OAuch: Exploring Security Compliance in the OAuth 2.0 Ecosystem

Pieter Philippaerts
imec-DistriNet, KU Leuven
Belgium
Pieter.Philippaerts@kuleuven.be

Davy Preuveneers
imec-DistriNet, KU Leuven
Belgium
Davy.Preuveneers@kuleuven.be

Wouter Joosen
imec-DistriNet, KU Leuven
Belgium
Wouter.Joosen@kuleuven.be

ABSTRACT

The OAuth 2.0 protocol is a popular and widely adopted authorization protocol. It has been proven secure in a comprehensive formal security analysis, yet new vulnerabilities continue to appear in popular OAuth implementations.

This paper sets out to improve the security of the OAuth landscape by measuring how well individual identity providers (IdPs) implement the security specifications defined in the OAuth standard, and by providing detailed and targeted feedback to the operators to improve the compliance of their service. We present a tool, called OAuch, that tests and analyzes IdPs according to the guidelines of the OAuth standards and security best practices.

We evaluate 100 publicly deployed OAuth IdPs using OAuch and aggregate the results to create a unique overview of the current state of practice in the OAuth ecosystem. We determine that, on average, an OAuth IdP does not implement 34% of the security specifications present in the OAuth standards, including 20% of the required specifications.

We then validate the IdPs against the OAuth threat model. The analysis shows that 97 IdPs leave one or more threats completely unmitigated (with an average of 4 unmitigated threats per IdP). No IdPs fully mitigate all threats.

We further validate the results by picking four attack vectors and using the tool's output to determine which IdPs to attack. The results were highly accurate, with a false positive rate of 1.45% and a false negative rate of 1.48% for the four attack vectors combined.

CCS CONCEPTS

• **Security and privacy → Authorization**; **Web protocol security**; **Security protocols**; *Access control*; *Web application security*.

ACM Reference Format:
Pieter Philippaerts, Davy Preuveneers, and Wouter Joosen. 2022. OAuch: Exploring Security Compliance in the OAuth 2.0 Ecosystem. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3545948.3545955

RAID 2022, October 26–28, 2022, Limassol, Cyprus

ACM ISBN 978-1-4503-9704-9/22/10…$15.00
https://doi.org/10.1145/3545948.3545955

1 INTRODUCTION

The OAuth 2.0 protocol [48] is a popular and widely adopted authorization protocol that enables a third party, the *relying party* (RP), to obtain limited access to an authorization/authentication service, hereinafter referred to as the *identity provider* (IdP)[1], on behalf of a user. Compared to its predecessor[2], OAuth 2.0 reduces the implementation complexity, resulting in an easier-to-understand protocol. However, like any security protocol, it must be carefully implemented. Security specifications are scattered all over the main OAuth standard, and additional specifications are described in documents such as the *OAuth Threat Model* [54] and the *OAuth Security Best Current Practices* [44].

The security of OAuth has been thoroughly analyzed [2, 5, 31], culminating in the work of Fett et al. [12, 13], which presents a comprehensive formal analysis of the security of OAuth 2.0 and the related OpenID Connect framework [56]. Yet, a stream of implementation vulnerabilities has been discovered over the years, including high-profile attacks [18, 58, 60, 61]. Despite the strong formal basis of the OAuth protocol, the security of many implementations continues to be inadequate.

We set out to enhance the security of the OAuth landscape by measuring how well OAuth IdP deployments adhere to the standards documents, and by providing detailed and targeted feedback to the operators to improve the compliance of their services. We test IdPs according to the guidelines of the security best practices and the OAuth threat model. The results are then aggregated to create a unique overview of the current state of the OAuth ecosystem.

Contributions In this paper, we introduce a tool that can be used to assess to what degree an OAuth IdP implements the security specifications from the OAuth standards. We present an analysis of the current OAuth ecosystem and we zoom in on specific countermeasures and threats. We confirm the results of our analysis by choosing four attack vectors to mount an attack on several OAuth IdPs that have been identified by the tool as vulnerable.

The assessment tool 'OAuch'. We introduce a tool called OAuch that analyzes OAuth 2.0 IdPs, including implementations that support the OpenID Connect (OIDC) extension. OAuch features 113 test cases that test the security specifications as defined by the various OAuth standards documents. The OAuch user is notified about missing countermeasures, deprecated features, and unmitigated or partially mitigated threats.

[1] The OAuth standard uses the terms *client*, *authorization server*, and *resource owner* to refer to the *RP*, *IdP*, and *user* respectively. In this paper, we adopt the more common terminology [4, 12, 24, 25, 27–29, 36, 37, 39, 42].
[2] This paper does not consider the older and vastly different OAuth 1.0 protocol. Whenever the term *OAuth* is used throughout this paper, version 2.0 of the protocol is assumed.

A large-scale analysis of the OAuth IdP ecosystem. We present the results of a large-scale security analysis of 100 publicly deployed OAuth IdPs to create an overview of the current OAuth ecosystem. The results show that, on average, an IdP does not implement 20% of the required security specifications and 33% of all security specifications. To the best of our knowledge, our work is the first large-scale ecosystem analysis that takes into account the full set of security requirements as defined in the OAuth standards.

Detailed analysis of specific threats and security requirements. We expand the OAuth threat model and propose threats for widely deployed OAuth extensions that are not covered by the threat model. Based on the ecosystem analysis, we discuss individual threats, countermeasures, and deprecated features. We show that many crucial countermeasures are often not implemented.

Demonstration of relevance. We validate the results by launching four attacks on IdPs that OAuch deemed vulnerable. The results were highly accurate, with a false positive rate of 1.45% and a false negative rate of 1.48% for the four attack vectors combined.

Interpretation of the results. Based on the interactions we had with the operators, we present several arguments that can explain why OAuth IdPs are missing so many countermeasures.

2 BACKGROUND

The OAuth 2.0 protocol [48] is a popular authorization framework that solves many of the problems found in the traditional client-server authentication/authorization model. It separates the role of the *RP* from that of the *user*. The user is the entity that can grant access to a protected resource, and the RP is an application requesting access to the protected resource on behalf of the user. The *IdP* issues access tokens when the user successfully authenticates and authorizes the RP to access the resource. These access tokens are essentially alternative and revocable credentials that are linked to a user and a specific *scope*. A scope defines granular permissions for an RP, for example to access data or perform actions. Access tokens can be used on the *resource server* to access the protected resource.

OAuth is an authorization *framework* that originally defined four modes of operation, called *grants* or *flows*. Additional flows that cater to other use cases have been proposed [47, 53], but are not widely implemented. Section A in the appendix illustrates the protocol with a detailed description of the authorization code flow.

RPs must be registered with the IdP before they can receive authorization. During enrollment, the RP receives a *client identifier* that must be present in the redirection URI of the authorization requests and during the authorization code exchange. In most cases, an RP must also register one or more *callback URIs* for valid redirection. If the IdP issues a secret or uses some other mechanism to authenticate an RP, the RP is said to be *confidential*. If the RP cannot securely store a secret (e.g., a mobile app), the RP is not issued authentication credentials and is called a *public RP*.

IdPs may grant *refresh tokens* together with access tokens. Refresh tokens are special tokens that are typically only used with confidential RPs and can be exchanged for a new access token and refresh token. This allows an access token — which is frequently used and more prone to leakage — to be short-lived. When the access token expires, the RP can simply use the refresh token to request a new access token, without having to involve the user.

OpenID Connect (OIDC) is an authentication protocol built on top of OAuth 2.0 for single sign-on and federated identity functionalities. OIDC extends OAuth with the concept of an *identity token* that contains information about the authenticated user.

3 RELATED WORK

Other research has looked into the security of the OAuth protocol and the implementation quality of RP and IdP deployments. The work that resembles our work the most is the OpenID Certification Tests website [57]. This test suite is technologically similar to the OAuch test suite but focuses on conformance tests for OpenID implementations. The tests include the security- and non-security-related requirements of the specifications. It also includes tests for some recommended mitigations (identifying missing mitigations with warnings) and does not include tests for optional mitigations. OAuch focuses on security, not conformance, and includes countermeasures of any requirement level. It also supports the OAuth best current practices, which is the most current source of OAuth security recommendations. The OAuth best current practices are currently not supported by the OpenID Certification Tests because they are published as an Internet-Draft and OIDC implementations technically do not have to comply with them.

Hedberg [17] implemented a deployment verification tool that specifically targets the OpenID Connect specification. The goal was not eliciting security vulnerabilities, but rather promoting interoperability across implementations of the standard.

Koponen [21] developed a secure implementation model of OAuth 2.0 and tested the implementation against the web application security weaknesses of the OWASP Top 10 [38] and other threats found in the literature. Similar to our work, the test suite analyzes OAuth 2.0 implementations following a black-box approach, testing a self-developed RP and IdP for vulnerabilities. Our work targets only the IdP, but covers many more test cases, and analyzes a broad ecosystem of OAuth 2.0 API and OIDC providers.

More recently, Fett et al. [10] carried out a formal analysis of the OpenID Financial-Grade API (FAPI), an Open Banking security profile of OAuth 2.0 intended for high-risk scenarios to defend against very strong attackers. Their systematic formal analysis leverages the Web Infrastructure Model [11] to build a comprehensive model of FAPI and various OAuth security extensions. The authors used this model to define and prove security properties and found authentication, authorization, and session integrity vulnerabilities whenever the proof failed. They were able to address the vulnerabilities and formally verify the security of the revised OpenID FAPI. Contrary to the work of Fett et al. that formally analyzes the specifications and is used to establish the best practices, our work analyzes the compliance of actual implementations with the OAuth specifications and best practices.

Many results either limit the number of OAuth providers that are tested, or focus on a small number of attack vectors. Ferry et al. [9] analyzed the security of OAuth 2.0 implementations, manually testing 21 sites against the recommendations of the OAuth threat model, and finding vulnerabilities with 2 sites. Li et al. [25] assessed the OIDC standard, but specifically focused on testing the security of a particular implementation, in this case forensically examining HTTP traffic of 103 relying parties leveraging Google's

implementation for signing in their users. Their study revealed serious vulnerabilities allowing a malicious adversary to log in as a victim user. Further analysis identified the cause to be a combination of Google's design of the OIDC service and the RPs sacrificing security for ease of implementation.

Shernan et al. [36] investigated non-compliance with the OAuth 2.0 standard, and in particular vulnerabilities to Cross-Site Request Forgery (CSRF) attacks on real-world OAuth 2.0 deployments of the authorization code flow by analyzing to what extent clients and authorization servers implement the mandatory CSRF protection. They evaluated 13 IdPs, of which only 4 enforce CSRF protections. While crawling the Alexa Top 10,000 domains, they observed that 25% of websites using OAuth 2.0 appear vulnerable to CSRF attacks. These vulnerabilities were caused by mistakes ranging from weaknesses in sample code, lack of documentation, to inconsistent implementations of APIs across a large company.

Related research on CSRF attacks against both OAuth 2.0 and OpenID Connect was carried out by Li et al. [26]. They propose a new technique to mitigate CSRF attacks that can be easily implemented by the RP and that requires no changes at the IdP. Their approach verifies whether the referrer header points either to a valid IdP domain or the RP domain. Follow-up work by the same authors [29] presents OAuthGuard, an OAuth 2.0 and OpenID Connect vulnerability scanner and protector. OAuthGuard aims to protect user security and privacy even when the RPs do not implement the standard correctly. The authors investigated 5 vulnerabilities (e.g., use of HTTP vs HTTPS, leaking user tokens) on 1,000 top-ranked websites by majestic.com. From the 137 websites that used Google Sign-in, 69 suffered from at least one serious vulnerability. OAuthGuard could protect user security and privacy for 56 of them, while warning the user about the insecure implementation for the remaining ones.

Contrary to previous work that relies on a manual discovery of new vulnerabilities in OAuth 2.0, Yang et al. [42] proposed OAuthTester, an adaptive model-based testing framework for automated, large-scale security assessments of OAuth 2.0 implementations. Based on the OAuth specifications, the authors created a coarse-grained system model in the form of a state machine that abstracts the behavior of all parties. They analyzed 4 major IdPs and 500 Alexa top-ranked US and Chinese websites that use the OAuth-based SSO service provided by the IdPs. Not only did OAuthTester rediscover known vulnerabilities, but it was also able to discover new exploits due to, for example, OAuth-based applications not adopting TLS to protect their OAuth sessions. From a technology perspective, OAuch follows a different implementation approach. While previous work mainly tests RPs, OAuch acts as an RP to analyze the security non-compliance of IdPs. OAuch can only discover compliance issues that are documented in the OAuth security guidelines, and does not detect new vulnerabilities.

Calzavara et al. [4] presented WPSE, a browser-side security defense to prevent attacks against the OAuth 2.0 and SAML 2.0 protocols. Specifically for OAuth 2.0, the authors focused on protocol flow deviations, secrecy, and integrity violations. They tested WPSE on 90 websites, and found security flaws in 55 websites due to, for example, tracking libraries.

Other research has also looked into the security of client implementations. Chen et al. [7] performed a large-scale study of 600 popular mobile applications and concluded that almost 60% of applications were incorrectly implemented. They redid the survey two years later [6], only to discover that the situation had not improved. Wang et al. came to a similar conclusion by testing popular Chinese Android apps [40] and international apps [39]. Al Rahat [32] proposed OAuthLint, a tool that encodes vulnerable patterns extracted from the OAuth specifications to find bugs in Android apps. Follow-up work by the same authors [33] investigated the security of OAuth service provider implementations, proposing a framework called Cerberus that implements a query-driven algorithm to test security-critical OAuth properties. It investigates vulnerabilities in OAuth server libraries widely used by service providers. OAuch does not analyze specific software libraries, but detects non-compliance of deployed IdPs running any implementation of the OAuth 2.0 protocol.

Ghasemisharif et al. [15, 16] researched shortcomings and security implications with Single Sign-On. In [16], the authors propose a countermeasure against account hijacking that implements Single Sign-Off capabilities on top of OIDC. Their follow-up work on account hijacking [15] proposes SAAT as a fully automated framework that analyzes whether RPs using Facebook as the IdP comply with security best practices and guidelines. Compared to SAAT, OAuch features less automation, but is not constrained to Facebook's IdP and focuses on IdP compliance instead of RP compliance.

Li et al. [30] performed a computationally sound security analysis of OAuth 2.0 in a three-party setting that covers all kinds of authorization flows. Complementary to OAuch , it aims to formally verify the OAuth 2.0 protocol itself rather than its implementation or deployment. Additionally, the authors validate the soundness of their model by identifying known attacks against OAuth 2.0.

Benolli et al. [3] empirically analyzed the prevalence of OAuth CSRF vulnerabilities in the wild in a large-scale study of 314 high-ranked sites that implemented the Facebook Login flow. OAuch analyzes similar URI redirection concerns as well as many more security threats and countermeasures, and explores these concerns beyond Facebook as the IdP.

Compared to some previous works, OAuch does not focus on a limited set of threats but tests more thoroughly in both breadth and depth as explained in Section 4. Related work [8, 43] has shown that executing an OAuth flow can be successfully automated with the help of browser automation tools and for a well-defined set of IdPs. OAuch, on the other hand, is designed to work on any browser without additional plug-ins and for any IdP. As a result, OAuch is not fully automated.

Table 5 in the appendix summarizes the contributions of the related work, categorized per threat. Though the related work investigates an extensive set of threats on the OAuth framework and its implementations, many threats are not covered. Works that are referenced in the table may also investigate other aspects of a threat, compared to our work. For example, they may assess RP mitigations, whereas OAuch examines IdP mitigations.

4 THE OAUCH TESTING FRAMEWORK

OAuch[3] is an open-source *security best practices and threats analyzer* for OAuth 2.0 IdP implementations. Its main goal is to analyze

[3] https://oauch.io/

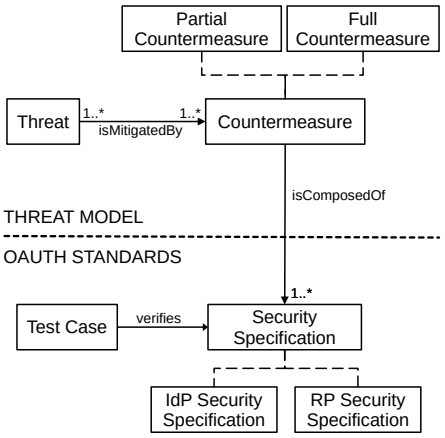

Figure 1: Mapping threats, countermeasures and security specifications.

an IdP's compliance with the OAuth standards to uncover unmitigated threats and point out security improvements. OAuch tests an IdP using a large set of test cases to check an IdP's compliance with the security specifications defined in the original OAuth 2.0 standard [48, 49], as well as other documents that refine the security assumptions and requirements. These documents include the OAuth threat model [54], the Security Best Current Practices [44], and others [45, 47, 50–52, 55]. In addition to OAuth, OAuch also supports OpenID Connect [56] providers.

4.1 Threat Model

The OAuth working group has published a comprehensive threat model [54] shortly after publishing the original OAuth 2.0 standard. This threat model is further refined in the latest *Security Best Current Practices* document [44] to include additional threats that have been observed in real-world usage of OAuth.

As depicted in Figure 1, the threat model describes for each threat how an implementation may be attacked and which countermeasures can be applied. Some threats are mitigated by a combination of multiple countermeasures, while others can be mitigated by a single countermeasure. In many cases, alternative sets of countermeasures may be used to address a threat. Some countermeasures may (partially) mitigate multiple threats.

The model assumes a powerful attacker that has full access to the network between the RP and the IdP, and the RP and the resource server. The attacker may eavesdrop on any communication between those parties and has unlimited resources to mount attacks. In addition, two of the three parties involved in the OAuth protocol may collude to mount an attack against the 3rd party.

This threat model has been adopted in OAuch and is used to offer precise feedback to the user. OAuch uses test cases to detect which countermeasures are implemented by the IdP. It then uses the information from the threat model to determine which threats are mitigated. For every threat, it takes the list of mitigations that are proposed by the threat model and compares it with the mitigations that have been detected. If the threat is properly mitigated, it is marked as *fully mitigated*. When no relevant countermeasures are active, the threat is *unmitigated*. Threats can also be *partially mitigated* if some countermeasures are present, but not all. When multiple sets of countermeasures can mitigate a threat, it is sufficient that only one set is fully implemented.

The OAuth threat model lists 49 threats, and the OAuth best current practices document adds 6 threats (as well as updating other threats in the original threat model). OAuch tests for 36 of these threats. Of the remaining threats, 8 are RP or user threats and are out of scope, 8 are not protocol-related (e.g., *manipulation of scripts on the authorization page*) and cannot be tested, and 3 are related to denial-of-service attacks. We extend the threat model with 6 additional threats to incorporate countermeasures from popular OAuth extensions that are not referenced in the official threat model. Table 5 lists the threats of the extended threat model as used by OAuch.

4.2 Supported OAuth Standards

OAuth 2.0 is a *framework* with multiple extensions that have been standardized over the years to support new use cases or to improve the security. The test cases in OAuch are based on the security specifications as written down in these OAuth-related standards documents. The following documents are supported:

- *The OAuth 2.0 Authorization Framework* (RFC6749)
- *The OAuth 2.0 Authorization Framework: Bearer Token Usage* (RFC6750)
- *OAuth 2.0 Token Revocation* (RFC7009)
- *JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants* (RFC7523)
- *Proof Key for Code Exchange by OAuth Public Clients* (RFC7636)
- *OAuth 2.0 Device Authorization Grant* (RFC8628)
- *OAuth 2.0 Security Best Current Practice*
- *OpenID Connect Core 1.0 incorporating errata set 1*
- *OAuth 2.0 Form Post Response Mode*

4.3 Mapping Threats, Countermeasures and Security Specifications

Converting the security specifications in the OAuth standards, which are written in natural language, into actionable security test cases is a manual process. Though this procedure is time-consuming and requires technical expertise, it must only be done once per standard document. The conversion uses the following approach:

(1) **Identifying Specifications.** The security specifications in the standard are enumerated. They can be identified by the accompanying keyword *must*, *should* or *may*. This keyword indicates the requirement level. *Must* indicates an absolute requirement to implement the specification, *should* implies a strong preference to implement it, and *may* merely offers it as a suggestion. The requirement level is always fully capitalized which makes it easily recognizable. Figure 2 shows a snippet of the core OAuth specification that specifies two separate countermeasures related to caching.
(2) **Filtering IdP Specifications.** Not all security specifications are relevant for OAuch. Only specifications that govern the behavior of the IdP are selected. Specifications that focus on

the RP or other actors in the OAuth protocol are out of scope and are excluded.

(3) **Selecting Specifications.** Some specifications are not directly related to the OAuth protocol and cannot be tested. An example is the suggestion that IdPs encrypt stored credentials. Other specifications may be too disruptive to test (e.g., testing the presence of denial of service protection). These specifications are also out of scope and excluded.

(4) **Implementing Test Cases.** A test case is implemented for each of the selected security specifications. Each test case performs a minimal check to validate the IdP's implementation of the specification. A test case can thus be seen as a yes-no question that provides information about the compliance of the implementation to a single security specification. This implies a one-to-one correspondence between security specifications and test cases.

Because a security specification is a very precise and well-defined technical requirement, implementing a test case is not complicated. For example, the OAuth standard requires that *confidential RPs or other RPs issued client credentials MUST authenticate with the IdP*. This specification is converted into a test case where OAuch tries to retrieve an access token from the IdP without using client authentication. If the request succeeds and an access token is granted (despite not using client authentication), OAuch has determined that the IdP does not implement this specific requirement. On average, a test case consists of only 11 executable lines of code.

After a careful manual analysis of the documents listed in Section 4.2, we extracted and implemented a total of 113 unique test cases. One security specification may be mentioned in multiple documents with varying requirement levels. For example, the original OAuth standard states that the redirect URI *should* be exactly matched to a preconfigured value, whereas the *Security Best Current Practice* document changes this to a *must*. In this case, OAuch assigns the most strict requirement level to the test case.

Not all security specifications in the documents are converted into test cases. This raises the question what percentage of specifications *are* represented by the OAuch test cases. We analyzed the main OAuth standard (RFC6749) to try and determine the coverage percentage of the mandatory security specifications. Roughly 60 unique security requirements were identified. OAuch implements a test case for 33 of these requirements. Another 15 requirements are aimed at RP implementations and are out of scope. Of the remaining requirements, 11 are not directly related to the protocol and could not be tested remotely, and one was related to DDoS protection and was considered too disruptive to test.

Finally, the test cases are mapped to specific threats in the OAuth threat model:

(1) **Selecting Threats.** Threats that are only applicable to RPs are not in scope and are discarded. Two threats that are related to DDoS attacks are too disruptive to test and are discarded as well.

(2) **Filtering Countermeasures.** Each threat has a list of countermeasures to mitigate the threat, as illustrated in Figure 3. Countermeasures that are not relevant to IdPs are removed.

(3) **Mapping Countermeasures to Test Cases.** The countermeasures in the threat model correspond directly to the security requirements in the OAuth specifications. For example, the cache countermeasure mentioned in Figure 3 corresponds to the two security requirements in Figure 2. Each security requirement has a matching test case. For each countermeasure, the corresponding test cases are associated with the threat. Section C in the appendix lists for every threat the corresponding OAuch test cases.

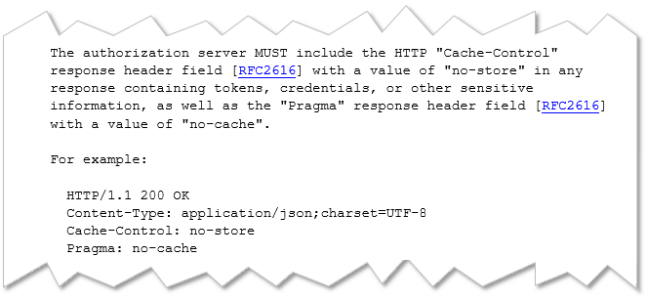

Figure 2: A snippet of the OAuth 2.0 core specification (RFC6749) that illustrates how security requirements are defined.

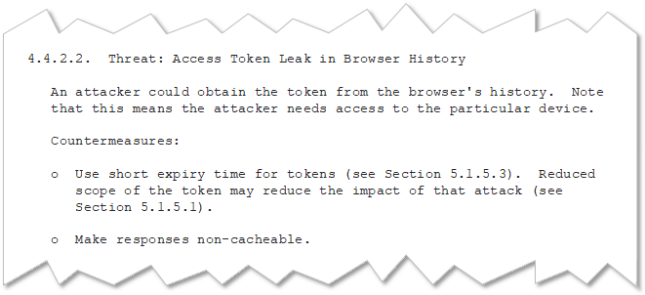

Figure 3: A snippet of the OAuth 2.0 Threat Model (RFC6819) that illustrates how a threat is documented.

The threat model lists for every threat *all* applicable countermeasures, but does not refer to the requirement levels that are mentioned in the security specifications. Hence, it does not show any preference for one countermeasure over another. OAuch adopts this approach and does not take the requirement level of a countermeasure into account when determining whether a threat is (partially) mitigated. However, the requirement levels of the failed test cases *are* reported to the user.

4.4 Running Test Cases with OAuch

OAuch tests an IdP with numerous authorization requests triggering a specific behavior that is either expected or not allowed.

OAuch works like an ordinary RP. When testing an IdP, the regular RP registration procedure must be followed. The IdP generates a *client id* and optionally a *client secret*. It tells the user which OAuth endpoint URIs to use for authorization and token requests, and generally asks the user to register a callback URI (depending on which flows are used). The user then creates a new test profile in OAuch and enters the settings they received from the IdP.

When the testing process is started, the test cases from the selected standards documents are run against the IdP. OAuch starts by detecting which flows and features are enabled on the IdP and executes the relevant test cases accordingly.

OAuch uses two browser windows during testing: one window drives the test process and shows the progress to the user, and a second window handles the callbacks from the IdP. Figure 4 shows the two browser windows during an active test run.

OAuch requires user interaction in two cases: when the user has to authenticate and authorize access, or when the IdP shows an error message to the user. During a test run, OAuch sends on average 39 authorization requests to the IdP. However, cookies are generally used to remember signed-in users and most IdPs automatically authorize a request if the user has granted access to the RP before. In practice, 84% of the authorization requests are granted automatically without any user interaction.

Because OAuch violates the OAuth protocol in many test cases, the IdP may show an error to the user instead of redirecting the browser back to the callback URI. Although this behavior is by design (and often a good sign), it stalls the test progress. OAuch continues to wait for a callback from the IdP but this callback will never happen. To solve this stalemate, the user clicks the *stalled test* button whenever the IdP shows an error. This signals OAuch that the callback will not happen, and forces it to move to the next step in the testing process. An average of 4 authorization requests per test run fail this way.

4.5 Reporting IdP Scores

OAuch calculates several statistics after each test run. The most important output is the number of unmitigated threats. These threats represent weak points in the implementation, which can be exploited under the right circumstances. The number of partially mitigated threats and deprecated features is the second most important output. Partially mitigated threats may or may not be exploitable; OAuch does not report to what degree these threats have been mitigated, only that there is at least one partial mitigation active. Deprecated features should be avoided if possible, as they are often deprecated on the grounds of being insecure.

In addition to these three important indicators, OAuch also computes the failure rates of the test cases. This metric is calculated by dividing the number of failed tests by the total number of tests that are executed, and converting the result to a percentage. This percentage indicates to what degree an IdP correctly implements the OAuth standard. An overall failure rate is reported, as well as the individual failure rates of the three requirement levels (*must*, *should*, *may*). The calculation only takes into account the tests that were executed and able to verify whether a security requirement holds. If a test fails for some unanticipated reason (e.g., a temporary network problem) or if a test is skipped because it is not relevant for the IdP, it is excluded from the calculation.

A test run executes more test cases if the IdP supports many flows or enables more features. This increases the number of failed test cases, but also increases the number of executed tests, keeping the failure rate relatively stable. To test this assumption of stability, we analyzed the failure rates of all IdPs that had two or more flows and recalculated the failure rate as if only the authorization code flow had been enabled. The overall failure rate increased by 2.05% ± 2.46%, confirming that the number of active flows has only a small impact on the calculated result.

4.6 Scope and Limitations

The OAuth standard defines security specifications for both RP and IdP implementations. It is clear that if either the RP or the IdP contains a vulnerability, the security of the entire system is broken.

OAuch focuses on the IdP specifications and only tests the behavior of an IdP implementation. Security issues on the IdP typically have a higher impact than issues in an RP. A vulnerability in an RP allows an attacker to abuse the data of the users using that RP. A vulnerability on the IdP may affect all users. Furthermore, fixing a vulnerability on the IdP may require all RPs to be updated as well (depending on the type of vulnerability). By focusing on the IdP, OAuch improves the security of RPs to a certain extent. If the IdP does not allow insecure behavior, the RP is forced to be more secure as well.

Some OAuth-related threats can be mitigated on the RP (e.g., a CSRF attack against the redirect URI). In the threat analysis, OAuch considers a worst-case scenario and assumes that an RP does not have these mitigations in place. This is not an unrealistic assumption: previous research has shown that many RPs have a flawed implementation and are unsafe [7, 24, 35, 37, 40, 42]. Threats that can only be mitigated on the RP and not on the IdP (e.g., phishing for end-user credentials using a compromised or embedded browser) are not considered.

OAuch only focuses on the security specifications in the OAuth documents. These documents propose mitigations for protocol-specific security issues, but they do not cover issues on other levels of abstraction. For example, implementations could be attacked through logical flaws on the application layer, like the mismanagement of the scope parameter that could lead to a privilege escalation vulnerability [21]. Another way to attack an IdP is through timing attacks, where an attacker takes advantage of race conditions in the implementation [63]. Programming bugs (e.g., buffer overflows) might be exploitable as well. Because these vulnerabilities are not specific to the OAuth protocol, they are not covered by the OAuth standard. As a result, OAuch does not test for these issues.

The test cases in OAuch each test a very specific part of the standard and the results are straightforward to interpret. Yet, due to the black box nature of the IdP's authentication and authorization process, 7 test cases use a heuristic to calculate their output. For example, when testing whether the IdP automatically grants an authorization request, OAuch must be able to differentiate between an automatic authorization and an authorization that the user manually approved. In this case, a heuristic is used to classify authorization responses either as automatically or manually granted. This introduces some inaccuracy in the results, as there is a small chance that the calculated result is wrong. A manual analysis of three of these test cases revealed that the error rate is small, with a false positive rate of 2.4% and a false negative rate of 2.0%. The 106 other test cases have no false positives or negatives.

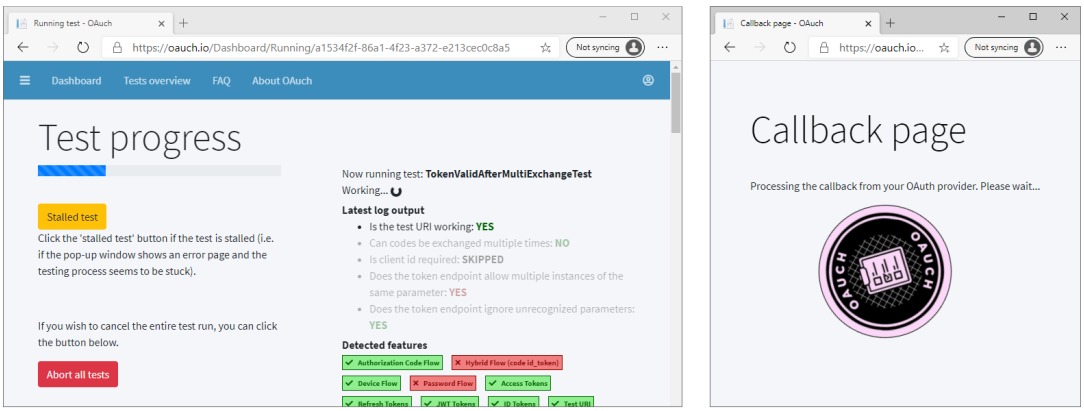

Figure 4: An active OAuch test run. The left browser window is driving the testing process and informs the user of the progress. The right window is used to handle callbacks from the IdP.

5 ANALYZING THE ECOSYSTEM

We use OAuch to test 100 publicly available OAuth IdPs that are deployed and in production, with the expectation that they have all relevant security precautions in place. This section analyzes the entirety of the ecosystem. Table 4 in the appendix contains the detailed results, on a per-IdP basis.

5.1 Approach

Our empirical study of a large selection of OAuth IdPs evaluates the OAuth implementation of each IdP with OAuch, and aggregates the results. Our analysis of these results and the current state of practice offers a unique overview of the OAuth threat landscape that identifies frequently missing mitigations.

The ProgrammableWeb[4] — an independent website that maintains an index of publicly available APIs — identifies 187 websites in the top 10,000 that host an OAuth 2.0 IdP, and over 300 additional websites in the top 1,000,000. Not every site is accessible to test. Providers may charge money for their service or require the purchase of specific hardware. Services may only be available to citizens of specific countries, or users with security clearance.

We selected a representative sample of 75 IdPs from the top 10,000 for the analysis. This sample closely follows the distribution of the full dataset, where we observed a strong presence of top sites (rank < 1000, and especially rank < 100). We further supplemented the list with another 25 sites in the top 1,000,000.

IdPs are classified into two categories: *API providers* and *OpenID Connect (OIDC) providers*. API providers are websites that allow RPs to consume data or programmatically interact with the site via an API. OAuch does not test the security of this API but only the compliance of the OAuth authorization process. OIDC providers are IdPs that support the OpenID Connect identity layer on top of an OAuth 2.0 service. One in five selected IdPs is an OIDC provider.

5.2 Authorization Grants

The original OAuth 2.0 standard defines four different authorization grants (or flows). The most popular flow is the authorization code flow, which is supported by 93% of the IdPs. Two grants — the implicit flow and the password flow — have been deprecated since 2019. Yet, the implicit flow is still supported by 40% of the IdPs. Only 4% of the tested IdPs support the password flow.

Table 1: An overview of the support in the ecosystem for various OAuth 2.0 authorization grants.

Authorization Grant	Overall	API Prov.	OIDC Prov.
Authorization Code Grant	94%	93%	100%
Client Credentials Grant	30%	30%	30%
Implicit Grant	37%	40%	25%
Password Grant	3%	4%	0%
Hybrid Grant (OIDC)	8%	0%	40%
Device Grant (RFC8628)	1%[5]	0%	5%[5]

OIDC providers are generally used as identity management services to enable social logins, but they may also be used for API authorization. The authorization code flow is supported by every IdP that was tested. The implicit flow can be used on 25% of the IdPs, and the hybrid flow — a flow specifically introduced for OIDC servers — was enabled on 40% of the IdPs. Only one IdP supported the device flow[5].

Table 1 gives the full overview of supported OAuth grants throughout the ecosystem.

5.3 OAuth Extensions

Most OAuth extensions are not widely supported in the ecosystem. Table 2 gives an overview of extensions that are supported by OAuch and the percentage of IdPs that support each extension.

Two results in Table 2 are noteworthy. The *PKCE* extension improves the security characteristics of the authorization code flow. It is an important mitigation against authorization code injection attacks, which explains why the security best practices mandate its use. Yet, only 21% of the tested IdPs support PKCE. Likewise, *Mutual TLS* is a recommended security extension to improve OAuth's client

[4]https://www.programmableweb.com/

[5]This number is an underestimate. In addition to Google, Microsoft also supports the device grant under certain conditions. Facebook has a flow that is similar to the device grant but is incompatible with RFC8628.

Table 2: An overview of the support in the ecosystem for various OAuth 2.0 extensions.

Normative Document	Overall	API Prov.	OIDC Prov.
Bearer Tokens (RFC6750)	92%	90%	100%
Token Revocation (RFC7009)	19%	11%	50%
JWT Assertions (RFC7523)	3%	0%	15%
PKCE (RFC7636)	21%	15%	45%
Server Metadata (RFC8414)	13%	3%	55%
Device Grant (RFC8628)	1%[5]	0%	5%[5]
Mutual TLS (RFC8705)	0%	0%	0%

authentication capabilities during the authorization process and when using an access token. None of the tested IdPs support it.

In addition to the documents listed in Table 2, the OAuth working group has created several other OAuth-related standards like token binding, dynamic client registration, and SAML assertions. However, these standards have a low adoption rate. None of the tested IdPs supported any of these extensions.

5.4 Current State of Practice

Table 4 in the appendix shows a great variety in terms of issues that have been found. The overall failure rate was 33%, meaning that on average about one in three (applicable) security specifications are ignored by implementers. If the failure rate is split up by requirement level, the failure rates are 20% for the required specifications, 56% for the recommended specifications, and 81% for the optional specifications. While the high failure rate for the optional specifications is somewhat expected, it is alarming that one in five requirements are not or incorrectly implemented.

The OIDC providers score better than the API providers with an average failure rate of 26%. This is an interesting result, because OIDC implementations are more involved than plain OAuth implementations, and are subjected to more test cases. An explanation may be that the OIDC consortium offers conformance tests (see Section 3) and several of the providers in this category have received this certification. Despite these conformance tests, all OIDC IdPs failed to implement several requirements from the standard, resulting in an average failure rate of 16% for the mandatory security specifications.

Figure 5 shows a histogram of the failure rates of the mandatory specifications. More than half of the test cases failed for less than 20% of the IdPs. This confirms our observation that IdPs have a relatively unique set of failed test cases. Only a few test cases fail for many IdPs, making it harder to give generally applicable advice for improvement.

Popular IdPs do not seem to perform better in our analysis. If the IdPs are split into categories according to their Tranco top sites ranking [23], the failure rates are similar over each category. For example, the categories *top 200*, *top 201–2,000*, *top 2,001–10,000*, and *top 10,001+* yield four similarly sized categories with respective failure rates of 32.0%, 33.0%, 36.7% and 32.9%.

OAuch can calculate statistics on a per-document basis. Table 3 shows the aggregation of these results over all IdPs. The reported percentages are the failure rates of the test cases that are relevant for a particular standard. Lower values indicate better compliance with the standard.

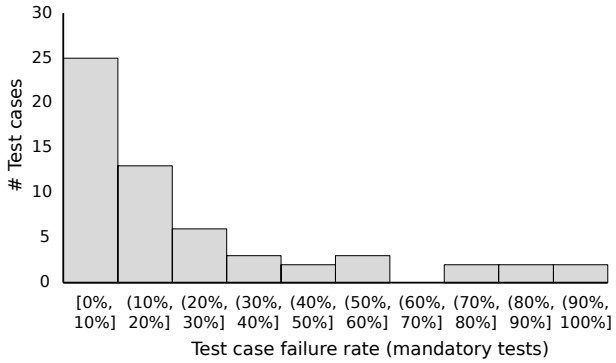

Figure 5: A histogram of the average test case failure rate. This only includes test cases for mandatory security specifications that have been run against at least 10 IdP implementations.

Table 3: The average security compliance scores over the entire ecosystem, listed per document. The percentages are the failure rates of the test cases (lower is better).

Normative Document	Overall	Must	Should	May
OAuth 2.0 (RFC6749)	25.0%	19.8%	40.7%	50.9%
Bearer Tokens (RFC6750)	7.9%	0.7%	60.6%	
Threat Model (RFC6819)	22.3%	2.8%	22.8%	63.9%
Token Revocation (RFC7009)	8.9%	5.8%	12.5%	25.0%
JWT Grant Type (RFC7523)	13.8%	4.2%		60.0%
PKCE (RFC7636)	19.5%	11.4%		100.0%
Device Grant (RFC8628)	10.0%	11.1%	0.0%	
Security Best Practices	61.1%	40.3%	66.0%	85.6%
OpenID Connect	12.6%	13.2%	5.3%	
All documents combined	33.0%	20.0%	56.3%	80.9%

There are large differences between the results, but one result stands out. No less than 40% of the security specifications in the *Security Best Current Practices (BCP)* document are missing in a typical IdP implementation. These results are alarming because the BCP is the most up-to-date source of OAuth security guidelines. A requirement present in the BCP but not in the original OAuth RFC is almost three times less likely to be implemented, compared to a requirement that is present in both. As a result, advanced injection, impersonation, and replay attacks might not be properly mitigated.

5.5 Threat Analysis

The most important indicator of an IdP's security posture is the number of unmitigated or partially mitigated threats. Figure 6 shows a histogram of the number of unmitigated threats, with most IdPs having between one and seven unmitigated threats. Assuming a worst-case scenario where no additional RP countermeasures are used, these IdPs are vulnerable under the attack assumptions of the OAuth threat model. Only three IdPs succeeded in having no unmitigated threats.

The results of the analysis can be used to rank the threats according to prevalence in the ecosystem. Table 5 in the appendix shows the detailed results for all threats. After filtering out threats

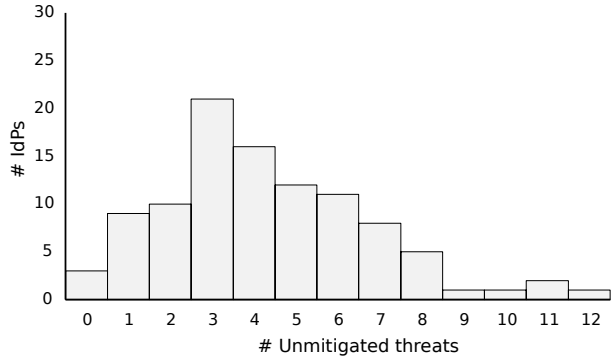

Figure 6: A histogram of the number of threats that are not mitigated. An unmitigated threat is a threat for which none of the countermeasures that are suggested in the OAuth threat model are implemented.

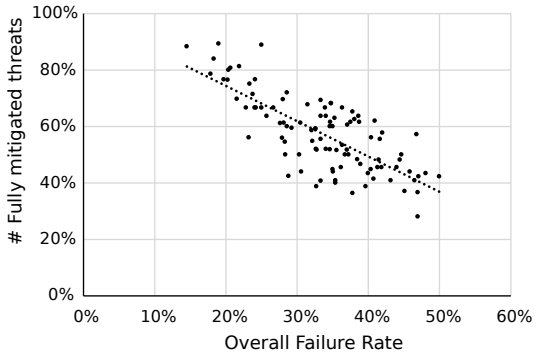

Figure 7: A plot showing the correlation of an IdP's overall failure rate with the percentage of fully mitigated (relevant) threats. The correlation coefficient is -0.75, which implies a fairly strong correlation between the parameters.

that are only relevant for a small number of IdPs, the following top five threats are obtained:

(1) *Authorization Code Injection* (78% failure rate) — This is directly linked to the low uptake of the PKCE security extension, as mentioned in Section 5.3, that mitigates this threat. Section 6.1 demonstrates that authorization code injection attacks are possible on all OIDC IdPs for which we found public RPs.
(2) *Obtaining Access Tokens* (73% failure rate) — To avoid access tokens from being leaked, they should be valid for only a short period. Only 27% of IdPs mint access tokens that are valid for an hour or less. While most access tokens are valid for less than a day, 10% of IdPs use tokens that are valid up to a week, and another 18% use tokens that are valid for more than a week.
(3) *Obtaining Client Secrets* (54% failure rate) — Most IdPs automatically grant authorization to an RP if the user has granted authorization before. This improves the usability of the IdP, but can also lead to an impersonation attack when a public RP (one that uses the implicit flow, for example) is used. Section 6.2 reports on the results of our proof-of-concept attack that worked on all the IdPs that OAuch believed to be vulnerable.
(4) *PKCE Downgrade Attack* (43% failure rate) — An attacker may trick the IdP into ignoring the PKCE parameters. This threat is different from the other threats because it originates from an implementation error instead of a missing countermeasure. Section 6.3 details our proof-of-concept downgrade attack that worked on all identified IdPs.
(5) *Token Leakage via Log Files and HTTP Referrers* (40% failure rate) — RPs should use the *Authorization* HTTP header to transmit the access token to API endpoints. If the resource server accepts an access token as a URL parameter, there is a risk that the access token leaks via log files or the HTTP referrer header[6].

[6]In November 2020, the W3C changed its recommendations on the default browser behavior with respect to referrers. Going forward, this change may help mitigate some attacks that fall under the *Token Leakage via Log Files and HTTP Referrers* threat.

Threats are linked to one or more test cases, but not all test cases are linked to a particular threat and some test cases are linked to multiple threats. This makes the relation non-linear and at times difficult to interpret. For instance, two IdPs with similar amounts of mitigated threats may have wildly different test case failure rates. Nevertheless, an overall correlation is observed. Figure 7 shows for the tested IdPs the relation between the overall test case failure rate and the percentage of relevant threats that are fully mitigated. As expected, the calculated correlation coefficient of -0.75 implies that a lower failure rate is correlated with the mitigation of more threats.

5.6 Deprecated Features

Many IdPs support deprecated features for backward compatibility. Although these features have been deprecated for security reasons, it is ultimately a business decision whether to continue to support them or not. OAuch informs the user about deprecated features but does not include them in the calculated failure rates.

In terms of which deprecated features are enabled most often, three stand out. Over half of the IdPs (51%) still support outdated versions of TLS. Access tokens can be passed via URI query parameters on 38% of the resource servers, and the deprecated implicit flow is enabled on 37% of the IdPs.

5.7 Case Studies

The ecosystem analysis has produced a lot of statistics on various implementation aspects of OAuth. This section contains three case studies that dig deeper into important parts of the OAuth infrastructure. The most important or surprising statistics are presented for each case study.

5.7.1 Client Authentication. The OAuth standard distinguishes between public and confidential RPs Out of all the IdPs that were tested, only one IdP (1%) used public RPs. All other IdPs used confidential RPs, with most IdPs (97%) opting for client passwords to authenticate an RP. Two IdPs (2%) allow RPs to upload a cryptographic key during the enrollment process. Of the IdPs that issued

a client secret, 11% did not require this secret to be presented when exchanging an authorization code, and 13% did not require the secret to exchange a refresh token.

5.7.2 Redirect URI Matching. The OAuth standard recommends allowlisting redirect URIs for confidential RPs and requires it for public RPs (or anyone using the implicit flow). The current advice is to always use exact string matching when comparing the value of the *redirect_uri* parameter with the allowlisted value, as other approaches turned out to be error-prone [12, 37, 41]. Nonetheless, only 53% of the tested IdPs follow this advice.

To prevent authorization code leakage through counterfeit websites, RPs must include the *redirect_uri* parameter when exchanging a code. The original OAuth standard document already included this requirement, yet 37% of the IdPs do not validate this parameter.

5.7.3 Authorization Codes and Refresh Tokens. Authorization codes are a prime target for attackers because they can be used to generate access tokens as well as refresh tokens. When an attacker steals an authorization code, the code is used twice (once by the attacker, and once by the legitimate RP). OAuth does not allow using an authorization code multiple times, yet 14% of the tested IdPs accept the code multiple times.

Refresh tokens that are not sender-constrained, i.e., not cryptographically bound to the RP, must use *token rotation*. A new refresh token is granted with each exchange and the old refresh token is invalidated but remembered. If the same refresh token is presented a second time, the IdP must assume that the refresh token was compromised and revoke the active refresh token. Only about half of the IdPs are compliant and use token rotation. Yet, 44% of the compliant IdPs still accept the old token. Of the IdPs that do not allow multiple exchanges of the same refresh token, none revoke the active refresh token.

6 USING OAUCH TO MOUNT ATTACKS

The purpose of OAuch is to measure the compliance of an IdP to the OAuth standards, and to document partially and unmitigated threats. It can also be used by an adversary to quickly uncover potential attack vectors. In this section, we test this assumption and use the results of Section 5.5 to investigate four threats. We use OAuch to list the potentially vulnerable IdPs and try to exploit them. Our results show that the analysis of OAuch is highly accurate and can be used to find actual vulnerabilities in implementations.

Despite the potential for abuse, our proof-of-concept attacks did not negatively impact any of the IdPs. All operators have been notified about these vulnerabilities as part of our responsible disclosure process. The attacks were executed after the vulnerabilities had been disclosed to the provider. As noted below, some providers had already fixed the vulnerabilities by that time.

6.1 Authorization Code Injection Attacks

In an authorization code injection attack, the attacker starts by stealing an authorization code from a user, for example, by tricking the user to install a malicious browser add-on, by abusing open redirects [41], by abusing proxy auto-config (PAC) files [22], by using URI scheme interception on mobile devices [46], or by abusing token leaks [59]. Once the authorization code has been acquired, the attacker uses the same RP on their own device and starts the authorization process. The stolen authorization code is injected into this process and is associated with the attacker's session. The attacker can now impersonate the victim in the RP.

To counter the attack, the IdP can implement the PKCE extension, which was specifically created to counter authorization code interception attacks. Alternatively, the RP can use the *nonce* parameter, as introduced by the OIDC standard. The nonce is a random value generated by the RP and can be included in the authorization request. The returned identity token contains this value, allowing the RP to verify that the received token belongs to the correct session. Note that the *nonce* parameter is specific to OIDC, whereas authorization code injection attacks apply to plain OAuth as well.

Because none of the tested IdPs require PKCE, no implementation fully mitigates the authorization code injection threat. If PKCE is supported, the threat is shown as *partially mitigated*. Otherwise, it is shown as *unmitigated*.

To test this attack on real implementations, we focused on the OIDC implementations and looked for websites that used these providers in the context of a social login. We found RPs for the following IdPs: Apple[7], BitBucket, Facebook, GitHub, Google, itsme, LinkedIn, Microsoft, Orcid, SalesForce, Twitch, and Yahoo. The use of PKCE or the *nonce* parameter by these RPs was virtually nonexistent. This made it easy to find one or more vulnerable RPs for each of these IdPs.

OAuch reported that none of the IdPs fully mitigated the authorization code injection threat. In most cases, RPs could have taken advantage of PKCE or the *nonce* parameter. Sadly, the results show that RPs cannot be expected to make the best security decisions. This highlights the importance of making strong security demands on the IdP side, which RPs are then forced to follow.

6.2 Implicit Flow RP Impersonation Attacks

The implicit flow does not support RP authentication, which makes it vulnerable to a particular type of impersonation attack [19]. To impersonate an RP, the attacker only needs to know the RP's identifier and redirect URI — two publicly known values. Note that these values can be collected from the RP even if it does not use the implicit flow. The only prerequisite for the attack is that the IdP supports the implicit flow. If the attacker can trick a user into installing a malicious application, the application can impersonate the RP by using the same app-uri (i.e., the RP's redirect URI that it uses for mobile and desktop applications) to steal access tokens that are linked to the impersonated RP.

One way to mitigate this threat is by explicitly asking the user to authorize each access token request. OAuch tests this behavior with a test case that is based on a heuristic (cf. Section 4.3). OAuch uses the implicit flow to send an authorization request, and if it receives an access token in a short amount of time, it assumes that the request was automatically granted and no explicit user authorization was requested.

To verify OAuch's analysis, we built a proof of concept based on the demonstration given in [18]. Of the 37 IdPs that OAuch

[7] The Apple OIDC implementation is not present in the ecosystem analysis because Apple requires its users to pay for a developer account. However, for the authorization code injection attack, RPs deployed by others are exploited.

examined for this threat, 20 were reported as vulnerable. By the time we implemented and executed our attack, two IdPs had already implemented the mitigation, another two IdPs had removed their support for the implicit flow, and one IdP discontinued public use of their API. Our tests showed that there were no false positives in the results of OAuch. All sites that were reported to be vulnerable were indeed exploitable. However, further analysis revealed that the heuristic in the test case misclassified two IdPs as not vulnerable, when they were actually exploitable. Thus, the actual failure rate of the test case is worse than what OAuch reported.

6.3 PKCE Downgrade Attacks

The PKCE extension is an important mitigation against authorization code misuse and CSRF attacks. The RP must add a random (and optionally hashed) value to the authorization request that is linked to the authorization code. When the RP tries to exchange the authorization code for an access token, it must present the (unhashed) random value in the token request. The IdP verifies that it matches with the original value before returning an access token.

Although the mechanics of the countermeasure are straightforward, a naive implementation may be susceptible to a downgrade attack. If an attacker can intercept the authorization request, they may potentially remove the random value from the authorization request. When the RP tries to exchange the authorization request, it will include the random value in the token request. If the IdP does not expect the random value (because the attacker removed it from the authorization request), it may choose to ignore the parameter. This eliminates the protection of PKCE without the RP's or IdP's knowledge. OAuch detected this bug in nine IdP implementations out of 21 that supported PKCE.

To validate the results, we searched for public RP's that used the vulnerable IdP's *and* used the PKCE extension. For five API providers, we were not able to find a public RP that met our criteria. One OIDC provider had several public RP's that used PKCE incorrectly by omitting a crucial parameter in the authorization request. We could not find an RP for this IdP that implemented PKCE correctly. The three remaining IdPs had fixed the vulnerability by the time we ran our PoC. The operators of these IdPs confirmed that the fixes were a direct result of our coordinated disclosure report.

Finally, we implemented our own RP that implemented PKCE correctly and relied on it to offer CSRF protection (i.e., in accordance with the OAuth Security BCP, it did not use the *state* parameter for CSRF protection). We tested this RP with each of the six vulnerable IdPs and performed a CSRF attack against the redirect uri as outlined in [54]. The attack succeeded for all the tested IdPs.

6.4 Clickjacking Attacks

Clickjacking attacks [1, 20, 34] are important threats to OAuth authorization endpoints. Although these attacks are not specific to OAuth, they are particularly dangerous in combination with the security-sensitive nature of the OAuth authorization process. They work by placing a transparent iframe above an innocuous-looking page controlled by the attacker. The OAuth authorization URI is loaded in the iframe, but it is not visible for the user. Despite being hidden, the browser still redirects all clicks to this transparent iframe. The attacker then tricks the user to click on the location of the authorization button by placing another (misleading) button on the page he controls, precisely under the authorization button. When the user tries to click the attacker's button, they unknowingly click the OAuth authorization button instead, unintentionally granting access.

OAuch identified 22 IdPs that do not use any of the two recommended HTTP headers to limit authorization page framing. These IdPs include eight top 1,000 sites, according to the Tranco top sites ranking list [23], and one banking site. We tested whether these implementations were vulnerable to a clickjacking attack, based on the OWASP testing guide [14]. The attack succeeded on 19 implementations, confirming that 86% of the IdPs identified by OAuch are exploitable.

The attack did not work on two IdPs because they used an intermediate page (without the recommended headers — as detected by OAuch) but then redirected the user via JavaScript to the actual authorization page. This page *was* protected by one of the recommended HTTP headers.

Another IdP used frame-busting JavaScript to stop clickjacking attacks. These scripts may not always be effective, and their use is discouraged in the OAuth threat model specification.

7 INTERPRETATION OF THE RESULTS

It is clear from the results in the previous sections that the OAuth ecosystem is in a bad state with respect to compliance. Most IdP implementations are missing crucial security requirements, and threats are often not or only partially mitigated. Nevertheless, it seems unlikely that most IdPs, including several high-profile targets, are left completely vulnerable. This raises the question of how the results should be interpreted.

OAuth IdPs can be better protected by implementing the missing security specifications. Yet, implementing all security specifications does not guarantee security. Likewise, not implementing all security specifications does not necessarily lead to vulnerabilities. An attacker may not be able to exploit a detected weakness.

Powerful attacker model The OAuth threat model assumes a powerful attacker with unlimited resources, advanced eavesdropping capabilities, and the ability to control multiple parties involved in the OAuth protocol. While this threat model anticipates nation-state attackers, most attackers are not nearly as powerful.

RP mitigations OAuch uses a worst case scenario where no mitigations are implemented by the RP. This is a reasonable assumption, as previous research [7, 24, 35, 37, 40, 42] and the results in Section 6.1 show that many RPs do not implement crucial countermeasures.

Complex exploitability Some threats make non-trivial assumptions. For example, the threat *Guessing Access Tokens* assumes that the attacker can brute force short access tokens without being blocked by the IdP. It is possible that an IdP does not protect against brute-force attacks [62], but even so, the attack remains challenging.

Another question that arises is why compliance with the security requirements of the OAuth protocol is so low, especially considering that OAuth *is* a security standard. In some cases, the IdP operator is aware that certain specifications are not implemented. From

our interactions with the operators, we can identify five types of justification:

Backward compatibility OAuth operators may not want to make changes that break RP implementations. It can make business sense to sacrifice some security for better compatibility. Still, many of the missing security requirements are backward compatible. They can be implemented without adversely affecting existing RPs.

Implementation or scalability issues It may not be possible to implement certain mitigations in the operator's environment. For example, APIs that process many requests, may not be able to verify the revocation status of a token on each request. This certainly applies to countermeasures that are active when tokens are used, but these only make up a small percentage of the OAuth security requirements.

Outdated standards The OAuth standard was published in late 2012, with additional standards following in the subsequent years. The most current source of up-to-date security guidelines is the *Security Best Current Practice* (BCP) document. However, this document is a working group draft. Some operators state that they do not support the BCP because it is not an official standard.

Shifting responsibilities Some OAuth operators seem reluctant to implement an IdP-side countermeasure if the threat can also be mitigated by an RP-side countermeasure. This makes business sense because it reduces development costs and processing overhead. In the event of a successful exploit, the operator can claim it is the RP's fault. Nevertheless, it is unwise from a security perspective. Many RPs do not follow the security best practices, so they might be vulnerable.

Reactive approach to security Some operators claim that they are aware of the problems in their implementation but prioritize other development work. Their approach to security seems to be reactive (i.e., when an actual exploit of their system is demonstrated) instead of proactive.

In most cases, the operators were unaware of the missing countermeasures. We observed three types of situations:

Invalid assumptions It is difficult to keep track of the latest developments in OAuth's security. Some assumptions that were true in the past may not be true today. For example, PKCE was originally intended only for public RPs but is now mandatory for confidential RPs as well. A few OAuth IdPs in our analysis support PKCE, but disallow it for confidential RPs.

Misplaced trust Programmers may not be aware of all the security requirements and may assume that the OAuth library they are using handles these. Unfortunately, libraries are configurable and may not have safe defaults. In addition, some parts of the OAuth implementation may not be managed by the library (e.g., the authorization page) but may still require countermeasures to be present.

Deceptive simplicity One design criterion of the OAuth protocol was to reduce its complexity as much as possible. This may entice developers to quickly build their own implementation of the protocol instead of using a well-established and maintained library.

OAuch provides the IdP operators with insights into which security specifications are missing and what the potential consequences are. Sometimes, business considerations may result in security risks being ignored. OAuch helps by allowing operators to make an informed choice.

8 ETHICS AND COORDINATED DISCLOSURE

OAuch is designed to be non-intrusive and avoids any harm to the IdPs that are tested. It generates a relatively small network data load and does not try to access private data of accounts that are not under the user's control. OAuch does not use tests that might impact the IdP's ability to respond to other requests (e.g., tests that evaluate DoS countermeasures).

In accordance with our coordinated disclosure policy, we reached out to all parties involved. Every operator received a detailed report of the test results, including the full log output of the failed test cases. They have received one year to process the report and update their services accordingly. In addition to these results, our work also identified non-security-related bugs in popular OAuth implementations, such as misnamed parameters or non-standard requirements. Because these issues might impact standards-compliant RPs, they were reported to the appropriate parties as well.

The general feedback we received during the coordinated disclosure process was positive, and most operators indicated that the report was useful. Some operators found it difficult to interpret the results of the test cases and to map them on vulnerabilities. We used this feedback to improve the reports by combining the test case results with the OAuth threat model. This clarifies the actual impact of the missing countermeasures.

Due to the large number of operators involved in the coordinated disclosure process, we were not able to follow up on every submission in detail. We did collaborate closely with three very large operators, due to the severity of the identified problems and the potentially large impact. We helped them to construct exploit prototypes to speed up the vulnerability assessment process.

9 CONCLUSION

In this paper, we set out to improve the overall security of the OAuth 2.0 landscape. We created a tool, called OAuch, that checks the compliance of IdPs with the OAuth security specifications and detects potential weaknesses in the implementation. The tool performs a comprehensive analysis, guided by the OAuth threat model. It generates a detailed report and gives tailor-made advice by identifying the security specifications that should be implemented to mitigate weaknesses.

We used OAuch on a set of 100 publicly available OAuth IdP implementations to determine the current state of practice in the OAuth ecosystem. We identify 1729 missing security specifications and found support for 157 deprecated features. The analysis of these results revealed a total of 431 unmitigated threats and an additional 693 partially mitigated threats. We present several possible explanations for the state of the ecosystem. These hypotheses are partly based on conversations with OAuth operators.

OAuch is intended to check for compliance with the OAuth standard, but it can also be used by an attacker as a guide to find weaknesses in target IdPs. To demonstrate this, we selected four attack vectors and used OAuch's output to identify potentially vulnerable sites. A manual check to verify that these sites were

exploitable revealed that the output of OAuch is very accurate, having a low false positive rate of 1.45% and false negative rate of 1.48%.

Unlike previous work, OAuch inspects the full set of security measures defined in the various OAuth standards, and reviews every relevant threat from the threat model. It is highly automated and can be used in any browser on any OAuth 2.0 implementation. Furthermore, it can easily be extended to support new OAuth standards.

ACKNOWLEDGMENTS

This research is partially funded by the Research Fund KU Leuven, the APISEC project, and by the Flemish Research Program Cybersecurity.

REFERENCES

[1] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. 2014. Clickjacking Revisited: A Perceptual View of UI Security. In *Proceedings of the 8th USENIX Workshop on Offensive Technologies (WOOT'14)*.

[2] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. 2014. Discovering concrete attacks on website authorization by formal analysis. *Journal of Computer Security* 22, 4 (2014).

[3] 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 *Proceedings of the 18th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer-Verlag, Berlin, Heidelberg, 21–41. https://doi.org/10.1007/978-3-030-80825-9_2

[4] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. 2018. WPSE: fortifying web protocols via browser-side security monitoring. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*. 1493–1510.

[5] Suresh Chari, Charanjit S Jutla, and Arnab Roy. 2011. Universally Composable Security Analysis of OAuth v2.0. *IACR Cryptol. ePrint Arch.* 2011 (2011).

[6] Eric Chen, Yutong Pei, Yuan Tian, Shuo Chen, Robert Kotcher, and Patrick Tague. 2016. 1000 ways to die in mobile OAuth. In *Blackhat USA*.

[7] Eric Y Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. 2014. OAuth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*. 892–903.

[8] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. 2020. The Cookie Hunter: Automated Black-Box Auditing for Web Authentication and Authorization Flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS'20)*. Association for Computing Machinery.

[9] Eugene Ferry, John O'Raw, and Kevin Curran. 2015. Security evaluation of the OAuth 2.0 framework. *Information and Computer Security* 23 (03 2015).

[10] Daniel Fett, Pedram Hosseyni, and Ralf Küsters. 2019. An Extensive Formal Security Analysis of the OpenID Financial-Grade API. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'19)* (San Francisco, CA).

[11] Daniel Fett, Ralf Kuesters, and Guido Schmitz. 2014. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. , 673–688 pages.

[12] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2016. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)* (Vienna, Austria). Association for Computing Machinery.

[13] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2017. The web SSO standard OpenID Connect: In-depth formal security analysis and security guidelines. In *Proceedings of the IEEE 30th Computer Security Foundations Symposium (CSF'17)*. 189–202.

[14] The OWASP Foundation. 2014. The OWASP Testing Guide 4.0. https://kennel209.gitbooks.io/owasp-testing-guide-v4/content/en/web_application_security_testing/testing_for_clickjacking_otg-client-009.html. [Online; accessed May 20, 2021].

[15] M. Ghasemisharif, C. Kanich, and J. Polakis. 2022. Towards Automated Auditing for Account and Session Management Flaws in Single Sign-On Deployments. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'22)*. IEEE Computer Society, Los Alamitos, CA, USA, 1524–1524. https://doi.org/10.1109/SP46214.2022.00095

[16] Mohammad Ghasemisharif, Amruta 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 *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 1475–1492.

[17] Roland Hedberg. 2012. OpenID Connect Deployment Verification Tool. https://kantarainitiative.org/confluence/download/attachments/3408008/Roland%20Hedberg%20-%20Kantara_summit_oic_test_tool.pdf

[18] Pili Hu and Wing Cheong Lau. 2014. How to Leak a 100-Million-Node Social Graph in Just One Week? A Reflection on OAuth and API Design in Online Social Networks. In *BlackHat USA*.

[19] Pili Hu, Ronghai Yang, Yue Li, and Wing Cheong Lau. 2014. Application impersonation: problems of OAuth and API design in online social networks. In *Proceedings of the second ACM conference on Online social networks*. 271–278.

[20] Lin-Shung Huang, Alex Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. 2012. Clickjacking: Attacks and defenses. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12)*.

[21] Ari-Pekka Koponen. 2016. *A secure OAuth 2.0 implementation model*. Master's thesis. University of Jyväskylä.

[22] Itzik Kotler and Amit Klein. 2016. Crippling HTTPS with unholy PAC. In *BlackHat USA*.

[23] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS'19)*.

[24] Wanpeng Li and Chris J Mitchell. 2014. Security issues in OAuth 2.0 SSO implementations. In *Proceedings of the International Conference on Information Security*. Springer.

[25] Wanpeng Li and Chris J Mitchell. 2016. Analysing the Security of Google's implementation of OpenID Connect. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'16)*. Springer, 357–376.

[26] Wanpeng Li, Chris J Mitchell, and Thomas Chen. 2018. Mitigating CSRF attacks on OAuth 2.0 and OpenID Connect. arXiv:1801.07983 [cs.CR]

[27] Wanpeng Li, Chris J Mitchell, and Thomas Chen. 2018. Mitigating CSRF attacks on OAuth 2.0 Systems. In *Proceedings of the 16th Annual Conference on Privacy, Security and Trust (PST'18)*. 1–5.

[28] Wanpeng Li, Chris J Mitchell, and Thomas Chen. 2018. Your code is my code: Exploiting a common weakness in OAuth 2.0 implementations. In *Proceedings of the Cambridge International Workshop on Security Protocols*. Springer.

[29] Wanpeng Li, Chris J. Mitchell, and Thomas Chen. 2019. 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 (SSR'19)* (London, United Kingdom). Association for Computing Machinery.

[30] Xinyu Li, Jing Xu, Zhenfeng Zhang, Xiao Lan, and Yuchen Wang. 2020. Modular Security Analysis of OAuth 2.0 in the Three-Party Setting. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 276–293. https://doi.org/10.1109/EuroSP48549.2020.00025

[31] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M Pai, and Sanjay Singh. 2011. Formal verification of OAuth 2.0 using Alloy framework. In *Proceedings of the International Conference on Communication Systems and Network Technologies*.

[32] Tamjid Al Rahat, Yu Feng, and Yuan Tian. 2019. OAUTHLINT: An Empirical Study on OAuth Bugs in Android Applications. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 293–304. https://doi.org/10.1109/ASE.2019.00036

[33] Tamjid Al Rahat, Yu Feng, and Yuan Tian. 2022. Cerberus: Query-driven Scalable Security Checking for OAuth Service Provider Implementations. (2022).

[34] Hossain Shahriar and Vamshee Krishna Devendran. 2014. Classification of clickjacking attacks and detection techniques. *Information Security Journal: A Global Perspective* 23, 4-6 (2014), 137–147.

[35] Mohamed Shehab and Fadi Mohsen. 2014. Towards enhancing the security of OAuth implementations in smart phones. In *Proceedings of the IEEE International Conference on Mobile Services*. 39–46.

[36] 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 *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'15)*. Springer, 239–260.

[37] San-Tsai Sun and Konstantin Beznosov. 2012. 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 (CCS'12)*. 378–390.

[38] The OWASP Foundation. 2013. *OWASP Top 10 - 2013*. Technical Report. http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf

[39] Hui Wang, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2016. The Achilles heel of OAuth: a multi-platform study of OAuth-based authentication. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC'16)*.

[40] Hui Wang, Yuanyuan Zhang, Juanru Li, Hui Liu, Wenbo Yang, Bodong Li, and Dawu Gu. 2015. Vulnerability assessment of OAuth implementations in Android applications. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*.

[41] Xianbo Wang, Wing Cheong Lau, Ronghai Yang, and Shangcheng Shi. 2019. Make Redirection Evil Again: URL Parser Issues in OAuth. In *BlackHat Asia*.
[42] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. 2016. Model-Based Security Testing: An Empirical Study on OAuth 2.0 Implementations. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIACCS'16)* (Xi'an, China).
[43] Yuchen Zhou and David Evans. 2014. SSOScan: Automated testing of web applications for Single Sign-On vulnerabilities. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*. 495–510.

STANDARDS

[44] John Bradley, Andrey Labunets, and Daniel Fett. 2020. OAuth 2.0 Security Best Current Practice. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics. [Online; accessed May 20, 2021].
[45] Brian Campbell, John Bradley, Nat Sakimura, and Torsten Lodderstedt. 2020. OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens. https://datatracker.ietf.org/doc/html/rfc8705. [Online; accessed May 20, 2021].
[46] William Denniss and John Bradley. 2017. OAuth 2.0 for Native Apps. https://datatracker.ietf.org/doc/html/rfc8252. [Online; accessed May 20, 2021].
[47] William Denniss, John Bradley, Michael Jones, and Hannes Tschofenig. 2019. OAuth 2.0 Device Authorization Grant. https://datatracker.ietf.org/doc/html/rfc8628. [Online; accessed May 20, 2021].
[48] Dick Hardt. 2012. The OAuth 2.0 Authorization Framework. https://datatracker.ietf.org/doc/html/rfc6749. [Online; accessed May 20, 2021].
[49] Dick Hardt and Michael Jones. 2012. The OAuth 2.0 Authorization Framework: Bearer Token Usage. https://datatracker.ietf.org/doc/html/rfc6750. [Online; accessed May 20, 2021].
[50] Michael Jones and Brian Campbell. 2015. OAuth 2.0 Form Post Response Mode. https://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html. [Online; accessed May 20, 2021].
[51] Michael Jones, Brian Campbell, and Chuck Mortimore. 2015. JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants. https://datatracker.ietf.org/doc/html/rfc7523. [Online; accessed May 20, 2021].
[52] Torsten Lodderstedt, Stefanie Dronia, and Marius Scurtescu. 2013. OAuth 2.0 Token Revocation. https://datatracker.ietf.org/doc/html/rfc7009. [Online; accessed May 20, 2021].
[53] Maciej Machulak and Justin Richer. 2018. User-Managed Access (UMA) 2.0 Grant for OAuth 2.0 Authorization. https://docs.kantarainitiative.org/uma/wg/rec-oauth-uma-grant-2.0.html. [Online; accessed May 20, 2021].
[54] Mark McGloin and Phil Hunt. 2013. OAuth 2.0 Threat Model and Security Considerations. https://datatracker.ietf.org/doc/html/rfc6819. [Online; accessed May 20, 2021].
[55] Nat Sakimura, John Bradley, and Naveen Agarwal. 2015. Proof Key for Code Exchange by OAuth Public Clients. https://datatracker.ietf.org/doc/html/rfc7636. [Online; accessed May 20, 2021].
[56] Nat Sakimura, John Bradley, Michael B. Jones, Breno de Medeiros, and Chuck Mortimore. 2014. OpenID Connect. https://openid.net/specs/openid-connect-core-1_0.html. [Online; accessed May 20, 2021].
[57] The OpenID Foundation. 2022. OpenID Certification. https://openid.net/certification/. [Online; accessed May 20, 2021].

VULNERABILITIES

[58] Daniella Genovese. 2019. Microsoft fixes login vulnerability. https://www.foxbusiness.com/technology/microsoft-vulnerability-login-system. [Online; accessed May 20, 2021].
[59] Cassio Gomes. 2019. Referer Leakage Vulnerability leads to OAuth token theft. https://hackerone.com/reports/787160. [Online; accessed September 21, 2021].
[60] Dan Goodin. 2020. Apple fixes bug that could have given hackers full access to user accounts. https://arstechnica.com/information-technology/2020/06/apple-fixes-bug-that-could-have-given-hackers-unauthorized-to-user-accounts/. [Online; accessed May 20, 2021].
[61] Abeerah Hashim. 2020. 10-Year Old Facebook OAuth Framework Flaw Discovered. https://latesthackingnews.com/2020/03/03/10-year-old-facebook-oauth-framework-flaw-discovered/. [Online; accessed May 20, 2021].
[62] Swati Khandelwal. 2016. Hacker Reveals How to Hack Any Facebook Account. https://thehackernews.com/2016/03/hack-facebook-account.html. [Online; accessed May 20, 2021].
[63] Max Moroz. 2017. Race Conditions in OAuth 2 API implementations. https://hackerone.com/reports/55140. [Online; accessed May 20, 2021].

A THE OAUTH PROTOCOL

The OAuth 2.0 protocol [48] is a popular authorization framework. The original specification defined four modes of operation, called

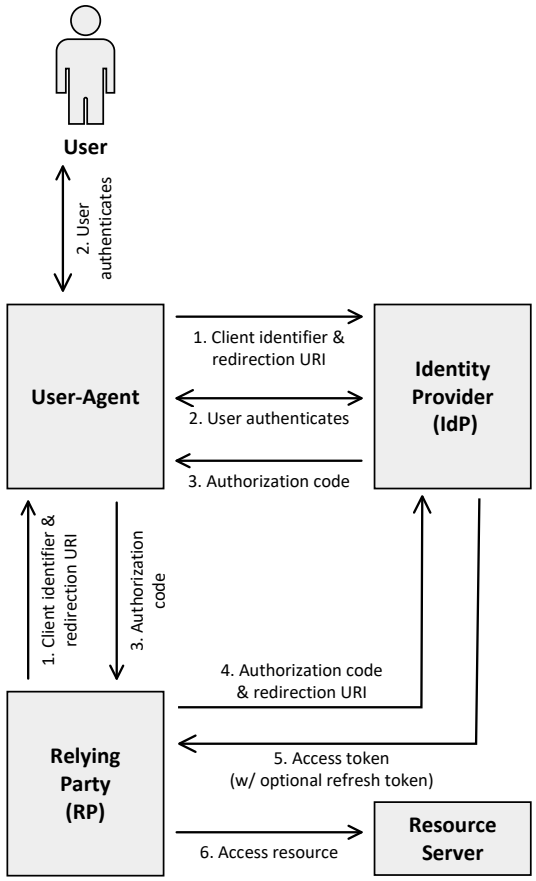

Figure 8: A schematic representation of the Authorization Code flow. Note: The lines illustrating steps 1, 2, and 3 are broken into two parts as they pass through the user agent.

grants or *flows*. One popular flow that involves a user is the *authorization code flow*. Figure 8 illustrates the actors in this grant type and the steps that have to be taken to authorize an RP and receive an access token. The *user agent* is the software used by the user to interact with the IdP (i.e., a browser, or some kind of web view embedded in an application).

The authorization process starts when the RP wants to access a protected resource. The RP sends the user agent a *redirection URI* that points to an appropriate IdP (step 1). The user agent navigates to the requested URI and starts the authentication and authorization process on the IdP (step 2). Authentication typically requires the user to log in with a password and/or some other means (e.g., an authenticator code). After identifying the user, the IdP asks the user to grant the RP access to the requested resource. If the user approves the authorization request, the IdP sends an *authorization code* to the user agent, which is then forwarded to the RP (step 3). The RP cannot use the authorization code to access the resource, but it can convert the code into an *access token* by sending it to the IdP (step 4). The IdP validates the code and sends back an access token to the RP (step 5). This access token can then be used to access the protected resource on the user's behalf (step 6).

Because the user only authenticates with the IdP, an RP never sees the user's credentials. Furthermore, the access token is sent directly from the IdP to the RP, avoiding any potential leaks in the user agent.

OAuch can test all the flows that are included in the original OAuth specification. Next to the authorization code flow, these flows are:

- *The Implicit grant.* This flow is similar to the authorization code grant, but instead of using an authorization code, the IdP sends the access token directly to the RP via the redirect URI (step 3). Because the RP now has the access token, there is no need for steps 4 and 5. This flow was originally meant for RPs implemented in a browser but has now been deprecated.
- *The Resource Owner Password Credentials grant.* In this flow, the RP uses the user's username and password to request a token from the IdP. This flow was initially intended for backward compatibility and trusted RPs created by the IdP provider. Due to security concerns, it has been deprecated.
- *The Client Credentials grant.* This flow does not involve a user but uses the client credentials as an authorization grant. It can be used when the RP is acting on its own behalf. This is the preferred flow to authorize machine-to-machine communication.

In addition to these flows, other flows have been defined. One of these flows is called the *device authorization grant* [47]. It allows input-constrained RPs (such as smart TVs, printers, ...) to obtain authorization by using a user agent on a separate device.

OpenID Connect extends OAuth's authorization code flow and implicit flow in such a way that identity tokens can be issued as well. OIDC also introduces a new flow, called the *hybrid flow*. This hybrid flow is a mix of the implicit and authorization code flows. It resembles the authorization code flow, but can also issue access tokens and identity tokens directly in the front channel (like the implicit flow).

B ECOSYSTEM RESULTS

This section contains the detailed output of the ecosystem analysis. Table 4 lists all the IdP implementations that have been tested. For every implementation, the supported flows are listed and statistics for the test case failure rates, threats, and deprecated features are given. OAuch automatically determines which flows are supported by a site. The *supported flows* column lists these results for every IdP. The following abbreviations are used:

- **ac** — The Authorization Code grant
- **cc** — The Client Credentials grant
- **im** — The Implicit grant
- **pw** — The Resource Owner Password Credentials grant
- **hy** — The Hybrid grant
- **dc** — The Device Authorization grant

The *Failure Rate* columns list the failure rates, as described in Section 4.5, of the executed test cases. The failure rates are reported for each requirement level, as well as an overall rate that combines the results of the three levels. Lower scores represent better adherence to the OAuth standards. The *Threats* and *Deprecated Features* columns show absolute numbers. Mitigated, partially mitigated and unmitigated threats are reported in separate columns. As explained in Section 5.5, a lower overall failure rate is correlated with a mitigation of more threats.

Table 5 shows the aggregated results of the threat statistics. For each threat that is supported by OAuch, the results of all IdPs for which that threat is relevant are combined. Not all threats are relevant to all IdPs. Some threats depend on specific features being enabled. The *Relevant* column shows the percentage of IdPs for which the threat is considered relevant. The three following columns contain the percentages of IdPs where the threat is fully, partially, or not mitigated. These percentages are only calculated over IdPs for which the threat was relevant. The final column lists related work that discusses the threat in the context of one or more public OAuth deployments. Threats denoted with a † are added to incorporate countermeasures from popular OAuth extensions that are not referenced in the official threat model.

Table 4: The detailed site results of the ecosystem analysis

	Supported	Failure Rate				Threats			Deprecated
Identity Provider	**Flows**	**May**	**Should**	**Must**	**Overall**	**Mit.**	**Part. Mit.**	**Not Mit.**	**Features**
API Providers									
Acuity	ac	100%	44%	**19%**	33%	13	6	**3**	0
Aha	ac, cc, im	60%	55%	**18%**	31%	11	9	**5**	2
Amazon	ac, im	80%	50%	**10%**	24%	23	6	**1**	4
Autodesk	ac, im	83%	55%	**16%**	31%	21	6	**4**	1
Avaza	ac, im	67%	55%	**32%**	40%	12	13	**6**	3
Basecamp	ac	80%	56%	**29%**	39%	16	7	**3**	2
BitBucket	ac, cc, im	80%	82%	**24%**	41%	12	12	**5**	2
Box	ac, cc	57%	64%	**21%**	35%	13	8	**8**	1
Campaign Monitor	ac	100%	86%	**36%**	50%	11	9	**6**	1
CitiBank	ac	67%	25%	**10%**	19%	25	2	**1**	0
ClickUp	ac	100%	78%	**22%**	41%	10	10	**2**	0
Dailymotion	ac, cc, im	100%	54%	**22%**	37%	15	11	**3**	3
DaniWeb	im	75%	62%	**10%**	30%	8	4	**4**	3
DevianArt	ac	71%	54%	**18%**	33%	15	10	**4**	2
Dexcom	ac	100%	71%	**34%**	47%	16	5	**7**	1
Discord	ac, cc, im	57%	53%	**11%**	26%	21	10	**2**	2
Dribbble	ac	100%	60%	**18%**	35%	9	10	**3**	1
Drift	ac	83%	82%	**27%**	44%	13	10	**4**	1
Drip	ac, cc	100%	60%	**19%**	36%	10	9	**3**	1
DropBox	ac	80%	44%	**25%**	35%	15	4	**3**	1
eBay	ac	100%	44%	**23%**	35%	16	5	**5**	1
Eventbrite	ac, im	80%	75%	**26%**	40%	14	6	**5**	1
Everypixel	cc	100%	60%	**11%**	25%	8	1	**0**	1
Facebook	ac, cc, im	80%	45%	**21%**	33%	13	5	**7**	3
FatSecret	cc	100%	50%	**14%**	29%	3	0	**2**	1
Figma	ac	80%	50%	**29%**	38%	16	7	**3**	0
FitBit	ac, cc, im	43%	60%	**15%**	28%	19	9	**6**	2
Flowdock	ac, pw	83%	45%	**20%**	33%	12	12	**7**	3
Formstack	ac, im	83%	71%	**36%**	47%	11	9	**10**	1
Foursquare	ac, im	80%	70%	**18%**	35%	10	8	**7**	3
Frame.io	ac, im	67%	25%	**11%**	20%	24	4	**2**	1
Freesound	ac	83%	91%	**19%**	42%	13	8	**6**	2
FreshBooks	ac, im	43%	47%	**20%**	29%	14	13	**6**	3
GetResponse	ac, cc, im	67%	73%	**24%**	39%	14	11	**5**	2
GitHub	ac	80%	56%	**36%**	45%	11	4	**7**	1
Harvest	ac	80%	55%	**26%**	37%	13	9	**4**	1
HelpScout	ac, cc	67%	64%	**19%**	33%	15	9	**3**	2
Heroku	ac	80%	80%	**26%**	42%	15	8	**3**	0
HubSpot	ac, im	100%	73%	**19%**	38%	14	12	**3**	2
Imgur	ac, im	100%	80%	**31%**	48%	13	9	**8**	2
Indeed	ac	100%	50%	**26%**	39%	14	4	**4**	0
InoReader	ac	83%	90%	**11%**	35%	17	7	**3**	1
Jamendo	ac	100%	91%	**21%**	45%	10	13	**4**	2
LinkedIn	ac	100%	44%	**19%**	33%	13	6	**3**	0
LiveChat	ac, im	100%	88%	**23%**	40%	13	12	**4**	2
Lufthansa	cc	100%	60%	**21%**	35%	3	0	**2**	1
MailChimp	ac	80%	100%	**25%**	42%	10	4	**8**	0
Mercedes Benz	ac	60%	45%	**5%**	18%	22	5	**1**	2
MicroBilt	cc	100%	50%	**29%**	36%	6	1	**2**	0
MindMeister	ac, cc, im	60%	38%	**14%**	24%	18	3	**6**	2
Monday	ac	80%	62%	**25%**	38%	15	5	**3**	0

Table 4: The detailed site results of the ecosystem analysis

	Supported	Failure Rate				Threats			Deprecated
Identity Provider	**Flows**	**May**	**Should**	**Must**	**Overall**	**Mit.**	**Part. Mit.**	**Not Mit.**	**Features**
MusicBrainz	ac	75%	55%	**14%**	28%	16	4	**3**	2
Netatmo	ac, pw	100%	70%	**23%**	40%	13	8	**9**	3
Nightbot	ac, cc, im	83%	85%	**11%**	36%	16	8	**6**	2
Patreon	ac	83%	60%	**19%**	35%	14	9	**4**	0
Podio	ac, cc, im, pw	80%	89%	**28%**	44%	15	6	**12**	2
Pushbullet	ac, im	100%	80%	**27%**	46%	11	8	**6**	2
Redbooth	ac, cc	83%	42%	**22%**	33%	11	12	**4**	2
Reddit	ac, cc	83%	31%	**12%**	23%	21	4	**3**	1
Slack	ac	80%	56%	**33%**	43%	9	8	**5**	1
SmartSheet	ac	67%	71%	**31%**	42%	15	8	**4**	0
Spotify	ac, cc, im	100%	36%	**20%**	31%	19	7	**5**	3
Stack Exchange	ac	80%	71%	**35%**	47%	9	6	**7**	0
Starling Bank	ac	100%	40%	**22%**	33%	18	6	**2**	0
StockTwits	ac, im	80%	82%	**30%**	47%	7	7	**11**	3
Strava	ac	80%	82%	**31%**	47%	11	7	**8**	1
Surveymonkey	ac	100%	62%	**22%**	38%	8	10	**4**	0
Teamleader	ac, im	100%	64%	**17%**	35%	18	8	**4**	2
Tipeeestream	ac, im	100%	58%	**14%**	32%	17	7	**5**	3
TSheets	ac	100%	70%	**18%**	37%	13	8	**5**	1
Twitter	cc	100%	60%	**11%**	25%	6	2	**1**	0
Typeform	ac	100%	44%	**22%**	35%	15	4	**3**	0
Uber	ac, im	60%	36%	**21%**	29%	18	2	**5**	1
Vimeo	ac	100%	75%	**12%**	33%	14	5	**3**	0
VK	ac, cc, im	80%	40%	**34%**	41%	13	3	**5**	2
Withings	ac	80%	70%	**16%**	34%	14	6	**2**	0
WordPress	ac, im	60%	67%	**21%**	34%	13	9	**3**	2
Wrike	ac	67%	64%	**19%**	34%	18	5	**4**	1
Yandex	ac, cc, im	83%	50%	**18%**	32%	17	10	**4**	3
Zoom	ac, cc, im	33%	47%	**23%**	29%	19	6	**7**	3
OpenID Connect Providers									
Adobe	ac, hy, im	80%	40%	**20%**	28%	22	12	**2**	4
Battle.net	ac, cc	100%	58%	**22%**	35%	11	8	**6**	2
GitLab	ac, im	75%	31%	**15%**	22%	23	7	**3**	2
Globus	ac, cc	50%	62%	**11%**	24%	20	5	**3**	0
Google	ac, hy, im, dc	67%	44%	**12%**	20%	26	4	**4**	7
IBM	ac, cc	100%	33%	**5%**	18%	26	4	**1**	0
Intuit	ac, cc	75%	38%	**11%**	21%	25	5	**1**	1
Itsme	ac	83%	44%	**16%**	28%	12	5	**5**	0
Legrand	ac	100%	50%	**27%**	36%	16	7	**8**	1
Microsoft	ac	83%	33%	**13%**	23%	20	9	**1**	2
Mozilla	ac, hy	50%	36%	**22%**	27%	18	6	**3**	3
Openstack	ac, hy	75%	45%	**12%**	23%	14	4	**7**	4
Orcid	ac, hy, im	100%	64%	**24%**	37%	20	7	**6**	3
Paypal	ac, cc, hy	100%	38%	**19%**	28%	19	10	**2**	3
PhantAuth	ac, hy	83%	42%	**20%**	28%	17	6	**11**	5
Salesforce	ac	75%	42%	**16%**	24%	20	9	**1**	1
Signicat	ac	50%	58%	**10%**	22%	26	5	**1**	1
Twitch	ac, cc, hy, im	100%	58%	**28%**	38%	20	6	**6**	3
Xero	ac	75%	38%	**4%**	14%	30	4	**0**	1
Yahoo	ac	50%	33%	**14%**	20%	23	7	**0**	1

Table 5: Aggregated results of the threat statistics

		Threats			Related
Threat	**Relevant**	**Mit.**	**Part. Mit.**	**Not Mit.**	**Work**
Obtaining Client Secrets	37%	45.9%	0.0%	54.1%	[6, 7, 9, 32, 40]
Obtaining Refresh Tokens	32%	0.0%	100.0%	0.0%	[9, 32, 42]
Obtaining Access Tokens	100%	27.0%	0.0%	73.0%	[9, 30, 32, 40]
Open Redirectors on Client	95%	52.6%	29.5%	17.9%	[4, 9, 28]
Password Phishing by Counterfeit Authorization Server	95%	100.0%	0.0%	0.0%	[9, 25, 29, 37, 39, 40, 42]
Malicious Client Obtains Existing Authorization by Fraud	95%	61.1%	24.2%	14.7%	[4, 9, 28, 40]
Open Redirector	95%	51.6%	30.5%	17.9%	[4, 9, 28, 33]
Eavesdropping Access Tokens in Transit	100%	96.0%	4.0%	0.0%	[6, 7, 9, 25, 29, 37, 39, 40, 42]
Disclosure of Client Credentials during Transmission	100%	96.0%	4.0%	0.0%	[9, 25, 29, 30, 37, 39, 40, 42]
Obtaining Client Secret by Online Guessing	99%	45.5%	24.2%	30.3%	[9]
Eavesdropping or Leaking Authorization ‘codes’	94%	10.6%	86.2%	3.2%	[6, 7, 9, 28, 30, 33, 37, 40]
Online Guessing of Authorization ‘codes’	94%	23.4%	76.6%	0.0%	[9]
Authorization ‘code’ Phishing	92%	88.0%	2.2%	9.8%	[28, 37]
Authorization ‘code’ Leakage through Counterfeit Client	94%	35.1%	56.4%	8.5%	[4, 6, 9, 28, 32]
CSRF Attack against redirect-uri	94%	96.8%	0.0%	3.2%	[3, 4, 6, 9, 24, 25, 28–30, 33, 36, 37, 42]
Clickjacking Attack against Authorization	95%	30.5%	48.4%	21.1%	[9, 36]
Code Substitution (OAuth Login)	18%	94.4%	0.0%	5.6%	[6, 7, 9, 32, 39]
Access Token Leak in Browser History	39%	0.0%	69.2%	30.8%	[6, 32, 40]
Accidental Exposure of Passwords at Client Site	3%	0.0%	0.0%	100.0%	
Client Obtains Scopes without End-User Authorization	3%	0.0%	0.0%	100.0%	
Client Obtains Refresh Token through Automatic Authorization	3%	0.0%	0.0%	100.0%	
Obtaining User Passwords on Transport	3%	100.0%	0.0%	0.0%	[9, 25, 29, 37, 39, 42]
Eavesdropping Refresh Tokens from Authorization Server	65%	96.9%	3.1%	0.0%	[9, 25, 29, 33, 37, 39, 40, 42]
Obtaining Refresh Token from Authorization Server Database	63%	85.7%	3.2%	11.1%	
Obtaining Refresh Token by Online Guessing	65%	50.8%	47.7%	1.5%	
Refresh Token Phishing by Counterfeit Authorization Server	65%	100.0%	0.0%	0.0%	[9, 25, 29, 37, 39, 42]
Eavesdropping Access Tokens on Transport	94%	27.7%	72.3%	0.0%	[9, 25, 29, 37, 39, 40, 42]
Replay of Authorized Resource Server Requests	94%	97.9%	2.1%	0.0%	[9, 25, 29, 37, 39, 42]
Guessing Access Tokens	94%	22.3%	67.0%	10.6%	
Leak of Confidential Data in HTTP Proxies	99%	57.6%	16.2%	26.3%	
Token Leakage via Log Files and HTTP Referrers	94%	59.6%	0.0%	40.4%	
Redirect URI Validation Attacks on Implicit Grant	37%	81.1%	5.4%	13.5%	[4, 9, 28, 30]
Leakage from the Authorization Server	95%	43.2%	49.5%	7.4%	
Authorization Code in Browser History	94%	87.2%	0.0%	12.8%	[32]
Authorization Code Injection	94%	0.0%	22.3%	77.7%	
PKCE Downgrade Attack	21%	57.1%	0.0%	42.9%	
Unverified JWTs (resource server) †	13%	92.3%	0.0%	7.7%	
Unverified JWTs for client authentication †	3%	33.3%	66.7%	0.0%	
Abuse of revoked tokens †	15%	66.7%	26.7%	6.7%	[15, 16]
Unauthorized revocation of tokens †	15%	80.0%	0.0%	20.0%	
Abuse of incomplete/invalid identity tokens †	19%	57.9%	42.1%	0.0%	
Falsifying identity tokens †	19%	89.5%	0.0%	10.5%	

C TEST CASE MAPPINGS

OAuch maps the threats that are defined in the OAuth threat model and the Security Best Current Practices, as well as the additional threats from other standards, to specific test cases. The result of this process, described in Section 4.3, is detailed below.

Each threat has a description and a list of test cases that are linked to it. The test cases are grouped into categories that refer to the OAuth infrastructure they apply to. Test cases with the same name but in different categories, are distinct (but similar) test cases.

Obtaining Client Secrets The attacker could try to get access to the secret of a particular client in order to obtain tokens on behalf of the attacked client with the privileges of that 'client_id' acting as an instance of the client.

Associated test cases (1):

- *Auth. endpoint:* RequireUserConsent

Obtaining Refresh Tokens Depending on the client type, there are different ways that refresh tokens may be revealed to an attacker. An attacker may obtain the refresh tokens issued to a web application by way of overcoming the web server's security controls. On native clients, refresh tokens may be read from the local file system or the device could be stolen or cloned.

Associated test cases (6):

- *Revocation endpoint:* CanRefreshTokensBeRevoked
- *Token endpoint:* InvalidatedRefreshToken, IsRefreshAuthenticationRequired, IsRefreshBoundToClient, RefreshTokenRevokedAfterUse, UsesTokenRotation

Obtaining Access Tokens Depending on the client type, there are different ways that access tokens may be revealed to an attacker. Access tokens could be stolen from the device if the application stores them in a storage device that is accessible to other applications.

Associated test cases (1):

- *Tokens:* TokenTimeout

Open Redirectors on Client An open redirector is an endpoint using a parameter to automatically redirect a user agent to the location specified by the parameter value without any validation. If the authorization server allows the client to register only part of the redirect URI, an attacker can use an open redirector operated by the client to construct a redirect URI that will pass the authorization server validation but will send the authorization 'code' or access token to an endpoint under the control of the attacker.

Associated test cases (2):

- *Auth. endpoint:* RedirectUriFullyMatched, RedirectUriPathMatched

Password Phishing by Counterfeit Authorization Server Auth makes no attempt to verify the authenticity of the authorization server. A hostile party could take advantage of this by intercepting the client's requests and returning misleading or otherwise incorrect responses. This could be achieved using DNS or Address Resolution Protocol (ARP) spoofing. Wide deployment of OAuth and similar protocols may cause users to become inured to the practice of being redirected to web sites where they are asked to enter their passwords. If users are not careful to verify the authenticity of these web sites before entering their credentials, it will be possible for attackers to exploit this practice to steal users' passwords.

Associated test cases (3):

- *Auth. endpoint:* HasValidCertificate, IsHttpsRequired, IsModernTlsSupported

Malicious Client Obtains Existing Authorization by Fraud Authorization servers may wish to automatically process authorization requests from clients that have been previously authorized by the user. When the user is redirected to the authorization server's end-user authorization endpoint to grant access, the authorization server detects that the user has already granted access to that particular client. Instead of prompting the user for approval, the authorization server automatically redirects the user back to the client. A malicious client may exploit that feature and try to obtain such an authorization 'code' instead of the legitimate client.

Associated test cases (3):

- *Auth. endpoint:* RedirectUriFullyMatched, RedirectUriPathMatched, RequireUserConsent

Open Redirector An attacker could use the end-user authorization endpoint and the redirect URI parameter to abuse the authorization server as an open redirector. An open redirector is an endpoint using a parameter to automatically redirect a user agent to the location specified by the parameter value without any validation. An attacker could utilize a user's trust in an authorization server to launch a phishing attack.

Associated test cases (3):

- *Auth. endpoint:* InvalidRedirect, RedirectUriFullyMatched, RedirectUriPathMatched

Eavesdropping Access Tokens in Transit Attackers may attempt to eavesdrop access tokens in transit from the authorization server to the client.

Associated test cases (8):

- *Device auth. endpoint:* HasValidCertificate, IsHttpsRequired, IsModernTlsSupported
- *Revocation endpoint:* IsModernTlsSupported, IsRevocationEndpointSecure
- *Token endpoint:* HasValidCertificate, IsHttpsRequired, IsModernTlsSupported

Disclosure of Client Credentials during Transmission An attacker could attempt to eavesdrop the transmission of client credentials between the client and server during the client authentication process or during OAuth token requests.

Associated test cases (4):

- *Token endpoint:* HasValidCertificate, IsAsymmetricClientAuthenticationUsed, IsHttpsRequired, IsModernTlsSupported

Obtaining Client Secret by Online Guessing An attacker may try to guess valid 'client_id'/secret pairs.

Associated test cases (3):

- *Token endpoint:* ClientSecretEntropyMinReq, ClientSecretEntropySugReq, IsAsymmetricClientAuthenticationUsed

Eavesdropping or Leaking Authorization 'codes' An attacker could try to eavesdrop transmission of the authorization 'code' between the authorization server and client. Furthermore, authorization 'codes' are passed via the browser, which may unintentionally leak those codes to untrusted web sites and attackers in different ways.

Associated test cases (4):

- *Token endpoint:* AuthorizationCodeTimeout, IsCodeBoundToClient, MultipleCodeExchanges, TokenValidAfterMultiExchange

Online Guessing of Authorization 'codes' An attacker may try to guess valid authorization 'code' values and send the guessed code value using the grant type 'code' in order to obtain a valid access token.

Associated test cases (6):

- *Token endpoint:* AuthorizationCodeTimeout, IsClientAuthenticationRequired, IsCodeBoundToClient, RedirectUriChecked
- *Tokens:* AuthorizationCodeEntropyMinReq, AuthorizationCodeEntropySugReq

Authorization 'code' Phishing A hostile party could impersonate the client site and get access to the authorization 'code'. This could be achieved using DNS or ARP spoofing. This applies to clients, which are web applications; thus, the redirect URI is not local to the host where the user's browser is running.

Associated test cases (2):

- *Token endpoint:* IsClientAuthenticationRequired, IsCodeBoundToClient

Authorization 'code' Leakage through Counterfeit Client The attacker leverages the authorization 'code' grant type in an attempt to get another user (victim) to log in, authorize access to his/her resources, and subsequently obtain the authorization 'code' and inject it into a client application using the attacker's account. The goal is to associate an access authorization for resources of the victim with the user account of the attacker on a client site. The attacker abuses an existing client application and combines it with his own counterfeit client web site. The attacker depends on the victim expecting the client application to request access to a certain resource server. The victim, seeing only a normal request from an expected application, approves the request. The attacker then uses the victim's authorization to gain access to the information unknowingly authorized by the victim.

Associated test cases (3):

- *Auth. endpoint:* RedirectUriFullyMatched, RedirectUriPathMatched
- *Token endpoint:* RedirectUriChecked

CSRF Attack against redirect-uri Cross-site request forgery (CSRF) is a web-based attack whereby HTTP requests are transmitted from a user that the web site trusts or has authenticated. CSRF attacks on OAuth approvals can allow an attacker to obtain authorization to OAuth protected resources without the consent of the user.

Associated test cases (3):

- *Auth. endpoint:* StatePresent
- *ID tokens:* NoncePresentInToken
- *PKCE:* IsPkceImplemented

Clickjacking Attack against Authorization With clickjacking, a malicious site loads the target site in a transparent iFrame overlaid on top of a set of dummy buttons that are carefully constructed to be placed directly under important buttons on the target site. When a user clicks a visible button, they are actually clicking a button (such as an 'Authorize' button) on the hidden page.

Associated test cases (2):

- *Auth. endpoint:* HasContentSecurityPolicy, HasFrameOptions

Code Substitution (OAuth Login) An attacker could attempt to log into an application or web site using a victim's identity. Applications relying on identity data provided by an OAuth protected service API to login users are vulnerable to this threat. This pattern can be found in so-called 'social login' scenarios.

Associated test cases (2):

- *Token endpoint:* IsClientAuthenticationRequired, IsCodeBoundToClient

Access Token Leak in Browser History An attacker could obtain the token from the browser's history. Note that this means the attacker needs access to the particular device.

Associated test cases (5):

- *API endpoint:* TokenAsQueryParameterDisabled
- *Auth. endpoint:* SupportsPostResponseMode
- *Token endpoint:* HasCacheControlHeader, HasPragmaHeader
- *Tokens:* TokenTimeout

Accidental Exposure of Passwords at Client Site If the client does not provide enough protection, an attacker or disgruntled employee could retrieve the passwords for a user.

Associated test cases (1):

- *Token endpoint:* IsPasswordFlowDisabled

Client Obtains Scopes without End-User Authorization All interaction with the resource owner is performed by the client. Thus it might, intentionally or unintentionally, happen that the client obtains a token with scope unknown for, or unintended by, the resource owner. For example, the resource owner might think the client needs and acquires read-only access to its media storage only but the client tries to acquire an access token with full access permissions.

Associated test cases (1):

- *Token endpoint:* IsPasswordFlowDisabled

Client Obtains Refresh Token through Automatic Authorization All interaction with the resource owner is performed by the client. Thus it might, intentionally or unintentionally, happen that the client obtains a long-term authorization represented by a refresh token even if the resource owner did not intend so.

Associated test cases (1):

- *Token endpoint:* IsPasswordFlowDisabled

Obtaining User Passwords on Transport An attacker could attempt to eavesdrop the transmission of end-user credentials with the grant type 'password' between the client and server.

Associated test cases (3):

- *Token endpoint:* HasValidCertificate, IsHttpsRequired, IsModernTlsSupported

Eavesdropping Refresh Tokens from Authorization Server An attacker may eavesdrop refresh tokens when they are transmitted between the authorization server and the client.

Associated test cases (3):

- *Token endpoint:* HasValidCertificate, IsHttpsRequired, IsModernTlsSupported

Obtaining Refresh Token from Authorization Server Database This threat is applicable if the authorization server stores refresh tokens as handles in a database. An attacker may obtain refresh tokens from the authorization server's database by gaining access to the database or launching a SQL injection attack.

Associated test cases (2):

- *Token endpoint:* IsRefreshAuthenticationRequired, IsRefreshBoundToClient

Obtaining Refresh Token by Online Guessing An attacker may try to guess valid refresh token values and send it using the grant type 'refresh_token' in order to obtain a valid access token.

Associated test cases (4):

- *Token endpoint:* IsRefreshAuthenticationRequired, IsRefreshBoundToClient
- *Tokens:* RefreshTokenEntropyMinReq, RefreshTokenEntropySugReq

Refresh Token Phishing by Counterfeit Authorization Server An attacker could try to obtain valid refresh tokens by proxying requests to the authorization server. Given the assumption that the authorization server URL is well-known at development time or can at least be obtained from a well-known resource server, the attacker must utilize some kind of spoofing in order to succeed.

Associated test cases (1):

- *Token endpoint:* HasValidCertificate

Eavesdropping Access Tokens on Transport An attacker could try to obtain a valid access token on transport between the client and resource server. As access tokens are shared secrets between the authorization server and resource server, they should be treated with the same care as other credentials (e.g., end-user passwords).

Associated test cases (4):

- *API endpoint:* HasValidCertificate, IsHttpsRequired, IsModernTlsSupported
- *Tokens:* TokenTimeout

Replay of Authorized Resource Server Requests An attacker could attempt to replay valid requests in order to obtain or to modify/destroy user data.

Associated test cases (3):

- *API endpoint:* HasValidCertificate, IsHttpsRequired, IsModernTlsSupported

Guessing Access Tokens Where the token is a handle, the attacker may attempt to guess the access token values based on knowledge they have from other access tokens.

Associated test cases (3):

- *Tokens:* AccessTokenEntropyMinReq, AccessTokenEntropySugReq, TokenTimeout

Leak of Confidential Data in HTTP Proxies An OAuth HTTP authentication scheme as discussed in RFC6749 is optional. However, RFC2616 relies on the Authorization and WWW-Authenticate headers to distinguish authenticated content so that it can be protected. Proxies and caches, in particular, may fail to adequately protect requests not using these headers. For example, private authenticated content may be stored in (and thus be retrievable from) publicly accessible caches.

Associated test cases (2):

- *Token endpoint:* HasCacheControlHeader, HasPragmaHeader

Token Leakage via Log Files and HTTP Referrers If access tokens are sent via URI query parameters, such tokens may leak to log files and the HTTP 'referer'.

Associated test cases (1):

- *API endpoint:* TokenAsQueryParameterDisabled

Redirect URI Validation Attacks on Implicit Grant Implicit clients can be subject to an attack that utilizes the fact that user agents re-attach fragments to the destination URL of a redirect if the location header does not contain a fragment. This allows circumvention even of very narrow redirect URI patterns, but not strict URL matching.

Associated test cases (3):

- *Auth. endpoint:* FragmentFix, RedirectUriFullyMatched, RedirectUriPathMatched

Leakage from the Authorization Server An attacker can learn 'state' from the authorization request if the authorization endpoint at the authorization server contains links or third-party content.

Associated test cases (5):

- *Auth. endpoint:* ReferrerPolicyEnforced, SupportsPostResponseMode
- *Token endpoint:* IsCodeBoundToClient, MultipleCodeExchanges, TokenValidAfterMultiExchange

Authorization Code in Browser History When a browser navigates to 'client.example/redirection_endpoint?code=abcd' as a result of a redirect from a provider's authorization endpoint, the URL including the authorization code may end up in the browser's history. An attacker with access to the device could obtain the code and try to replay it.

Associated test cases (2):

- *Auth. endpoint:* SupportsPostResponseMode
- *Token endpoint:* MultipleCodeExchanges

Authorization Code Injection In an authorization code injection attack, the attacker attempts to inject a stolen authorization code into the attacker's own session with the client. The aim is to associate the attacker's session at the client with the victim's resources or identity.

Associated test cases (3):

- *PKCE:* HashedPkceDisabled, IsPkceImplemented, IsPkceRequired

PKCE Downgrade Attack An authorization server that supports PKCE but does not make its use mandatory for all flows can be susceptible to a PKCE downgrade attack.

Associated test cases (2):

- *PKCE:* IsPkceDowngradeDetected, IsPkcePlainDowngradeDetected

Unverified JWTs (resource server) An attacker can remove or forge the signature of a JWT to impersonate another user.

Associated test cases (1):

- *JWT:* AcceptsNoneSignature

Unverified JWTs for client authentication An attacker can use an expired or otherwise invalid token to impersonate another user.

Associated test cases (9):

- *JWT:* HasAudienceClaim, HasIssuerClaim, HasSubjectClaim, IsExpirationChecked, IsIssuedAtChecked, IsJwtReplayDetected, IsNotBeforeChecked, IsSignatureChecked, IsSignatureRequired

Abuse of revoked tokens Leaked (and potentially long-lived) access or refesh tokens that cannot be revoked may enable an attacker to impersonate a user.

Associated test cases (4):

- *Revocation endpoint:* AccessRevokesRefresh, CanAccessTokensBeRevoked, CanRefreshTokensBeRevoked, RefreshRevokesAccess

Unauthorized revocation of tokens An authentication server that supports token revocation must verify the ownership of a token before revocation.

Associated test cases (2):

- *Revocation endpoint:* IsBoundToClient, IsClientAuthRequired

Abuse of incomplete/invalid identity tokens An attacker may attempt to re-use an identity token that was acquired for another client or for another authorization session.

Associated test cases (12):

- *ID tokens:* CodeHashValid, HasAuthorizedParty, HasAzpForMultiAudience, HasCorrectAudience, HasCorrectIssuer, HasCorrectMac, HasRequiredClaims, IsAccessTokenHashCorrect, IsAccessTokenHashPresent, IsAuthorizationCodeHashPresent, KeyReferences, NoncePresentInToken

Falsifying identity tokens Resource servers that do not verify the signature of an identity token, or that accept identity tokens that are signed with weak keys, are subject to an impersonation attack.

Associated test cases (2):

- *ID tokens:* ClientSecretLongEnough, IsSigned