKEPATH%2F..
Client.com/callback%2FFAKEPATH%2F%2e%2e
Client.com/callback%252FFAKEPATH%252F..
Client.com/callback%252FFAKEPATH%252F%252e%252e

Client.com/callback/;/../../FAKEPATH
Client.com/callback/%3B/../../FAKEPATH
Client.com/callback/%3B%2F..%2F..%2FFAKEPATH
Client.com/callback/%3B%2F%2e%2e%2F%2F%2e%2eFAKEPATH
Client.com/callback/%253B%252F..%252F..%252FFAKEPATH

Client.com/callback/%0A%0D/../../FAKEPATH
Client.com/callback/%0A%0D%2F..%2F..%2FFAKEPATH
Client.com/callback/%0A%0D%2F%2e%2e%2F%2F%2e%2eFAKEPATH
Client.com/callback/%250A%250D%252F..%252F..%252FFAKEPATH
```

Figure 2: Path confusion payloads used in the experiment. "Client.com/callback/" represents the legitimate redirect endpoint, and the remaining components are malicious modifications. The attacker's goal is to redirect the victim to an intended FAKEPATH endpoint on the Client site, and red sections are confusion techniques including path traversal tricks, encoded special characters, and layered encoding.

combine the basic payload with path traversal tricks, encoded special characters, and multiple encoding layers to create increasingly complex URLs that trigger parser quirks and validation flaws.

4.2 Methodology

Setup. We start with a setup phase that enables us to automate OAuth 2.0 flows and `redirect URI` modifications for testing. We seed our experiment with a collection of Client sites and crawl each site in this dataset to identify their user authentication pages and the IdPs they support. This is a two-step process. First, our detection logic uses regular expressions and simple heuristics, looking for keywords (e.g., login, sign-in, join) and HTML tags (e.g., input tags of type `password`) in the page content to detect the login pages. Next, we use a second layer of similar heuristics on these pages to detect the presence of all HTML elements (e.g., buttons, hyperlinks) that start an OAuth 2.0 flow (i.e., *OAuth 2.0 triggers*). Note that a Client can support multiple IdPs; we detect and subsequently experiment with all of them. For implementation details of these heuristics, please see our publicly available source code.

At this stage, creating accounts with all identified IdPs is necessary to perform an end-to-end flow with them for experimentation. This is a manual effort where we create test accounts and provide as account details (e.g., email address, user name) unique values that we can later identify reflected on a Client callback page, which would indicate the successful completion of OAuth 2.0.

Finally, we verify our findings by exercising the OAuth 2.0 trigger we found on Client sites. Specifically, we use an *OAuth 2.0 Player* tool we developed, which automatically drives a real browser to start OAuth 2.0 from the Client site, authenticates to IdP using our test accounts, and then lands back on the Client callback endpoint. The tool verifies on the Client that all previously identified HTML elements initiate the flow, on the IdP site that the landing page is the IdP login page, and that the URL contains the necessary OAuth 2.0 parameters (e.g., `redirect URI`, `state`). We discard any OAuth 2.0 triggers that fail to pass this verification (e.g., in cases where our detection heuristics did not work as expected), and we proceed to the next phase of the experiment with the rest.

Data Collection. We once again exercise all OAuth 2.0 triggers with the OAuth 2.0 Player, but this time also utilize *a man-in-the-middle proxy* to intercept the flows and inject our path confusion payloads into the `redirect URI` parameters in flight. We test every flow separately with all 20 path confusion payloads shown in Figure 2. We collect raw dumps of all network traffic, intercepting proxy logs, browser screenshots at each step, and information regarding the presence of our unique test account identifiers on the final Client callback page.

Vulnerability Detection. In this final phase, we analyze the data collected in the previous step to determine which IdPs are impacted by path confusion payloads, meaning they perform insufficient `redirect URI` validation. More specifically, we flag IdPs that did not terminate the protocol upon receiving a maliciously modified `redirect URI` or otherwise sanitize the "FAKEPATH" marker included in our attack payloads, but instead proceeded to redirect the browser to a callback endpoint containing the same "FAKEPATH" component (i.e., the Authorization Response URL contains "FAKEPATH").

Inspecting the raw network traffic dumps for this final malicious redirect request is sufficient to identify a vulnerable IdP. The remaining data sources provide complementary signals that help verify that the user authentication to the IdP and Client authorization for data access are also performed correctly.

4.3 Experiment & Results

We performed our experiment using the above methodology, also summarized in Figure 3. We implemented the OAuth 2.0 Player using Node.js and *puppeteer* to drive the Chrome browser. We used *mitmproxy* to intercept the traffic.

We seeded the experiment with a Client dataset of the Top 15K sites of the Tranco list[3] generated on 15 February 2022 [26]. Among these, our setup crawl and heuristics detected 728 sites with an authentication page supporting at least one IdP. Because these sites used many niche IdPs, making a deep analysis of them infeasible, we focused our investigation on the most popular picks. To that end, we selected only those IdPs used by at least 3 Client sites, resulting in 28 IdPs. We further filtered out the IdPs in this set that required valid personal information to register, enforced geo-restrictions, or mandated two-factor authentication. As a result, our data collection phase started with 22 IdPs in scope. While running the OAuth 2.0 flow experiments, we ran into further issues with sites that used bot management solutions or CAPTCHAs to block automated logins. Ultimately, we ran **464 successful OAuth 2.0 flows between 378 Client sites and 16 IdPs.**

[3] https://tranco-list.eu/list/KXNW.

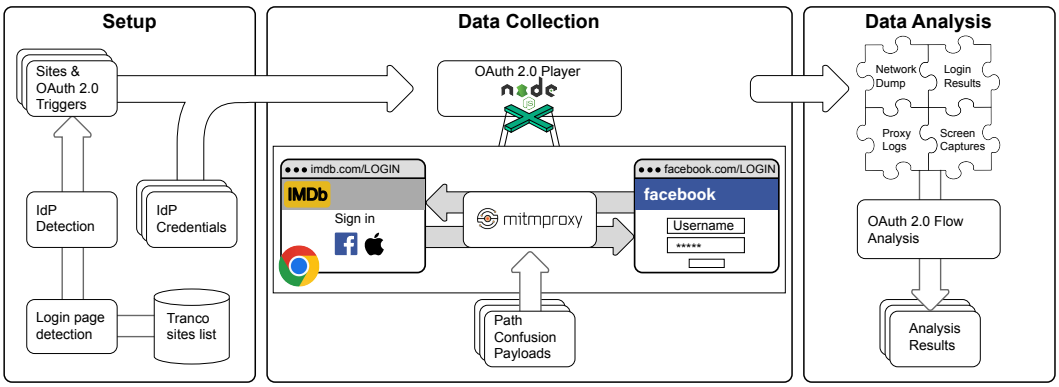

Figure 3: Experiment methodology for detecting IdPs vulnerable to path confusion attacks.

Analysis of the experimental data revealed that **6 out of the 16 IdPs we tested did not correctly validate `redirect URI`, and were exposed to *path confusion* attacks.** The vulnerable IdPs were Atlassian, Facebook, GitHub, Microsoft, NAVER, and VK. This experiment empirically confirms our hypothesis that the RFC-prescribed `redirect URI` validation strategy is insufficient and that *path confusion* attacks on OAuth 2.0 are practical. We answer our research question (Q1) affirmatively.

5 BAD VALIDATION PART II: PARAMETER POLLUTION

We now answer our next research question (Q2) (see Section 3.2) by exercising IdPs with parameter pollution payloads.

5.1 OAuth 2.0 Parameter Pollution (OPP)

HTTP parameter pollution (HPP) is a well-known web application exploitation technique where an attacker crafts a request that includes multiple parameters with identical names, but different values. The processing order for such parameters (or whether they are processed at all) is implementation dependent. The attacker can elicit unusual behavior or bypass security checks by targeting applications made up of multiple components that process the same query string inconsistently [2].

Building on previous work demonstrating parameter pollution in OAuth 2.0 (i.e., [13, 28]), and combining both observations from Section 3, that the RFC allows `redirect URI` values with differing lengths to pass validation and that IdPs are required to keep query strings intact, we set out to investigate whether HPP attacks apply to OAuth 2.0 flows more generally. We call this rendition of the attack *OAuth 2.0 parameter pollution*, or OPP.

OPP has one express goal: To influence an OAuth 2.0 flow so that, at the end of the Authorization Process, the victim is redirected to a Client callback endpoint with *two* distinct `code` parameters, one being the legitimate value, and the other injected by the attacker. We present the attack in Figure 4 and describe how it plays out below. **We emphasize that we will describe how this capability enables an end-to-end attack in the following sections. Here, our sole goal is to describe the technique and verify that IdPs are indeed impacted.**

The attacker first crafts a URL pointing to the target IdP's authorization endpoint, including all the necessary and valid query string parameters `response type = code`, `client ID`, `state`, and `redirect URI`. However, they then modify the included `redirect URI` by appending it a query parameter `code`. The value of this parameter may be an arbitrary string; or alternatively, the attacker can obtain and use a valid `code` value by starting another OAuth 2.0 flow and prematurely stopping it after the Authorization Process. In either case, the net effect is a malicious URL already containing a `code` parameter appended to its `redirect URI` parameter, shown in blue below. Note that the attacker encodes the "?" and "=" characters in the appended query string, shown in red, to minimize the chances of a parsing error on the IdP end.

```
https://idp.example.com/oauth/authorize?
  response_type=code&client_id=<valid ID>&
  state=<value>&
  redirect_uri=
  https://client.example.com/
          oauth/callback%3Fcode%3D<value>
```

Once the attack URL is ready, the attacker tricks a victim into visiting it via social engineering or injection techniques. (1) This starts a normal OAuth 2.0 flow, taking the victim's browser to the IdP's legitimate authorization page. (2) The victim logs into their account, authorizing the Client to access their data. During this step, the IdP performs validation on `redirect URI` as prescribed, but there is no reason to flag the unexpected query parameter `code`, as the prefix perfectly matches the registered `redirect URI` value, therefore passing the validation successfully. (3) Finally, the IdP takes the `redirect URI` that already includes the attacker injected `code`, keeps it intact as mandated in RFC 6749 Section 3.1, and appends to it a second `code` freshly generated for this flow. (4) Ultimately, the victim lands on the Client callback endpoint with two `code` parameters. If the Client implementation chooses to process the attacker-injected `code`, the victim's valid code remains unused, ready to be leaked via another vulnerability for an account takeover.

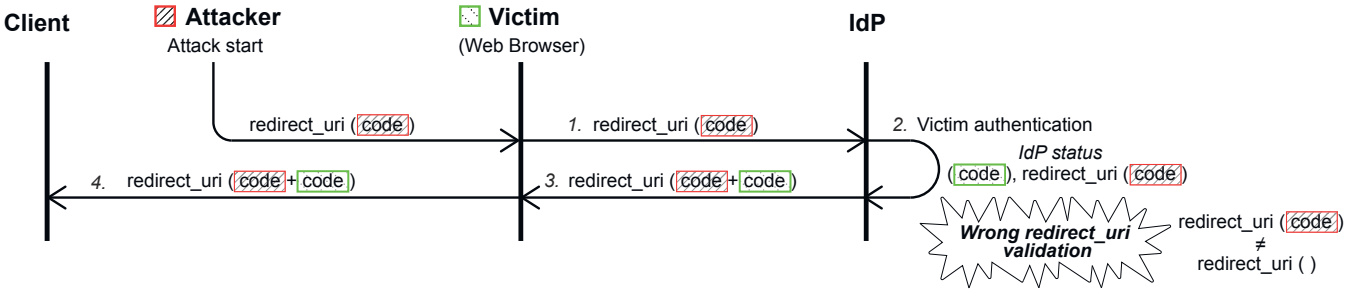

Figure 4: Attack flow for OAuth 2.0 parameter pollution.

5.2 Experiment & Results

We tested the viability of OPP by creating a simple Client application, registering it with IdPs, and participating in OAuth 2.0 with them. We replicated the conceptual attack steps described above, injecting duplicate code parameters into flows. We conducted this experiment with the same set of 16 IdPs as determined in the previous path confusion experiments; we omit those redundant phases of the methodology.

The results showed that **10 out of 16 IdPs were impacted by OPP**. They did not terminate the flow or strip away the superfluous parameter, which resulted in our browser landing on the callback endpoint with both code parameters intact. The impacted IdPs were GitHub, LINE, LinkedIn, Microsoft, NAVER, OK, ORCID, Slack, VK, and Yahoo. This experiment again confirms our hypothesis that the RFC-prescribed `redirect URI` validation is inadequate and validates the previous findings in literature. We answer our research question (Q2) affirmatively.

6 IMPACT

So far, we have presented two abuse techniques targeting IdPs that do not validate `redirect URI` correctly during the Authorization Process. This is not due to arbitrary bugs or design decisions, but they are rooted in the OAuth 2.0 specification; in other words, IdPs that strictly follow the formal validation guidance may still be vulnerable. The result is that the authorization code is delivered to a maliciously modified callback endpoint.

However, the victim is not compromised yet. For a successful end-to-end attack, two more conditions are necessary: (1) The attacker must be able to gain possession of the victim's code, and ultimately (2) redeem it for an `access token` resulting in a complete account takeover. In this section, we explain how these additional steps can be achieved in practice, what our abuse techniques contribute to the security concerns already covered in the OAuth 2.0 specification, and how we significantly expand the attack surface of applications. This addresses our research question (Q3) (see Section 3.2).

6.1 code Leakage

Exposure of sensitive OAuth 2.0 parameters to third/fourth-party code included on a callback endpoint is a concern that the protocol specification already recognizes. The RFC calls out this risk and assigns the responsibility of protecting the Authorization Response to the Client:

> ***RFC 6749 Section 3.1.2.5*** *The Client SHOULD NOT include any third-party scripts (e.g., third-party analytics, social plug-ins, ad networks) in the redirection endpoint response. Instead, it SHOULD extract the credentials from the URI and redirect the user-agent again to another endpoint without exposing the credentials (in the URI or elsewhere). If third-party scripts are included, the Client MUST ensure that its own scripts(used to extract and remove the credentials from the URI) will execute first.*

Even if a Client ignores this requirement and the code ends up being leaked, attacks are not trivial. Foremost, the attacker cannot influence the leak destination unless a very specific XSS, JavaScript inclusion, or open redirect vulnerability is already present on the precise callback page–a code leaked to an arbitrary legitimate third party is of no value to the attacker. Next, even if the attacker could gain access to the leaked code, they must then enter a tight race condition with the legitimate OAuth 2.0 flow to use the code first–the code is a short-lived, single-use token. As a result of these limitations, code leakage attacks are often not considered a relevant risk, and the research community has not focused on them.

Our attack techniques remove these limitations and make code leakage viable.

In particular, path confusion and OPP eliminate the aforementioned race condition, as the victim's code remains unused. Path confusion redirects the user to an entirely different endpoint on the Client, where the application logic does not expect an OAuth 2.0 flow, and therefore does not consume the code. OPP tricks the Client into proceeding with the flow using an attacker-injected code, leaving the victim's original code intact.

Path confusion has another powerful property. Now that the attacker can influence the callback endpoint, a data exfiltration vulnerability present *on any path* of the Client can be weaponized to compromise OAuth 2.0 and escalate to a complete account takeover. This greatly increases the attack surface of a web application, transforming (even non-exploitable) common vulnerabilities into critical security issues. For instance, an attacker can inspect a web application to find any of the below issues, on any path, and redirect their victim to that path to steal their code reliably:

- **XSS, style, or HTML injection** of any kind that allows the attacker to extract query string parameters and trigger a request to a domain they control, giving them direct access to the code.

- **Open redirect vulnerabilities**, immediately re-routing the Authorization Response to an attacker domain.
- **Multi-tenant sites**, where different entities can reside on the same domain name under different paths, and the attacker sign up as a legitimate tenant to hijack the Authorization Response.
- **Leaky third-party code inclusion,** the original threat that the OAuth 2.0 specification advice attempts to mitigate on callback endpoints, now becoming a concern across the entire Client site.

We present two real-life examples of these scenarios in more detail later in this section.

6.2 `redirect URI` Validation in Redeem Process

Once the attacker obtains the victim's `code`, they need to redeem it for an `access token`, and this step poses a final challenge. Recall from our overview of OAuth 2.0 in Section 2, Figure 1, Step (7) that the Client includes another `redirect URI` parameter in the Access Token Request. The protocol specification requires this value to match the `redirect URI` that was previously supplied in the Authorization Request:

> ***RFC 6749 Section 4.1.3*** The Client makes a request to the token endpoint by sending the following parameters [...]
> **redirect_uri** REQUIRED, if the "redirect_uri" parameter was included in the authorization request as described in Section 4.1.1, and their values MUST be identical.

This requirement implies that the attacker's modifications to the `redirect URI` in the Authorization Request must be correctly reflected in the Access Token Request. This is problematic for the attacker, because they do not have control over this second `redirect URI` parameter: The Authorization Request is sent from the User-Agent that the attacker operates, whereas the Access Token Request is issued by the Client, protected from the attacker's influence.

Once again, the quoted RFC section mandates an identical value without concrete guidance on how this validation should be performed. In light of this observation, we hypothesize that IdPs will follow the same improper `redirect URI` validation prescribed in RFC 6749 Section 3.1.2.3 (as also suggested in the OAuth 2.0 Security Best Current Practice), or otherwise, either Clients or IdPs will make arbitrary design decisions that may be hazardous.

Unfortunately, it is not feasible to explore how exactly IdPs perform the check from an external vantage point, without visibility into the IdPs' implementation. Therefore, verifying this hypothesis within a scientific framework is outside the scope of our work. Instead, we present a number of experiments that empirically demonstrate what IdPs under our lens perform the Redeem Process validation incorrectly, enabling a complete attack.

Experiments. In the first experiment, we use our Client application and perform a series of OAuth 2.0 flows against each IdP. We launch the described path confusion attack in the Authorization Request by modifying the `redirect URI`. However, we use the original, unmodified `redirect URI` in the Access Token Request. If the OAuth 2.0 completes successfully regardless of the mismatch between the two `redirect URI` values, we conclude that the IdP performs an incorrect validation action. We stress again that we cannot experimentally determine what that incorrect validation action is without observing IdP internals; this is necessarily a black box test. The second experiment follows the same methodology, but this time with an OPP attack introduced in the Authorization Request.

In both experiments, **we found the 2 IdPs GitHub and NAVER to perform insufficient validation in the Redeem Process and allow an end-to-end account takeover attack.**

In order to understand what might be happening under the hood, we explored the documentation for each service. GitHub references the `redirect URI` parameter in the Redeem Process, but the provided definition (i.e., "The URL in your application where users are sent after authorization.") is incomplete at best; this value must be required to match the `redirect URI` used in the Authorization Request. Moreover, the parameter is marked optional, even when a `redirect URI` is provided in the Authorization Request [10]. With further testing, we were indeed able to verify that entirely omitting this value also results in a successful flow. NAVER's documentation and examples did not include a `redirect URI` in the Access Token Request [21] at all. Likewise, performing a complete OAuth 2.0 with NAVER was possible when our Client provided no `redirect URI`. In either case, it was not clear whether the string matching strategy was flawed when a `redirect URI` is provided by the Clients, or whether the IdPs omitted validation on the provided values at all times. Regardless, both IdPs were exploitable in practice.

Influencing the Access Token Request. We make a final observation that depending on how real-life Clients construct the Access Token Request, an attacker may be able to influence the process, and trick the Client into re-creating an identical `redirect URI` to the attack payload. As a result, both `redirect URI` values would naturally match, *in theory* defeating all validation checks. We present an example of how this might play out with a typical Client implementation of the Access Token Request build process in Figure 5, zooming into Steps (6) and (7) in our OAuth 2.0 overview diagram previously shown in Figure 1.

On the left, we see a normal flow, where (1) the Client receives a benign Authorization Response at the correct callback endpoint, (2) parses the query string into three components `code`, `state`, and everything else that comes after as a monolithic block to capture the application-specific parameters, (3) performs the `state` check, (4) and finally constructs the new `redirect URI` by appending to the callback endpoint the previously parsed block of custom parameters. This is the expected behavior, required by RFC 6749 Section 4.1.3, so the query strings in the old and new `redirect URI` values match. On the right, we see the outcome of the same build process, but for an Authorization Response that was polluted with a superfluous `code` as a result of an OPP attack. As the figure demonstrates, the attacker-injected `code` is now treated as part of the custom parameter block, and directly copied to the new `redirect URI`, which becomes identical to the previous `redirect URI` that the attacker manipulated to trigger the OPP. The subsequent `redirect URI` validation in the IdP should find a perfect match.

Surprisingly, when we tested this scenario with the 10 IdPs vulnerable to OPP, only 6 (i.e., GitHub, LinkedIn, NAVER, OK, Slack, and VK) completed the protocol. That is, the remaining 4 IdPs refused to validate matching `redirect URI` values. This was contrary

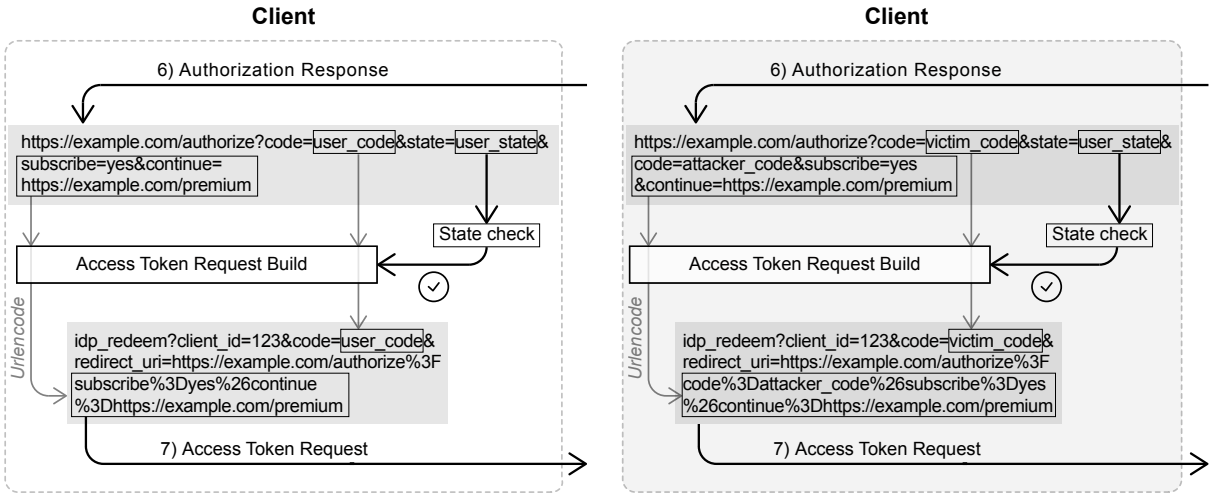

Figure 5: Typical implementation of Access Token Request build process. On the left: The Client builds the Access Token Request, correctly matching the application-specific query string parameters received in the request to the newly constructed `redirect URI`. On the right: The same process during an OPP attack results in a `redirect URI` value that matches the attack payload.

to our expectations; the two `redirect URI` values were identical, and both the RFC-prescribed validation strategy and an exact string comparison should have succeeded. This again demonstrates that IdPs may be following arbitrary validation routines designed to fill the gaps in the RFC, or maintaining a custom state about the observed `redirect URI` values, as opposed to doing a straightforward string comparison. Although that had the desirable effect of blocking the OPP attack here, non-standard validation is error-prone, and such inconsistent behavior is a common cause of hazardous interactions in systems-centric security.

6.3 Case Studies

As discussed, the real-life exploitability of insufficient `redirect URI` validation vulnerabilities depends on both Client and IdP implementations. Due to the infeasibility of performing detailed testing with each website in the wild, we present two real-life attacks as case studies. We leave an exploration of the automated discovery of end-to-end attacks for future work.

Weaponizing Open Redirects. An open redirect is a common web application vulnerability that allows an attacker to influence the URL to which a victim is redirected when they visit a vulnerable site. Open redirect vulnerabilities that may be present on callback endpoints are formally acknowledged as a threat to OAuth 2.0 in the specification. However, using our novel path confusion technique and the knowledge of IdPs that do not perform the Redeem Process validation properly, we are now equipped to weaponize *any* open redirect on a site to compromise OAuth 2.0.

Because open redirect vulnerabilities are so common, instead of doing our own testing, we searched the Open Bug Bounty program for sites from our dataset with known, but unresolved issues [23]. The issue we picked was reported in 2018, assessed as a very low risk, and presumably not fixed as a result. However, because the site integrates with NAVER as an IdP, the combination escalates this low-risk vulnerability to a complete OAuth 2.0 account takeover.

We crafted the proof-of-concept attack below that takes the link to the NAVER Authorization Server and appends a malicious `redirect URI` that contains our path confusion payload. We redact the site as this vulnerability remains exploitable as of this writing, but our methodology is trivial to repeat.

```
https://nid.naver.com/oauth2.0/authorize?
  client_id=<REDACTED>&
  response_type=code&
  redirect_uri=https%3A%2F%2F<REDACTED>%2F
    openapi%2Fsocial%2Flogin.php/
    %252e%252e/%252e%252e/%252e%252e/
    redirect.php%3Ftarget%3Dhttps%3a%2F%2F
    <attacker-domain>%2F&
  state=random-state
```

The attack then plays out as expected: (1) The attacker tricks the victim into clicking on this link via social engineering. (2) The victim lands on the legitimate NAVER login page and enters their credentials. (3) NAVER redirects the victim back to <REDACTED>, but to the page that contains the open redirect vulnerability due to our path confusion payload. (4) The open redirect forwards the request to an attacker-controlled domain, leaking the code. (5) With access to the code, the attacker starts a new OAuth 2.0 flow, intercepts it at the browser before sending the Authorization Response, and injects into it the victim's stolen code before forwarding it to <REDACTED>. (6) <REDACTED> performs the rest of the Redeem Process, and because NAVER does not implement correct validation of the `redirect URI`, the protocol is successfully executed, giving the attacker full control of the victim's resources.

We presented one specific case here; however, attackers can scrape bug bounty reports or perform their own testing to exploit open redirects at scale by following the same simple methodology.

Table 1: Summary of findings.

IdP	Path Confusion	OPP	Redeem Validation
Atlassian	Vulnerable	Not Vulnerable	Correct
Dropbox	Not Vulnerable	Not Vulnerable	Correct
Facebook	Vulnerable	Not Vulnerable	Correct
GitHub	Vulnerable	Vulnerable	Incorrect
Kakao	Not Vulnerable	Not Vulnerable	Correct
LINE	Not Vulnerable	Vulnerable	Correct
LinkedIn	Not Vulnerable	Vulnerable	Correct
Microsoft	Vulnerable	Vulnerable	Correct
NAVER	Vulnerable	Vulnerable	Incorrect
OK	Not Vulnerable	Vulnerable	Correct
ORCID	Not Vulnerable	Vulnerable	Correct
Slack	Not Vulnerable	Vulnerable	Correct
Twitter	Not Vulnerable	Not Vulnerable	Correct
VK	Vulnerable	Vulnerable	Correct
Yahoo	Not Vulnerable	Vulnerable	Correct
Yandex	Not Vulnerable	Not Vulnerable	Correct

Abusing Real-Time Bidding. As we previously pointed out, RFC 6749 Section 3.1.2.5 states that Clients should never include third-party scripts in OAuth 2.0 endpoints to prevent `code` leaks. As part of an exploratory study, we measured the prevalence of this unsafe practice. Specifically, we inspected the network flows recorded in our previous experiments (see Section 4), identifying such a leak to third-party domains in 46 measurements out of 464 (10%), involving 11 IdPs out of 16 (68%), and 30 sites (8%). We identified **76 distinct domains** as leak destinations, the largest category being Ad networks with 30% of these domains.

Our investigation showed that this complex Ad network infrastructure can be abused as a viable OAuth 2.0 `code` leakage vector, specifically by targeting the *Real-Time Bidding (RTB)* mechanism. RTB allows advertisers to bid in real-time for Ad placement by providing them with information about the audience visiting a page. Our data showed that this information includes the referral headers of visitors. Therefore, when the callback endpoint contains such an Ad service, advertisers receive bid requests that contain OAuth 2.0 parameters. Anybody can sign up as an advertiser and access `code` parameters in real-time.

This attack vector is not critical for the reasons we have stated earlier; the `code` is a one-time token that expires after use, and the legitimate OAuth 2.0 flow would redeem it before a malicious bidder can act. However, if an attacker utilizes OPP to inject an invalid code and break the legitimate OAuth 2.0 flow, the victim's code that is leaked will be available for use without a race condition. When combined with an IdP that does not correctly perform the Redeem Process `redirect URI` validation, the situation escalates to a complete account takeover. We verified that this attack is practicable with real-life websites.

This RTB attack can also be combined with *path confusion* when ad services are not present on the callback endpoint but elsewhere on the site.

7 DISCUSSION AND CONCLUSION

Summary. In this paper, we have presented our observations on the OAuth 2.0 `redirect URI` validation requirements and security recommendations by referencing specific guidance from the protocol specification. We investigated the potential gaps in them in light of the contemporary systems-centric web application attacks.

Our experiments prove that the current "best practice" is not good enough, leaving IdPs, Clients, and Internet users exposed to attacks. In particular, we have shown that path confusion and parameter pollution attacks are viable with popular IdPs, affirmatively answering our research questions (Q1) and (Q2). We summarize the full list of IdPs we experimented with and our findings in Table 1.

The vulnerabilities we discovered are not mere implementation bugs, but they are rooted in the OAuth 2.0 specification where language is not prescriptive enough, or otherwise where the requirements miss threats like path confusion that have only recently started to gain traction in security literature. As a result, IdPs that systematically follow the relevant RFCs still run the risk of exposing `redirect URI` validation vulnerabilities.

It is important to stress that not all of these vulnerabilities translate to exploitable scenarios. OAuth 2.0 is a reasonably mature protocol that has received much security attention, resulting in adequate mitigating controls. Elsewhere, IdPs and Clients fill in the gaps and may address the protocol's weaknesses via their custom design decisions. Nevertheless, we have shown that end-to-end exploits affect real-life applications and have severe consequences, addressing our research question (Q3).

Recommendations. The steady stream of systems-centric web attacks like HTTP request smuggling and cache poisoning demonstrate that, strictly prescribed input validation instructions are paramount for consistent behavior in protocols that involve complex interactions. Thankfully, improving the OAuth 2.0 validation requirements is not an intractable effort. Devising a standard, narrowly defined string comparison strategy, and better input validation on sensitive parameters would immediately block the techniques we have presented, with minor implementation barriers.

Consequently, we conclude our paper with simple yet effective recommendations, addressing our final research question (Q4). All recommendations apply during both the Authorization Process and Redeem Process validation, and in fact must be implemented consistently in both checks to avoid further hazardous processing discrepancies.

`redirect URI` *validation must be performed via a strict string equality check, and this requirement must be clearly stated in formal specifications. That is, the compared URIs must be of equal size, and must be made up of an identical byte sequence.* This ensures that validation checks cover all components of the URI.

OAuth 2.0 parameters (e.g, `code`, `state`*) must be reserved names. Servers must check* `redirect URI` *for these reserved names and fail the validation if they are present.* Observing these parameters in `redirect URI` is either an attack indication, or a Client namespacing issue which could lead to hazardous interactions. Performing the check on the server shifts Client-side implementation responsibilities to the IdP, allowing consistent security guarantees.

Servers must ***NOT*** *perform input sanitization on* `redirect URI`. Any URI transformation or encoding/decoding operation on untrusted input could be weaponized by an attacker to elicit parsing discrepancies between a Client and the IdP, bypassing validations. Examples include the path confusion payloads we presented here and the security issues already documented in the specification, such as the abuse of URI fragments. `redirect URI` must always be *validated*, never sanitized.

One implementation hurdle we foresee with IdPs enforcing these recommendations is maintaining compatibility with the vast number of existing Clients with unusual or buggy protocol implementations. For instance, a Client may be reordering the `redirect URI` query string parameters between the Authorization Process and Redeem Process, or they may be fronting OAuth 2.0 endpoints with proxies that perform request transformations. This is a valid concern; however, it is also one that IdPs must address via opt-in non-secure configuration options that allow permissive validation checks for Clients that desire it. The OAuth 2.0 specification must provide prescriptive and correct guidance.

Ethical Considerations. All experiments described in this work were designed and conducted ethically, posing no risk to the tested Client sites, IdPs, or their users.

The data we used to seed the experiments and collected through our experiments was obtained using publicly available sources.

Following the common Internet measurement practice, our crawlers were limited to send below 15 requests per minute. We expect this added traffic load to be well below the threshold for performance degradation, an availability issue, or any other security anomaly that could get flagged by the tested Clients or IdPs, causing them undue effort to investigate.

We designed our testing methodology and proof-of-concept attacks to have no negative effects on the Clients, IdPs, or their users, persistent or otherwise. We used our own Client application and IdP accounts in all tests, demonstrating the attacks on our resources. We did not otherwise disrupt the everyday activities of the involved parties. Since we could not influence the OAuth 2.0 flows of Internet users, there was no possibility of inadvertent damage.

We notified all IdPs of our findings promptly. We notified the IdPs that were found to be impacted by improper validation throughout our experiments as we discovered vulnerabilities. When applicable to their circumstances, we provided them with detailed reports of our findings and proof-of-concept attack videos. We notified the remaining, non-vulnerable IdPs at the conclusion of our research by sending them a copy of this paper. All in all, we notified all 16 IdPs we tested, allowing them more than 90 days to mitigate their vulnerabilities. At the time of this writing, only Microsoft has confirmed that they mitigated the issue. The remaining IdPs acknowledged receipt of the notification but did not share mitigation plans or report progress.

We coordinated our findings with the OAuth Working Group (OWG) from the early stages of this work. This has resulted in an update to the OAuth 2.0 Security Best Current Practice, Section 4.1.3, clarifying the requirement for an exact string match during `redirect URI` validation [16].

8 ACKS

We thank Daniel Fett, Rifaat Shekh-Yusef and Hannes Tschofenig from the OAuth Working Group for their guidance and coordination with us throughout this work.

We also thank Avinash Sudhodanan for his helpful insights.

This work was partially supported by the EU Horizon project DUCA (HORIZON-MSCA-2021-SE-01 programme under GA 101086308) and by NSF grants 2329540, 2219921, and 2127200.

REFERENCES

[1] Sajjad Arshad, Seyed Ali Mirheidari, Tobias Lauinger, Bruno Crispo, Engin Kirda, and William Robertson. 2018. Large-Scale Analysis of Style Injection by Relative Path Overwrite. In *International World Wide Web Conference*.

[2] Marco Balduzzi, Carmen Torrano Gimenez, Davide Balzarotti, and Engin Kirda. 2011. Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In *Network and Distributed System Security Symposium*.

[3] Adam Bannister. 2023. OAuth 'masterclass' crowned top web hacking technique of 2022. PortSwigger–The Daily Swig. https://portswigger.net/daily-swig/oauth-masterclass-crowned-top-web-hacking-technique-of-2022.

[4] Michele Benolli, Seyed Ali Mirheidari, Elham Arshad, and Bruno Crispo. 2021. The Full Gamut of an Attack: An Empirical Analysis of OAuth CSRF in the Wild. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.

[5] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. 2018. WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring. In *USENIX Security Symposium*.

[6] Suresh Chari, Charanjit Jutla, and Arnab Roy. 2011. Universally Composable Security Analysis of OAuth v2.0. *Cryptology ePrint Archive* (2011).

[7] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2016. A Comprehensive Formal Security Analysis of OAuth 2.0. In *ACM Conference on Computer and Communications Security*.

[8] Mohammad Ghasemisharif, Chris Kanich, and Jason Polakis. 2022. Towards Automated Auditing for Account and Session Management Flaws in Single Sign-On Deployments. In *IEEE Symposium on Security and Privacy*.

[9] Mohammad Ghasemisharif, Amrutha Ramesh, Stephen Checkoway, Chris Kanich, and Jason Polakis. 2018. O Single Sign-Off, Where Art Thou? An Empirical Analysis of Single Sign-On Account Hijacking and Session Management on the Web. In *USENIX Security Symposium*.

[10] GitHub Docs. 2023. Authorizing OAuth Apps. https://docs.github.com/en/apps/oauth-apps/building-oauth-apps/authorizing-oauth-apps#web-application-flow.

[11] Dick Hardt. 2005. RFC 3986–Uniform Resource Identifier (URI): Generic Syntax. https://datatracker.ietf.org/doc/rfc3986/.

[12] Dick Hardt. 2012. RFC 6749–The OAuth 2.0 Authorization Framework. https://datatracker.ietf.org/doc/rfc6749/.

[13] Lauritz Holtmann. 2021. Insufficient Redirect URI validation: The risk of allowing to dynamically add arbitrary query parameters and fragments to the redirect_uri. (Web-)Insecurity Blog. https://security.lauritz-holtmann.de/post/sso-security-redirect-uri-ii/.

[14] David Krispin and Nir Swartz. 2021. Microsoft and GitHub OAuth Implementation Vulnerabilities Lead to Redirection Attacks. https://www.proofpoint.com/us/blog/cloud-security/microsoft-and-github-oauth-implementation-vulnerabilities-lead-redirection.

[15] Wanpeng Li, Chris J. Mitchell, and Thomas Chen. 2019. OAuthGuard: Protecting User Security and Privacy with OAuth 2.0 and OpenID Connect. In *ACM Workshop on Security Standardisation Research*.

[16] T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett. 2023. OAuth 2.0 Security Best Current Practice. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics.

[17] T. Lodderstedt, M. McGloin, and P. Hunt. 2013. RFC 6819–OAuth 2.0 Threat Model and Security Considerations. https://datatracker.ietf.org/doc/rfc6819/.

[18] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. 2020. Cached and Confused: Web Cache Deception in the Wild. In *USENIX Security Symposium*.

[19] Seyed Ali Mirheidari, Matteo Golinelli, Kaan Onarlioglu, Engin Kirda, and Bruno Crispo. 2022. Web Cache Deception Escalates!. In *USENIX Security Symposium*.

[20] Srivathsan G. Morkonda, Sonia Chiasson, and Paul C. van Oorschot. 2021. Empirical Analysis and Privacy Implications in OAuth-Based Single Sign-On Systems. In *Workshop on Privacy in the Electronic Society*.

[21] NAVER Developers. 2023. API Specification. https://developers.naver.com/docs/login/api/api.md.

[22] OAuth 2.0. 2014. OAuth Security Advisory: 2014.1 "Covert Redirect". https://oauth.net/advisories/2014-1-covert-redirect/.

[23] Open Bug Bounty. [n. d.]. Free Bug Bounty Program and Coordinated Vulnerability Disclosure. https://www.openbugbounty.org.

[24] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M. Pai, and Sanjay Singh. 2011. Formal Verification of OAuth 2.0 Using Alloy Framework. In *International Conference on Communication Systems and Network Technologies*.

[25] Pieter Philippaerts, Davy Preuveneers, and Wouter Joosen. 2022. OAuch: Exploring Security Compliance in the OAuth 2.0 Ecosystem. In *International Symposium on Research in Attacks, Intrusions and Defenses*.

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

[27] Frans Rosén. 2022. Account hijacking using "dirty dancing" in sign-in OAuth-flows. https://labs.detectify.com/2022/07/06/account-hijacking-using-dirty-dancing-in-sign-in-oauth-flows/.

[28] Youssef Sammouda. 2021. More secure Facebook Canvas: Tale of $126k worth of bugs that lead to Facebook Account Takeovers. https://ysamm.com/?p=708.
[29] Ethan Shernan, Henry Carter, Dave Tian, Patrick Traynor, and Kevin Butler. 2015. More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.*
[30] Avinash Sudhodanan and Andrew Paverd. 2022. Pre-hijacked accounts: An Empirical Study of Security Failures in User Account Creation on the Web. In *USENIX Security Symposium.*
[31] San-Tsai Sun and Konstantin Beznosov. 2012. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *ACM Conference on Computer and Communications Security.*
[32] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. 2013. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *USENIX Security Symposium.*
[33] Xianbo Wang, Wing Cheong Lau, Shangcheng Shi, and Ronghai Yang. 2019. Make Redirection Evil Again - URL Parser Issues in OAuth. Black Hat Asia. https://www.blackhat.com/asia-19/briefings/schedule/#make-redirection-evil-again---url-parser-issues-in-oauth-13704.
[34] Yuchen Zhou and David Evans. 2014. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *USENIX Security Symposium.*