Cerberus: Query-driven Scalable Security Checking for OAuth Service Provider Implementations

Conference Paper · May 2022

CITATIONS
0

READS
41

3 authors, including:

Tamjid Al Rahat
University of Virginia
6 PUBLICATIONS **11** CITATIONS
SEE PROFILE

Yuan Tian
University of Virginia
53 PUBLICATIONS **502** CITATIONS
SEE PROFILE

Cerberus: Query-driven Scalable Security Checking for OAuth Service Provider Implementations

Tamjid Al Rahat
University of Virginia
tr9wr@virginia.edu

Yu Feng
University of California, Santa Barbara
yufeng@cs.ucsb.edu

Yuan Tian
University of Virginia
yuant@virginia.edu

ABSTRACT

OAuth protocols have been widely adopted to simplify user authentication and service authorization for third-party applications. However, little effort has been devoted to automatically checking the security of the libraries that service providers widely use. In this paper, we formalize the OAuth specifications and security best practices, and design *Cerberus*, an automated static analyzer, to find logical flaws and identify vulnerabilities in the implementation of OAuth service provider libraries. To efficiently detect security violations in a large codebase of service provider implementation, *Cerberus* employs a query-driven algorithm for answering queries about OAuth specifications. We demonstrate the effectiveness of *Cerberus* by evaluating it on datasets of popular OAuth libraries with millions of downloads. Among these high-profile libraries, *Cerberus* has identified 47 vulnerabilities from ten classes of logical flaws, 24 of which were previously unknown. We got acknowledged by the developers of eight libraries and had three accepted CVEs.

KEYWORDS

Software Security, OAuth, Static Analysis, Dynamic Analysis

1 INTRODUCTION

OAuth is widely used for authorizations across different software services. It defines a process for end-users (resource owners) to grant a third-party website/application (also noted as relying party or client applications [21]) access to their private resources stored on a service provider, without sharing their passwords for the service providers with the client applications. As a multi-party protocol, the security of OAuth depends on the service providers (also noted as authorization server or *OAuth server* [21]), the client applications, and the resource owners. In particular, vulnerabilities in OAuth server implementations could lead to severe consequences because that would impact all the client applications (along with all the resource owners of the applications) that work with the service provider. For example, a vulnerability discovered in 2019 in the Microsoft OAuth server's redirect URI validation mechanism allowed the attackers to take over Microsoft Azure Accounts [4]. Also, in 2018, a vulnerability in Facebook OAuth server allowed the attackers to steal access tokens (issued by authorization server) of almost 50 million users [2].

Even well-known service providers such as Microsoft and Facebook still make severe mistakes in their authorization server implementation – which leads to a pressing question, *how to vet OAuth service provider implementation for security?* Since OAuth is very high-profile and critical for user security and privacy, many previous works have been on OAuth security. However, as far as we know, there is no automatic tool to identify the security-sensitive logical flaws in the service provider implementation.

Researchers have devoted significant effort to investigating OAuth security by analyzing the protocol [78] and proposing formal verification tools [53, 61]. However, these tools analyze the security at the protocol level and do not consider the diversified implementation details of OAuth server. Chen et al. [42] perform the first in-depth study on the security issues of OAuth by running a manual analysis of mobile apps and monitoring the network traffic. Later on, researchers develop semi-automated tools [36, 84] to report various security issues with OAuth client applications. However, these analyses often need an extensive manual setup and do not scale for large programs. Most recently, *S3kvetter* [80] performs symbolic execution, and *OAuthlint* [64] leverages whole-program data-flow analysis to check client-side properties. As we show later in the evaluation, these approaches are not applicable for checking OAuth server properties for the following reasons: first, symbolic execution suffers from path explosion [74] and whole-program analysis also does not scale well, especially for large codebase like OAuth server programs. Second, data-flow predicates alone are not expressive enough to describe crucial properties that require control-flow predicates.

In addition, most existing works assume that the OAuth servers are securely implemented, which clearly is not the case according to our study in this paper. As a result, it is urgent to help developers check the security of the OAuth server implementations. Unfortunately, as far as we know, there is no systematic security study of OAuth server implementations. Most previous works in OAuth security either focus on client applications or report security issues on an ad hoc basis.

We compare the representative OAuth tools from previous work using five relevant criteria: (1) *Automated:* ability to identify vulnerabilities automatically, (2) *Coverage:* the ability to cover and reason about the behavior of OAuth implementation, (3) *Server-side flaws:* the ability to analyze logical flaws of service provider implementations, (4) *Extensibility:* the ability to provide an interface to define and check new security properties, (5) *Scalability:* scalable for extensive programs. We summarize the comparison in Table 1, which clearly shows the gap.

To bridge the gap, we propose systematically checking the security of OAuth server implementations of authorization flows. We start by investigating popular open-source OAuth server libraries. We find many developers use these libraries to implement their service provider applications instead of starting from scratch due to their complexity. For example, Node OAuth2 Server [9] library is used by more than 2,400 repositories on Github. Since developers widely use these open-source libraries to integrate OAuth

in their implementations, checking the security of OAuth server implementations in these libraries also provides us insights about many implementations built on top of these libraries.

Challenges: However, developing an automatic tool for finding logical flaws of OAuth server implementations is quite challenging due to the following reasons:

- *Expressiveness*. The original specifications for OAuth protocols are written in plain English, which are difficult to be turned into checkable invariants consumed by the analyzers. Consequently, properties for implementing a secure and effective server-side implementation for authorization using OAuth are not well-defined.
- *Scalability*. Implementing the protocols on OAuth server is typically complex and large and often relies on many third-party libraries. Therefore, as shown in Sec. 6, a naive whole-program analysis will not scale well.
- *Generality*. Checking security properties in OAuth server implementations is not straightforward as they depend on various factors such as the client's platform and types of grants (i.e., authorization flows) used by the client during the authorization process. Thus, it is non-trivial to devise a framework that unifies all properties enforced by the OAuth specifications.

Our solution: To address the challenges mentioned above, we design and implement *Cerberus*, a scalable and automated static analysis tool for detecting logical flaws in large-scale OAuth server programs written in server-side languages like Java and Javascript. In particular, we first design a domain-specific language (DSL) to enable security analysts to express the security properties of OAuth server that are recommended by OAuth specification [21] and OAuth current security best practices [12].

Second, given an OAuth property Q encoded in our query language, we represent both the desired property Q and the OAuth server programs as *System Dependence Graphs* (SDGs). Therefore, *Cerberus* converts the problem of checking OAuth properties into a graph query problem, which an off-the-shelf Datalog engine [67] can answer. Note that our graph representation aims to strike a good balance between expressiveness and scalability by capturing temporal sequencing of API calls, data flows between arguments and returns of a procedure, data flows between various program objects, etc.

Table 1: Comparison with existing OAuth tools.

Tool	Auto.	Covg.	Serv.	Ext.	Scal.
AuthScan [36]	◐	◐	○	○	○
SSOScan [84]	●	◐	○	○	○
OAuthTester [81]	●	○	○	○	○
S3kVetter [80]	●	●	○	●	○
OAuthLint [64]	●	●	○	◐	○
Cerberus	●	●	●	●	●

● Full support ◐ Partial support ○ No support.
Auto.: Automated, Covg.: Coverage, Serv.: Server-side flaws, Ext.: Extensibility, Scal.: Scalability

However, there is a steep trade-off between the precision of an SDG and the cost of constructing it. For example, SDGs constructed using a context-insensitive pointer analysis tend to grossly overapproximate the targets of virtual method calls, which leads to unacceptable false alarms. On the other hand, more precise SDGs obtained using context-sensitive pointer analysis can take hours to construct. Currently, analyses that rely on system dependence graph information must implement their own ad hoc analysis [46, 56] to answer application-specific queries. To mitigate this challenge and only reason about program fragments relevant to the query, we introduce a query-driven approach based on automata theory. Since the original system dependence graph is obtained by stitching all control flow graphs from the methods, our key intuition is to keep track of the methods relevant to the OAuth property. In particular, we define the OAuth request endpoint, which is the precondition of the OAuth property, specifying the entry and exit points of the relevant code snippet. After that, we leverage the request endpoints to pinpoint a sub-callgraph between the request endpoints. Since the sub-callgraph is typically small, its corresponding system dependence graph will also be small.

Findings: We evaluate our tool with the ten most popular libraries that use the standard OAuth protocol for implementing OAuth server, and find pervasive vulnerabilities in these popular libraries. All the ten popular libraries we studied have at least one security-critical property violation (i.e., vulnerability). In total, we identify 47 vulnerabilities, 24 of which were previously unknown. We study ten security-critical logical flaws in OAuth server, five of which (P4, P5, P6, P7, and P9 in Table 2) are studied by us for the first time. Violation of these properties may lead to severe attacks (Appendix B) that are commonly observed on OAuth servers. We got acknowledged by the developers of eight libraries, among which five libraries, including Spring Auth Server [20] and Node OAuth2 Server [9]), immediately took actions to fix the vulnerabilities. Three classes of new vulnerabilities in these libraries also lead us to have new CVE entries (*CVE-2020-26877, CVE-2020-26937, and CVE-2020-26938*)[1]. These vulnerabilities can lead to account breaches and confidential information leakage for both clients and resource owners. Eight libraries immediately acknowledged our findings, five have fixed the issues after we reported them, and the rest of the libraries are currently taking action. Finally, *Cerberus* is effective in that it significantly outperforms *OAuthlint*, a state-of-the-art checker [64] in terms of running time (> 20× faster) and expressiveness.

Contributions. In summary, we make the following contributions:

- We identify and formalize the security properties required to implement the OAuth protocol on the service provider from the standard specifications and security best practices for OAuth.
- We, for the first time, design and implement an efficient automated tool, *Cerberus*, to find logical flaws in OAuth server implementation based on the standard specifications and security best practices, and we plan to make *Cerberus* open-source.

[1]The CVEs are currently marked as "Reserved" and will be published after we make the vulnerabilities public

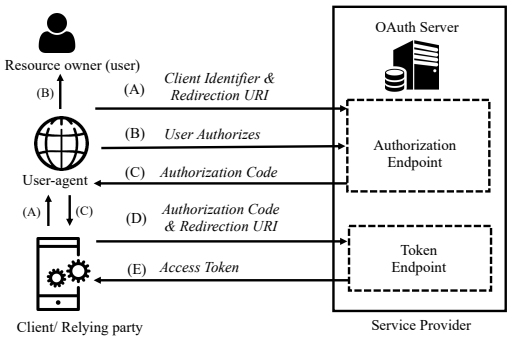

Figure 1: Authorization code grant flow for OAuth.

- Our analysis finds that many open-source libraries for OAuth server make critical security mistakes due to omitting or incorrectly implementing the security properties.
- We show that our query-driven scalable analysis is 25× faster in identifying the security property violations when compared with the eager analysis approach over the entire program.

2 BACKGROUND

This section describes the widely-used authorization protocol OAuth and the communication between different entities during the authorization process. In this paper, we focus on OAuth 2.0, which is the current industry-standard protocol for authorization, and throughout the paper, we refer to OAuth 2.0 when we say OAuth. OAuth 2.0 is an open standard authorization protocol where users delegate client applications (relying parties) to access their information hosted on the service providers without giving away their passwords. OAuth specification defines four types of grants–(1) authorization code grant, (2) implicit grant, (3) resource owner password credentials grant, (4) client credentials grant. Two grants are most widely used in practice – authorization code grant and implicit grant. In the following, we explain the details of these two grants.

Implicit Grant. Implicit grant is the simplest grant type. It has two steps. First, the user is redirected to the OAuth server to grant the relying party (client) access to their protected resources. After the user grants permission, the server redirects the user back to the relying party along with an access token. The relying party can then use this access token to request the user's protected resource from the OAuth server.

Authorization Code Grant. Authorization code grant can be used by both web apps and native apps to obtain the access token. The flow of authorization code grant is illustrated in Fig. 1. The authorization code grant augments the implicit grant by adding a step for authenticating the relying party (client). After user grants permission to the relying party, OAuth server redirects the user back to the relying party. Instead of providing the access token directly to the relying party, the server sends an authorization code this time. Then the relying party can use the authorization code to exchange for the access token by making a new request at the token endpoint. To get the access token, the relying party needs to include its identity in this request so that the server can verify if the authorization code is granted to the same party.

Although the authorization code grant provides better security benefits than the other grants, it is still vulnerable to code interception attacks, specially for public clients (e.g., native desktop apps), where the attackers intercept the authorization code returned from the authorization endpoint (step C in Fig. 1) and obtain the access token by exchanging the code at the token endpoint (step D). To mitigate the risk of code interception attack, OAuth specification introduces an extension of the authorization code grant called Proof Key for Code Exchange (PKCE) [19] and requires all OAuth servers to support PKCE for public clients. It is worth mentioning that, although PKCE was originally designed to protect public clients, it is recommended [12] to use PKCE for all kinds of clients, including web applications.

PKCE. Since public clients cannot maintain confidentiality and cannot securely store the client's secret, using PKCE allows OAuth server to authenticate clients without the secret key. PKCE utilizes a dynamically created cryptographic random key called *code verifier*. A unique *code verifier* is generated by the client for every authorization request. The transformed value of the *code verifier*, called *code challenge*, is sent to the OAuth server to obtain the authorization code. When a client makes a new request at the token endpoint to obtain the access token, it also sends the *code verifier* along with the authorization code it received from the previous request. To validate the proof of possession of the *code verifier* by the client, OAuth server transforms the *code verifier* and validates it with the previously received *code challenge*. This approach helps to mitigate the authorization code injection attack as an intercepted authorization code from the authorization endpoint cannot be exchanged for an access token without the one-time key of *code verifier*.

3 OVERVIEW

This section briefly explains how our tool detects vulnerabilities in OAuth server implementations using a motivating example. In what follows, we first describe the threat model our system, and then with a real-world example, we explain how insufficient security checks in the OAuth server allow malicious clients to steal sensitive OAuth credentials and how our tool is designed to detect such logical flaws.

Threat Model: We aim to detect vulnerabilities in the OAuth server at the implementation level. We assume attackers can be a malicious relying party or a malicious user (resource owner) who interacts with the victim OAuth server, also known as service provider. We assume the attackers cannot directly modify the source code or logic of the service provider but can initiate attacks by sending requests to their server. The relying party attackers control their own malicious relying party apps. For example, the relying party attackers might send malicious requests to the victim service provider to access the user's information without the user's approval. The resource owner attackers use their own devices to communicate with the benign service provider to log in on behalf of the victim user.

Vulnerable OAuth server implementation. The industry-standard authorization protocol of OAuth is well designed for access delegation, but a wrong implementation or incorrect usage can have a colossal impact. During the authorization process, client gets an access token with specific permissions to take actions on behalf of the user to whom the token belongs. Once the attacker gets this highly privileged access token, they can even control the user's account.

When responding to an authorization request from the client, the OAuth server passes the authorization code or tokens to the client application using a redirect URI, which describes the destination where the code or tokens are passed. The client application sets an allowed list of trusted URIs to receive the OAuth tokens during the registration process. However, while handling the authorization request, many OAuth servers do not appropriately validate redirect URI, leading to the possibility of passing the tokens to a malicious URI under the attackers' control. In recent years, many attacks exploiting the redirect URI have been observed in the OAuth servers, including the popular ones like Microsoft [4] and Twitter [23]. Similar attacks have been observed in the open-source OAuth servers, as they also utilize the same OAuth protocol.

For example, ApiFest [6] is a popular open-source OAuth server implementation that uses OAuth protocol to provide a secure API management service. We identify a new vulnerability of incorrect redirect URI validation in this library (CVE-2020-26877). Fig. 2 demonstrates a simplified implementation of their authorization request endpoint. ApiFest server makes mistakes in the critical step of checking the validity of the redirect URI submitted by the client, which allows an attacker to obtain the authorization code by using a maliciously crafted redirect URI during the authorization request. In particular, this implementation makes two severe security mistakes at this step (line 6-9 in Fig. 2): (1) it does not check whether the redirect URI submitted with the request is registered to the corresponding client, (2) it omits the required validation for the redirect URI format (i.e., being absolute URI or presence of any fragment component in the URI). These mistakes allow the attackers to steal the authorization code by leveraging two different attack vectors. First, as the server does not match the submitted redirect URI with the client's registered URI, it allows the attacker to craft the redirect URI parameter with attacker's own redirect URI and thereby, steal the authorization code of a legitimate client when user agent redirects the code to the redirect URI. Secondly, as the server does not perform any validation for the submitted redirect URI, the attacker can also leverage open redirectors of the user agent to steal the authorization code. These mistakes also violate the standard OAuth specification as described in RFC-6749 [21].

To detect such security vulnerabilities caused by the omitted or incorrect implementation of logical properties for OAuth, in this paper, we design and implement an automated and scalable tool, called *Cerberus*, to analyze the large codebase that implements the authorization process on OAuth server. We first identify the security-sensitive properties based on the standard OAuth specification [21] and security best practices [12, 13]. Then, we meticulously design a query language to formally express the properties so that developers can easily define them and they are understandable by the analysis tool as well. *Cerberus* then represents the OAuth server program at the statement level using system dependence graphs (SDGs), while maintaining the control and data flow relationship between the statements. However, as OAuth server programs can be huge, running analysis on the statement-level representation for the whole program does not scale well. To overcome the scalability challenge, *Cerberus* automatically identifies the program component corresponding to OAuth query. Finally, after *Cerberus* pin-points its scope in the OAuth relevant implementation in the

```
1  public void authRequest(HttpRequest request,
       HttpResponse response) {
2      String clientId = request.client_id;
3      Client client = getClient(clientId);
4      String redirectUri = request.redirect_uri;
5      [...]
6      //incorrect Redirect URI validation
7      if (redirectUri == null) {
8          redirectUri = client.redirect_uri;
9      }
10     [...]
11     AuthCode authCode = generateCode(clientId, scope,
           ...);
12     authCode.setClient(clientId);
13     authCode.setRedirectUri(redirectUri);
14     DB.storeAuthCode(authCode);
15     [...]
16     QueryStringEncoder encUri = new QueryStringEncoder
           (redirectUri);
17     encUri.addParam("code", authCode.getCode());
18     //redirects with auth code
19     response.sendRedirect(encUri.toString());
20 }
```

Figure 2: Code example for an authorization endpoint that is vulnerable to the redirect URI manipulation during the authorization flow.

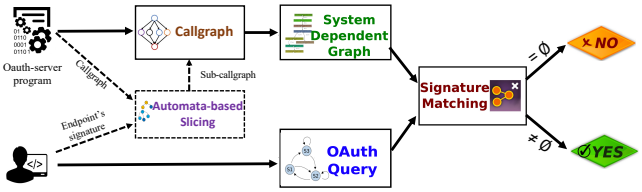

Figure 3: Overview of *Cerberus*. Here, Automata-based slicing is an optimization step discussed in Sec. 5.

program, it executes the query to identify the violation of properties that might expose the server to security attacks.

4 SYSTEM DESIGN

In this section, we discuss the design and implementation of *Cerberus*, our end-to-end static analysis tool for systematically checking the security issues of OAuth server implementation.

Fig. 3 shows an overview of the *Cerberus* approach for checking OAuth properties for Java or Javascript programs. *Cerberus* takes two inputs: (1) the source or byte code of an application, and (2) a user-provided OAuth property Q specifying the correct behavior using our query language. Given these two inputs, *Cerberus* performs signature matching by checking whether there exists an *embedding* of the application with respect to property Q.

4.1 Code Representation

Given a program, *Cerberus* first generates its abstract representation using static analysis. In particular, we leverage *system dependence graph* (SDG), which summarizes both data- and control-dependencies among all the statements and predicates in the program.

More formally, *System Dependency Graph* (SDG) for an application A is a graph (V, X, Y) where:

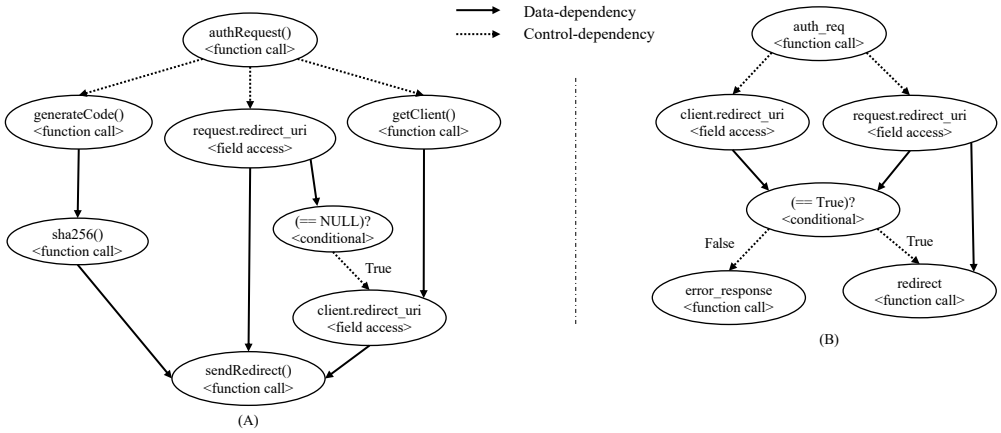

Figure 4: (A) System Dependence Graph (SDG) representation for the code example in Fig. 2 and (B) graphical representation of the query for Redirect URI property P1.

- V is a set of vertices, where each $v \in V$ is a program statement of A.
- X encodes control-dependency edges. Specifically, $(v, v') \in X$ indicates that during execution, v can directly affect whether v' is executed. Precisely, SDG creates three additional edges to handle function calls: (1) call edge, (2) parameter-in edge, and (3) parameter-out edge. Call edge connects the node at callsite (in caller) to the entry node of the called procedure (i.e., callee). Parameter-in edges connect the actual-in (caller) parameter nodes to the formal-in (callee) parameter nodes of the called procedure, and parameter-out edges connect the formal-out (callee) nodes to the actual-out (caller) nodes.
- Y is a set of data-dependency edges. In particular, $(v, v', d) \in Y$ indicates that statement v and v' are related by metadata d. Here, we use metadata d to denote *taint sources* that will be propagated by the data-flow analysis. Also, X is data dependent on Y if Y is an assignment and the value assigned in Y can be referenced from X.

Example 1. *Fig. 4(A) shows a simplified SDG constructed from the code example in Fig. 2. In the SDG, the nodes represent statements of the program such as function calls, field access, etc. The edges represent the control- and data-dependencies between the statements. For example, since the if condition (line 7) depends on the value of redirect_uri field (line 4), the conditional node (==Null?) has a data-dependency edge from the field access node (request.redirect_uri). On the other hand, control-dependency between the function call nodes, GetClient() and AuthRequest(), implies the GetClient() is invoked after the AuthRequest() is invoked.*

4.2 Facts and Inference Rules for SDG

Inspired by *OAuthlint*'s formalization [64], *Cerberus* converts the application's SDG into its corresponding facts and rules using Datalog. We first give some preliminaries on Datalog program, and then describe the syntax and semantics of *Cerberus*'s built-in predicates.

A Datalog [26] program is a set of *facts* and *rules* written in a declarative logic language. Facts correspond to predicates that evaluate to *true*. For example, in SDG's context, `edge(`$Stmt_1$`,` $Stmt_2$`)` implies that statement node $Stmt_1$ and $Stmt_2$ are connected by an edge. Each rule is a Horn clause [71] defining a predicate as a conjunction of predicates. For example, the following program:

```
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

says that `path(x, y)` is true if `edge(x, y)` is true, or both `path(x, z)` and `edge(z, y)` are true. In addition to variables, predicates can also contain constants (surrounded by double quotes), or "don't cares", denoted by '_'.

4.2.1 Base Facts in Cerberus. Unlike *OAuthlint* whose analysis is flow-insensitive, many crucial properties in OAuth server require a flow-sensitive analysis. Therefore, *Cerberus*'s facts take the form of $A(L, y, x_1, ..., x_n)$, where A is the instruction name, L is the instruction's label, y is the variable storing the instruction result (if any), and $x_1, ..., x_n$ are variables given to the instruction as arguments (if any). For example, the instruction $l_1 : r_1 = 0$ is encoded as `assign`$(l_1, r_1, 0)$. Additionally, `alloc`(l_1, y, x) means that variable x may point to abstract location y and `alias(x,y)` denotes that variable x and y may point to the same abstract location. Furthermore, the branch instruction `branch`(L_1, X, L_2, L_3) denotes that if X is evaluated to `true`, then the next instruction will be L_2, otherwise $L3$. Using the base facts described above, *Cerberus* computes the semantic facts of: (i) control-dependency predicates, which capture instruction dependencies according to the application's CFG, and (ii) data-dependency predicates.

4.2.2 SDG Predicates. Cerberus provides built-in predicates to encode SDG generated from Sec. 4.1. In particular, the `depOn`(y, x) predicate indicates that the value of variable x has data-dependence on y. Similarly, the `followBy` predicate is inferred from the application's CFG. Intuitively, `followBy`(L_1, L_2) holds for L_1 and L_2 if both are in the same basic block and L_2 follows L_1, or there is a path from the basic block of L_1 to the basic block of L_2. The SDG

predicates are computed using the following datalog rules:

$$\begin{aligned}
\mathsf{depOn}(x,y) &:- \mathsf{alloc}(_,y,x)\\
\mathsf{depOn}(x,y) &:- \mathsf{assign}(_,y,x)\\
\mathsf{depOn}(x,z) &:- \mathsf{assign}(_,y,x), \mathsf{depOn}(y,z)\\
\mathsf{depOn}(x,z) &:- \mathsf{alias}(y,z), \mathsf{depOn}(x,y)\\
\mathsf{followBy}(x,y) &:- \mathsf{follow}(x,y)\\
\mathsf{followBy}(x,z) &:- \mathsf{followBy}(y,z), \mathsf{follow}(x,y)
\end{aligned}$$

Here, we use the $\texttt{follow}(L_1, L_2)$ as the base case which holds if L_2 immediately follows L_1 in the CFG.

4.2.3 OAuth Predicates. In addition to basic facts from the SDG, *Cerberus*'s query language also defines a list of predicates specific to the OAuth domain. The predicate $\texttt{OAuthTag}(L, T)$ defines that, the value at label L is assigned tag T. Here, tag T are associated with program statements that hold OAuth-specific resources such as *redirect_uri*, *access_token*, etc. For instance, the field access at line 8 in Fig. 2 is denoted as $\texttt{OAuthTag}(8, \texttt{client\_redirect\_uri})$. Similar to [64], the `OAuthTag` predicate is computed through a standard data flow analysis in WALA. Specifically, our analysis first marks a set of APIs that could return OAuth resources as the sources. All variables assigned (either direct or transitive) by the sources will also point to the corresponding sources. In particular, since it is difficult to precisely pinpoint strings that correspond to redirected URLs, we use both pattern matching (i.e., regular expressions) and domain-specific knowledge (i.e., API that may return a redirected URI or object values accessed by the keys defined in the specification) to over-approximate the domain of URI. Furthermore, $\texttt{invoke}(L, m)$ predicate implies that method m is invoked at a location with label L. Here, m can be both OAuth-specific methods (e.g., `gen_token`) and programming language-specific methods (e.g., `matches`). $\texttt{sload}(L, q)$ predicate defines the load data operation for server's storage model q (e.g., database). Similarly, `sstore`/`sdelete` defines the store/delete operations. Finally, $\texttt{error}(L, e)$ implies that an OAuth exception is thrown with an error message e.

4.3 Query Language

In this section, we show how to express OAuth properties over semantics facts of an application. We begin by defining the query language for expressing security patterns. This construction enables us to determine whether an application complies with a given security property. Specifically, for each OAuth property, the user defines a unique predicate that serves as the signature for the property. The user may also define additional helper predicates used by the signature.

Syntax. The syntax of the query language is given by the following Backus–Naur form (BNF):

$$\begin{aligned}
\phi \quad ::= \quad & \mathsf{depOn}(v_1, v_2) \mid \mathsf{followBy}(v_1, v_2) \mid \mathsf{branch}(\ldots)\\
\mid \quad & \mathsf{OauthTag}(\mathsf{x}, \mathsf{y}) \mid \mathsf{invoke}(\mathsf{l}, \mathsf{m}) \mid \mathsf{sload}(\mathsf{l}, \mathsf{q}) \mid \ldots\\
\mid \quad & \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi
\end{aligned}$$

We now formally define our *property signatures* and state what it means for an app to *match* a property. Intuitively, a signature for a property $\mathcal{F}$ is an SDG (V_0, X_0, Y_0) that captures semantic properties. Ideally, $G_0 = (V_0, X_0, Y_0)$ would satisfy the following: G_0 occurs as a subgraph (defined below) of the SDG $G_S = (V_S, X_S, Y_S)$ of an OAuth application.

By "occurs as a subgraph", we mean there exists an embedding $F_S : V_0 \to V_S$ such that the following properties hold:

- **One-to-one.** For every $v, v' \in V_0$ where $v \neq v'$, F_S cannot map both v and v' to the same vertex, i.e., $F_S(v) \neq F_S(v')$.
- **Type preserving.** For every $v \in V_0$, F_S must map v to a vertex of the same type, i.e., $T(v) = T(F_S(v))$
- **Edge preserving.** For every $v, v' \in V_0$, F_S must map an edge $(v, v') \in X_0$ to an edge in X_S:

$$(v, v') \in X_0 \Rightarrow (F_S(v), F_S(v')) \in X_S.$$

Given property $G_0 = (V_0, X_0, Y_0)$ and app S with SDG $G_S = (V_S, X_S, Y_S)$, we say that G_0 *exactly matches* (or simply *matches*) S if G_0 occurs as a subgraph of G_S. In other words, given a signature (V_0, X_0, Y_0) and a sample S with SDG (V_S, X_S, Y_S), we can check whether (V_0, X_0, Y_0) matches S. If so, we have determined that $S \in \mathcal{F}$; otherwise, $S \notin \mathcal{F}$.

4.3.1 Expressing OAuth properties. Table 2 shows the list of properties that we express using our query language. We obtain these properties from the documentation of standard OAuth specifications [13, 19, 21] and security best practices [12]. In what follows, we briefly discuss the properties and their formal representation. Interested readers can find more details of these properties, including their security implications in Appendix A as well as corresponding attack examples in Appendix B.

OAuth requires that the redirect URI submitted during the authorization request match the client's registered URI (P1). Specifically, we tag these OAuth parameters by $\texttt{OAuthTag}(\_, \texttt{req\_URI})$ and $\texttt{OAuthTag}(\_, \texttt{client\_URI})$, respectively. In addition, since this URI is used to transfer sensitive information like authorization code, the specification also requires the URI to be an *absolute URI* (P2), preventing attackers from utilizing the open redirector of the user agent to intercept the sensitive information. Server programs commonly use pattern matching APIs (e.g., '`matches`') to match the redirect URI value.

To prevent code replay attacks during token request, OAuth also requires the authorization code to be single-use (P3), meaning the code can be used only once (to generate token) by the client. Therefore, once the code is used, OAuth server removes it from the storage (e.g., database) and rejects requests (with error message) when the code is not found in storage. We use $\texttt{OAuthTag}(\_, \texttt{code})$ to tag authorization code and $\texttt{invoke}(\_, \texttt{gen\_token})$ to track the OAuth-specific API call for generating token.

OAuth security best practices require OAuth server supporting PKCE [19] to protect public clients (e.g., native desktop apps). To securely implement PKCE, servers first receive and store the PKCE parameters (e.g., `code_challenge`) at authorization endpoint (P6), and transforms (by '`sha256`') the `code_verifier` parameter to match with the `code_challenge` (P7) at the token endpoint.

OAuth also requires access token to be constrained to a certain client to prevent token injection attacks (P9). Mutual-TLS [11] (also noted as m-TLS) is a standardized and widely used client-constrained mechanism that allows the clients to demonstrate the proof of possession when using the access token. In m-TLS, the server first obtains the client's certificate from TLS stack and associates it with the token before sending to the clients. The certificate is decoded and encoded using standard Base64 decode-encode APIs. In our formalization, we use $\texttt{OAuthTag}(\_, \texttt{client\_cert})$ to tag

Prop.	Grants	Description	*Cerberus* Query
P1	AuthCode, Implicit	OAuth server must validate the redirect URI parameter in the authorization request exactly matches with the client's registered URI before sending the redirected response to the URI. (RFC-6749)	`invoke(L1, auth_req),OAuthTag(L2, req_URI), OAuthTag(L3,client_URI), invoke(L4,redirect), error(L5, _), branch(L6,X,L4,L5), followBy(L1,L2), followBy(L1,L3), depOn(L2,X), depOn(L3,X).`
P2	AuthCode, Implicit	Redirect URI parameter in authorization request must be an *absolute* URI. OAuth server implementations utilize pattern matching APIs to perform this validation. (RFC-6749, RFC-6819).	`OAuthTag(L1, auth_req), OAuthTag(L2, req_URI), OAuthTag(L3,abs_URI), invoke(L4,matches), invoke(L5,redirect), error(L6,_), branch(L7,X,L5,L6), followBy(L1,L2), depOn(L2,L4), depOn(L3,L4), depOn(L4,X).`
P3	AuthCode	Authorization code must be single-use, meaning the code must be revoked once it is exchanged for token. If a revoked code is used again, OAuth server must deny the request with an error message. (RFC-6749).	`invoke(L1, token_req), OAuthTag(L2, code), sload(L3, db),invoke(L4, gen_token), sdelete(L5, db), error(L6, _), branch(L7,X,L4,L6), followBy(L1,L2), depOn(L2,L3), depOn(L3,X), followBy(L4,L5), depOn(L2,L5).`
P4	AuthCode	Authorization code must be bound to a certain client. Before issuing the token from the token request endpoint, OAuth server must check the code is the same as the one issued to the client. (RFC-6749, OAuth Security Best Practices).	`invoke(L1, token_req), OAuthTag(L2, code), OAuthTag(L3,client_code), invoke(L4, gen_token),error(L5, _), branch(L6,X,L4,L5), followBy(L1,L2),followBy(L1,L3), depOn(L2,X), depOn(L3,X).`
P5	AuthCode	Authorization code must be bound to the client's redirect URI to where it was issued to. Before issuing the token, the server must check that the code is associated with the client's redirect URI. (RFC-6749).	`invoke(L1, token_req), OAuthTag(L2, code_URI), OAuthTag(L3, client_URI), invoke(L4,gen_token), error(L5, _), branch(L6,X,L4,L5), followBy(L1,L2), followBy(L1,L3), depOn(L2,X), depOn(L3,X).`
P6	AuthCode	OAuth server must store the PKCE parameters (i.e., code challenge and code challenge method) at the authorization endpoint to be validated at token endpoint (RFC-7636, OAuth Security Best Practices).	`invoke(L1, auth_req), OAuthTag(L2, code_challenge), OAuthTag(L3, code_challenge_method), sstore(L4,db), followBy(L1,L2), followBy(L1,L3), depOn(L2,L4), depOn(L3,L4).`
P7	AuthCode	OAuth server must verify the authenticity of PKCE parameters at token endpoint. The transformed (i.e., sha256) value of `code_verifier` parameter must be same as the `code_challenge` value (RFC-7636, OAuth Security Best Practices).	`invoke(L1, token_req),OAuthTag(L2, code_challenge), OAuthTag(L3, code_verifier), invoke(L4, sha256), invoke(L5, gen_token), error(L6, _), branch(L7,X,L5,L6), followBy(L1,L2), followBy(L1,L3), depOn(L3,L4), depOn(L4,X), depOn(L2,X).`
P8	AuthCode, Implicit	OAuth server must provide CSRF protection by handling the `state` parameter in authorization request. Value of the `state` parameter must be added to the redirected response of authorization request. (RFC-6749, OAuth Security Best Practices).	`invoke(L1, auth_req), OAuthTag(L2, state), invoke(L3, redirect), error(L4,_), branch(L5,X,L3,L4), followBy(L1,L2), depOn(L2,X), depOn(L2,L3).`
P9	AuthCode, Implicit	Access tokens issued by the OAuth server should be constrained to a certain client. *m-TLS* is a standardized and widely used client-constrained mechanism, in which the server first obtains the client's certificate from TLS stack, decodes and hashes the certificate and finally, associates it with the access token. (RFC-8705, OAuth Security Best Practices).	`OAuthTag(L1, access_token), OAuthTag(L2, client_cert), invoke(L3, b64_decode), invoke(L4, b64_encode), invoke(L5, sha256), invoke(L6, add_cert), depOn(L2,L3),depOn(L3,L5),depOn(L5,L4), depOn(L1,L6),depOn(L4,L6).`
P10	AuthCode, Implicit	OAuth server should not store access tokens as clear-text and should store access token hashes only (RFC-6819).	`OAuthTag(L1, access_token), invoke(L2, sha256), sstore(L3,db), depOn(L1,L2), depOn(L2,L3).`

Table 2: Based on the standard OAuth specifications (RFC-6749 [21], RFC-6819 [13], RFC-7636 [19]) and security best practices [12], we define ten security properties to detect the most commonly observed logical flaws on OAuth servers. WE define the *Cerberus* query expressed using the predicates described in Sec. 4.2 for each property. For OAuth server, five (P4, P5, P6, P7, and P9) of these security properties are studied by us for the first time. Due to the space limitation, we provide more details including the security implications of these properties in Appendix A and corresponding attack examples in Appendix B

statements holding client's certificate, and `invoke(L4, b64_encode)`, `invoke(L4, b64_encode)`, etc. to track the corresponding domain-specific APIs.

Once a query expressing a property from the OAuth specification is submitted, *Cerberus* checks the it against the SDG representation of the input program.

Example 2. *Fig. 4(B) shows a graphical representation of the query that* over-approximates *the redirect URI property (P1) in Table 2. It represents a signature where a redirect URI received during an authorization request (request.redirect_uri) is matched with the redirect URI field of a client instance (client.redirect_uri) before making a redirect function call, which must be used for a successful response to the authorization request. Therefore, this signature can be used to check that a successful redirection occurs only after the check for redirect URIs is performed. Otherwise, an error response is generated. On the other hand, the SDG (Fig. 4(A)) of the program for processing authorization request retrieves the 'redirect_uri' from the 'request' instance, but the 'redirect_uri' is matched only against the NULL value before sending the redirection using 'sendRedirect' function call. However, OAuth specification requires that the server must check whether the 'redirect_uri' of the authorization request is the same as the 'redirect_uri' registered to the client, which is not done in this program. Therefore, Cerberus reports a violation.*

5 IMPLEMENTATION

This section discusses the design and implementation of *Cerberus*, as well as a few key optimizations.

5.1 Query-driven Exploration

Cerberus leverages System Dependence Graph (SDG) that captures program's control- and data-flow dependencies. However, there is a steep trade-off between the precision of an SDG and the cost of constructing it. For example, SDGs constructed using a context-insensitive pointer analysis tend to grossly overapproximate the targets of virtual method calls, leading to unacceptable false alarms. On the other hand, more precise SDGs obtained using context-sensitive pointer analysis can take hours to construct. Currently, analyses that rely on system dependence graph information must implement their own ad hoc analysis [46, 56] to answer application-specific queries. To mitigate this challenge, we introduce a query-driven approach whose key insight is to only reason about small program fragments relevant to the OAuth property.

Since the original system dependence graph is obtained by stitching all control flow graphs from the methods, our key intuition is to keep track of the methods relevant to the OAuth property. In particular, we define OAuth request endpoint Q, which is the precondition of the actual OAuth property specifying the entry and exit points of the relevant code snippet. After that, we leverage the request endpoints to pinpoint a sub-callgraph between the request endpoints. Since the sub-callgraph is typically small, its corresponding system dependence graph will also be small.

Given an endpoint regex as input, we implement a lightweight *program slicing* using automata theory [34]. Specifically, *Cerberus* first constructs the so-called *query automaton (QA)* [51] for the request endpoints and the *callgraph automaton (CGA)*. Here, the query automaton is simply an NFA-representation of the regular expression specified by the user. Since the problem of converting regular expressions to finite state machines is well-studied [37], we do not explain the QA construction in detail here.

We now explain the syntax and semantics of its query language for specifying request endpoints. For a given program P, *Cerberus* accepts specifications written in the following query language:

$$\begin{aligned} \textit{Query}\, Q \quad := \quad & f \in \text{methods}(P) \\ & \mid \ . \ \mid Q_1 \rightarrow Q_2 \\ & \mid Q_1 + Q_2 \mid Q^* \mid Q^+ \mid (Q) \end{aligned}$$

The building blocks of queries are method names in program P, denoted by methods(P). The dot character (".") matches any method name, and the $\rightarrow$ operator indicates a call from one method to another. The "+" operator is used for taking the disjunction of two queries. As usual, the "$*$" operator stands for Kleene closure, and Q^+ is syntactic sugar for $Q \rightarrow Q^*$.

The callgraph automaton CGA for a given application P with respect to a callgraph C is a finite state machine where states include all methods of P and transition functions correspond to call edges (i.e., function calls.). After constructing the query and callgraph automata, the next step is to compute the *intersection* of those two using the JSA [44] tool. The output is a *product automata* that encode a relevant program slice with respect to the query.

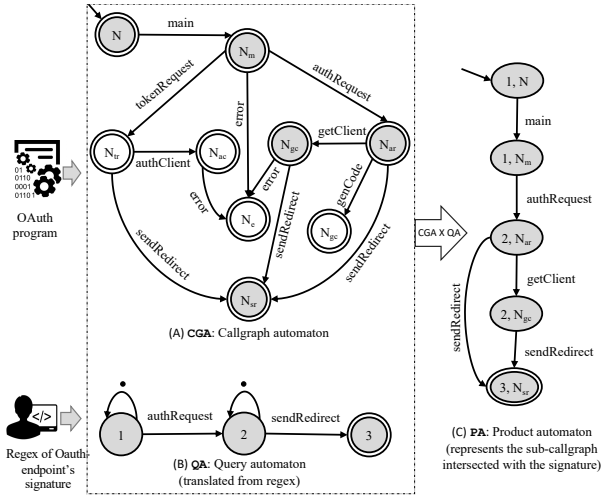

Figure 5: An illustration of our query-based slicing technique where (A) A partial callgraph automaton (CGA) constructed for the OAuth program in Fig. 2. (B) an example of query automaton (QA), which we use to represent OAuth endpoint's signature, and (C) a partial product automaton constructed by *Cerberus*.

Example 3. *Fig. 5 (A) shows a partial call graph Automaton (CGA) constructed from the OAuth program in Fig. 2. Since the original program may lead to a large call graph that is difficult for a whole-program analysis, the user can optionally obtain a relevant program slice by specifying a query "(.* → `authRequest` → .* → `sendRedirect`)", which is a regular expression representing the entry and exit point of OAuth endpoints. Here, our goal is to extract the intersection between the callgraph and the user-specified query. Fig. 5 (B) shows a query automaton (QA) that is obtained from the previous regular expression through a standard algorithm based on induction [34]. Having constructed the query and callgraph automata, the next step is to determine whether the intersection of the two is empty. Fig. 5 (C) shows a partial product automaton (PA) constructed by the CGA (A) and QA (B), i.e., $L(PA) = L(CGA) \cap L(QA)$ where $L(PA)$ represents the langauge accepted by product automaton PA, which is again can be computed by a standard algorithm from [34]. Here, every state that would appear in the product automaton must also appear in both the call graph automaton and query automaton. Therefore, the resulting PA represents a* program slice *that is relevant to the query.*

We note that, compared to a naive whole-program analysis, our query-driven exploration does not miss a property violation as long as all the necessary endpoints are provided. Since the endpoints' signature defining the entry and exit node is the precondition for the each property, *Cerberus* will not prune any paths that might violate the corresponding property.

5.2 Hybrid Analysis for Dynamic Features

The system dependence graph may not capture the full semantics of some OAuth properties. For instance, one CVE entry (*CVE-2020-26938*), identified by *Cerberus*, is caused by checking if the redirect

URI contains an absolute URI using an incorrect URI pattern *"[a-zA-Z][a-zA-Z0-9+.-]+:"*. This may lead to incorrect results since our current abstraction does not reason about the semantics of regular expressions. Also, since our tool is based on static analysis, it may not determine whether a branch condition evaluates to true or not. However, a fully dynamic analysis will be prohibitive since we have to deal with libraries with a large codebase.

To mitigate the above-mentioned challenge, *Cerberus* incorporates a hybrid approach: in particular, given an application that may potentially contain dynamic features or regular expressions that go beyond the scope of our current static analysis, we first make the most conservative assumption by assigning relevant predicates to `false`. For instance, the `branch` predicate will evaluate to false if its condition contains regular expressions, which will fail the signature matching procedure and raise a potential *false alarm*. After that, we perform a light-weight *delta testing* as follows: for each predicate that is assigned to `false` due to our conservative assumption, we dynamically exercise the relevant code to recover the missing facts. For example, for the case with regular expressions, given a set of input strings, we will test whether the actual regular expression's output is the same as the ones generated by the correct regular expression. If so, we turn its corresponding predicate to `true` and rerun *Cerberus*. We iterate this process until *Cerberus* confirms a violation or all false alarms are eliminated. Our current implementation can handle cases of dynamic features (e.g., reflective calls in Java) where it takes arguments with string constants. In such cases, we leverage the current data-flow analysis to keep track of the strings that may be used as class or function names.

5.3 The *Cerberus* Tool

We implemented our core static analysis on top of the WALA framework [22], which provides compilation and analysis infrastructure for both Java and Javascript. *Cerberus*'s implementation consists of approximately 9,860 lines of Java code. We use an Andersen-style pointer analysis [35] and the CHA (Class Hierarchy Analysis) callgraph algorithm provided by WALA. An OAuth-endpoint expressed in regular expression, is converted into its query automata using the JSA library [44].

From source code to Datalog programs. Given a service provider program P written in Java or Javascript, *Cerberus* first leverages the WALA framework to generate its corresponding system dependence graph (SDG) from bytecode (for Java) or scripts (for Javascript). As shown in Figure 4, each node of the SDG corresponds to a statement using WALA's intermediate representation in Static Single Assignment (SSA) form. Each statement will be translated into its Datalog fact for allocation, assignment, function call, etc. Second, each edge is translated into it corresponding predicate in Datalog. For instance, the control- and data-dependence edges are translated into their corresponding `followBy` and `depOn` predicates discussed in Section 4.2, respectively. Finally, *Cerberus* leverages the Soufflé [67] Datalog solver for checking conformance between the SDG and the OAuth properties.

Resolving *Node.js* modules. For Javascript, the callgraph is constructed directly from the source code (i.e., scripts). Therefore, function calls from an included module from *Node.js* framework is not automatically resolved as the required source files are not known to the analysis. For example, to implement a HTTP server in *Node.js* framework, developers may call *http.createServer()* from the HTTP module by using *require('http')*. Therefore, to resolve the call *createServer()*, we first need to identify the required source file to be included in the analysis. We use the pointer analysis to identify the strings that can flow to a *require* call. Then the corresponding file is loaded in the analysis, and the target method is included in the callgraph.

6 EVALUATION

To determine the effectiveness of *Cerberus*, we evaluate it on popular OAuth-server libraries to answer the following research questions:

- **RQ1**: Can *Cerberus* identify real-world vulnerabilities?
- **RQ2:** Is our query-driven approach efficient and effective?

6.1 Experimental Setup

We conduct all experiments on a Quad-Core Intel Core i5 computer and 16G of memory running on the macOS 10.15 operating system. In what follows, we elaborate on the details of the setup.

6.1.1 Dataset. We consider high-profile OAuth-server libraries written in Java or Javascript. With extra engineering effort, our techniques can technically be applied to applications in other languages as well. To answer the research questions, we consolidate two datasets of open-source OAuth server libraries that implement the standard OAuth specification [21].

- $\texttt{DataSet}_U$: This dataset contains 10 popular open-source libraries of OAuth server (i.e., service provider) where it is *unknown* whether the libraries satisfy the security properties. Table 3 shows the key statistics for the selected libraries. To select the libraries, we first consider their popularity among the web developers. For example, *Node Oauth2 Server* [9] library has more than 200k downloads each month. Similarly, *Oauth2orize* [17] and *Node oidc provider* [10] have approximately 179k and 65k downloads each month. Additionally, we also consider the number of dependant repositories (i.e., repositories that use the APIs of the library) as an indicator of the popularity of our selected libraries. All of the libraries support the widely used authorization code grant [21], and three also support the implicit grant. Four of these libraries are implemented in Javascript, and six are implemented in Java. Some libraries (e.g., *OxAuth*) are considerably larger than others as they also support additional endpoints (e.g., token introspection) and platform-specific custom grants for their authorization server.
- $\texttt{DataSet}_K$: This dataset contains 12 OAuth libraries with 49 OAuth-specific logical flaws that are confirmed by Github issues or online forums. Among the 49 logical flaws, 13 were caused for improper handling of authorization code, 12 for mishandling access token, 7 for redirect URI validation, 8 for missing or incorrect PKCE validation, 3 for state parameter, and 6 for other issues such as incorrect client validation. While the selection criteria for these libraries are similar to $\texttt{DataSet}_U$, we exclude libraries whose ground truths are unknown. Six of these libraries are implemented in Javascript, and six are implemented in Java. All libraries follow the standard OAuth specifications and support all the commonly used grants.

OAuth-server libraries	Version	Language	#stars (github)	Supported grants	#lines of code
1) Node oauth2 server [9]	3.1.1	JS	3,180	Auth code	3,936
2) Oauth2orize [17]	1.11.0	JS	3,119	Auth code, Implicit	3,483
3) Node oidc provider [10]	6.29.5	JS	1,291	Auth code, Implicit	15,493
4) Oauth2 server [16]	1.0	JS	105	Auth code	2,946
5) Spring auth server [20]	1.0	Java	2,978	Auth code	28,374
6) Clouway server [7]	1.0.6	Java	37	Auth code	8,462
7) Jobmission server [14]	1.0	Java	321	Auth code	6,480
8) Apifest [3]	0.3.1	Java	67	Auth code	14,371
9) Yoichiro server [15]	1.0	Java	96	Auth code	6,764
10) OxAuth [8]	3.0.2	Java	276	Auth code, Implicit	62,857

Table 3: Statistics for the open-source OAuth authorization server libraries in $\texttt{DataSet}_U$.

6.1.2 State-of-the-arts. Existing tools focus on OAuth properties by analyzing the flows observed from client-side applications while treating the server-side as blackbox. Therefore, there is no prior work that directly performs whitebox analysis on OAuth server programs. *OAuthlint* [64] is the closest to our work as they also statically analyze source code of client applications to check OAuth properties. Unlike *Cerberus*, *OAuthlint* only supports OAuth properties expressed via data-flow predicates. Furthermore, *OAuthlint* performs whole-program analysis over the client application. *S3kvetter* [80] leverages symbolic execution to check security properties in OAuth SDKs, which provides APIs for implementing client applications. Similar to *OAuthlint*, *S3kvetter* also consider server-side implementation as blackbox. In fact, *S3kvetter* implements it's own *model* of the server to analyze the client-side flows for all of the SDKs. Its properties are defined in terms of the request-response behavior observed from client apps. Therefore, we cannot compare against *S3kvetter* because of unsupported properties (e.g., checking expressions, API calls, etc.) and unsupported languages (e.g., *S3kvetter* only supports Python).

6.1.3 User inputs. *Cerberus* takes as inputs a target program P and a property Q. For each benchmark, we check it against all properties defined in Table 2. To quantify the manual effort of writing the queries before running the evaluation, we recruited 18 independent students from a graduate-level security class – who were provided with the OAuth properties (i.e., from specification written in English) and documentation for our query language. We provide five randomly selected properties to each participant and measure the time they spent expressing the properties in our query language. We found participants spent on average 10.6 minutes to specify the given properties using our query language.

To speed up the analysis, *Cerberus* provides built-in OAuth endpoints (as defined in the specification) to leverage the query-driven slicing method (Sec. 5.1). Meanwhile, we allow users to provide (as query) their own custom endpoints in regex format to check any additional endpoints that is not covered by the standard specification. Thanks to the library documentation, these endpoints' format are clearly specified for most libraries and can be easily defined using regex for our tool. This flexibility also allows the analysts to use *Cerberus* to efficiently check security properties in extended or custom endpoints (e.g., token introspection) as well as new OAuth extensions (e.g., DPoP token [31]).

6.2 Evaluation Results

6.2.1 Discovered Vulnerabilities. To answer **RQ1**, we use *Cerberus* to identify logical flaws from both known and unknown datasets.

Detection of known vulnerabilities. We first evaluate *Cerberus* using $\texttt{DataSet}_K$ which consists of 12 popular OAuth libraries with 49 known flaws. As shown in Table 4, using the properties from Table 2, *Cerberus* identifies 43 out of 49 logical flaws.

Detection of unknown vulnerabilities. We next evaluate *Cerberus* using $\texttt{DataSet}_U$ with ten popular open-source libraries of OAuth service providers.

Ground truth determination.

OAuth-server libraries	#Known logical flaws	#Flaws identified by *Cerberus*	#FN
1) OxAuth[8]	9	8	1
2) Mitre Server[28]	7	6	1
3) Spring Auth Server[20]	6	6	0
4) Node OAuth2 Server[9]	6	4	2
5) Loopback OAuth[29]	4	4	0
6) OAuth Provider[30]	4	3	1
7) Egg OAuth2 Server[27]	4	4	0
8) ApiFest OAuth[3]	1	1	0
9) Clouway Server[7]	3	3	0
10) OAuth2orize[17]	3	2	1
11) Java OAuth Server[24]	1	1	0
12) Connect OAuth2[25]	1	1	0
Total	**49**	**43**	**6**

Table 4: Evaluation results of *Cerberus* on $\texttt{DataSet}_K$.

To determine the ground truth for $\texttt{DataSet}_U$, we perform both dynamic analysis and source code inspection with the OAuth flows simulated for each library. We deploy each library on a local server and implement client-side programs to simulate the OAuth flows according to the specification. In particular, for each property, we manually construct the parameters to initiate the corresponding OAuth requests and analyze the responses from the server. For example, to verify the result reported by *Cerberus* for redirect URI property (P1), we first deploy the library on a local server and create an authorization server instance. We create two client instances– one with a benign redirect URI and another with a malicious one. We initiate an authorization request from the benign client, but replace the '*redirect_uri*' parameter with the redirect URI from the malicious client. We observe the traces generated by the server. If the server redirects the authorization response (with *authorization code*) to the malicious URI, we mark a violation to the redirect URI property. Even if *Cerberus* reports no property violation, we still investigate the library, in the same way to find if the library actually violates the property.

As presented in Table 5, we discover 47 confirmed vulnerabilities in the ten popular libraries of the authorization server, 24 of which were previously unknown. Security impacts of these vulnerabilities can cause severe attacks (Appendix B) on the authorization server. We discuss these vulnerabilities and their implications below:

OAuth-server libraries	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	#Violation	#FP	#FN
1) Node oauth2 server	✓	×	✓	✓	✓	×	×	✓*	×	×	5	0	1
2) Oauth2orize	✓	×	×	✓	✓	×	×	×*	×	×	6	1	0
3) Node oidc provider	✓	×	✓	✓	✓*	✓	✓	✓	✓*	✓	1	0	2
4) Oauth2 server	×	×	✓	×*	✓	×	×	×	✓	✓	5	1	0
5) Spring auth server	✓	×	×	✓	✓	×	×	✓	✓	✓	4	0	0
6) Clouway server	✓	×	×	×*	✓	✓	✓	✓	×	×	4	1	0
7) Jobmission server	×*	×	×	×	×	×	×	×*	×	✓	7	2	0
8) Apifest	×	✓	✓	✓	×*	×	×	✓	×	×	5	1	0
9) Yoichiro server	×*	×	×	×	×	×	×	×	×	✓	8	1	0
10) OxAuth	✓	×	✓	✓	×	✓	✓	✓	✓	✓	2	0	0
Total vulnerabilities:	2	9	6	2	3	7	7	2	5	4	47	7	3

Table 5: Evaluation results of *Cerberus* on $\texttt{DataSet}_U$ whose vulnerabilities are unknown. (✓: library satisfies the property, ×: library violates the property). Here, ×* indicates a false positive (FP), where library satisfies the desired property, but *Cerberus* marks it as a violation due to the limitation of the underlying static analysis. On the other hand, ✓* indicates false negative (FN), which could occur due to imprecisely resolved call-sites.

Redirect URI manipulation (violates P1 or P2). *Cerberus* successfully identifies two libraries violating the property that the redirect URI from authorization request must match the URI registered by the client (P1). In addition, *Cerberus* finds nine libraries violating the property that the redirect URI must be an *absolute* URI (P2). For example, to check if the redirect URI parameter contains an absolute URI, the Node oauth2 server library matches it with an incorrect URI pattern "[a-zA-Z][a-zA-Z0-9+.-]+:". Violation of these properties allows the attackers to manipulate the redirect URI while initiating the authorization request and obtain a legitimate client's authorization code or access token directly from the authorization server. We describe this attack with an example in Appendix B. Our finding indicates that developers often do not follow the requirement for redirect URI, possibly because it may not seem security-critical from reading the specification.

Authorization code injection (violates P3, P4, P5, P6, or P7). *Cerberus* successfully identifies six libraries violating at least one property required to protect the clients against the authorization code injection attack. We find six libraries omitting or incorrectly implementing the property that the authorization code must be single-use (P3). This vulnerability allows the attackers to replay the authorization code intercepted from a legitimate client's request and obtain the access token. We also identify two libraries that do not bind the authorization code to any particular client (P4), allowing the attackers to exchange the code using a malicious client application. Additionally, *Cerberus* successfully identifies four libraries that do not bind the authorization code to the redirect URI (P5). This vulnerability prevents the authorization server from verifying that the redirect URI submitted during the token request is the same as the one where the authorization code was issued. Each of these vulnerabilities leaves an open door for the attackers to perform an authorization code injection attack that eventually allows the attackers to obtain a victim resource owner's information, which we describe with an attack example in Appendix B. *Cerberus* also finds seven libraries that do not support PKCE (P6 & P7)–a recommended mechanism designed for the authorization server to mitigate authorization code injection attacks for public clients.

Access token injection (violates P9 or P10). *Cerberus* successfully identifies five libraries violating the properties required to protect against the access token injection attack. All five libraries do not issue client-constrained access tokens from the token endpoint (P9). A client-constrained access token limits the applicability of the token to a particular client. It prevents the attackers from using an access token that is obtained through a malicious client application. In addition, *Cerberus* also identifies four libraries that store the access token in clear text format (P10). Storing access tokens as clear-text is vulnerable since it can expose the tokens to the attackers through accessing the local storage or launching a SQL injection attack.

Vulnerable to CSRF (violates P8). *Cerberus* successfully identifies two libraries that do not provide CSRF protection using OAuth's *'state'* parameter (P8). By design, libraries supporting the PKCE are CSRF-protected. Therefore, libraries without PKCE must handle the authorization request's *'state'* parameter to protect their clients against CSRF. *Cerberus* reports a library as vulnerable if it neither supports PKCE, nor handles *'state'* parameter.

Property	***OAuthlint*** (× slicing) Exec. time (min)		***Cerberus*** (✓ slicing) Exec. time (min)	
	$\texttt{DataSet}_U$	$\texttt{DataSet}_K$	$\texttt{DataSet}_U$	$\texttt{DataSet}_K$
P6	48.3	64.8	**2.1**	**3**
P9	55.8	71.6	**2.5**	**3.7**
P10	34	52.9	**1.7**	**2.2**

Table 6: Comparing *Cerberus* against *OAuthlint* in terms of average running time using the $\texttt{DataSet}_U$ and $\texttt{DataSet}_K$.

6.2.2 Analysis Accuracy. *Cerberus* successfully identifies 47 property violations (Table 5) and reports 7 false-positive (FP) cases and 3 false-negative (FN) cases in $\texttt{DataSet}_U$, and 6 FN cases (Table 4) in $\texttt{DataSet}_K$. Our assessment finds the primary reason for the FP and FN cases is the spurious points-to targets from the WALA framework. Additionally, unsoundness can also occur for a few cases during Javascript library analysis. We summarize the reasons for the FP and FN cases in the following:

- Semantics of complex string operations (e.g., `subString` or `concate`) cannot be precisely modeled by WALA–which leads our analysis to be unable to keep track of the precise values held by some string variables.
- Semantics of functions for executing dynamically generated code (e.g., Javascript's `eval`) are not currently modeled as we do not reason dynamic code in this work.
- For reflected calls where a reference to a target method is computed at runtime (e.g., Java's `invoke`), our static analysis may not accurately resolve the target method, which can result in FNs.
- Javascript libraries with dynamic features (e.g., dynamic module loading) may cause WALA to generate a broken and imprecise callgraph that leads to FPs and FNs.

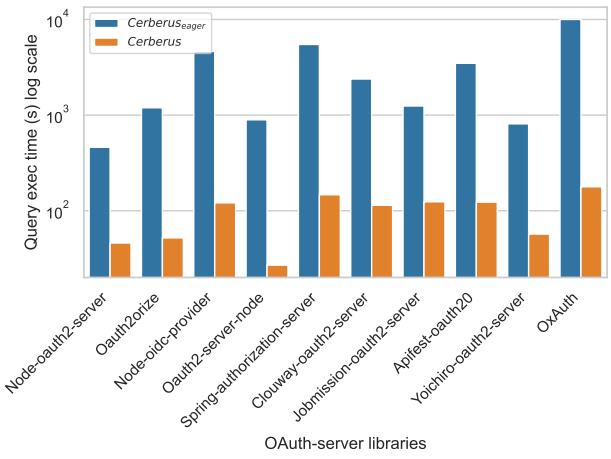

Figure 6: Performance comparison for libraries in $\texttt{DataSet}_U$ based on average query execution time (in log scale) between *Cerberus*, which constructs the on-demand callgraph , and *Cerberus*$_{eager}$, which constructs the callgraph eagerly for the whole program. Bar(s) touching the maximum time (10^4) indicate the library not terminating or causing memory explosion within the maximum time limit (3 hours).

6.2.3 Acknowledgments from developers. We reported all the unknown vulnerabilities to the developers of the libraries. So far, we have received confirmation from the developers of eight libraries, who have acknowledged our findings. By the time of writing this paper, five libraries had fixed the redirect URI, PKCE, and authorization code issues as we reported; the other remaining issues are undergoing a review process. Developers of the two libraries responded that they are taking action on our reports, and the issues will be addressed in their next release. Developer of one library confirmed the issues we reported, but declined to take action as they no longer work for the corresponding vendor. Three classes of unknown vulnerabilities in these libraries led us to have three new CVE entries (CVE-2020-26877, CVE-2020-26937, and CVE-2020-26938). However, we have not yet received feedback from two libraries. We are making our best effort to reach the developers through different channels and help them fix the vulnerabilities.

Result for RQ1: We identified 47 vulnerabilities in ten popular OAuth libraries, including developers' acknowledgment from eight libraries and three accepted CVE entries.

6.2.4 Performance Evaluation.

To answer **RQ2** and evaluate the effectiveness of our query-driven algorithm discussed in Sec. 4.1, we compare *Cerberus* with its variant, which constructs the callgraph for the entire program eagerly (noted as *Cerberus*$_{eager}$). Unlike *Cerberus*, which constructs an on-demand callgraph based on the OAuth query and prunes away any program components that are irrelevant with respect to the query, *Cerberus*$_{eager}$ does not consider any relevance with the query. Instead, it constructs the callgraph eagerly for the whole program. We execute our queries with both *Cerberus*$_{eager}$ and *Cerberus* and compare their performances.

Fig. 6 shows the average query execution time for the two different design choices of callgraph construction by our tool: *Cerberus* and *Cerberus*$_{eager}$. Y-axis shows the average time in seconds (in log scale) to answer queries for each library (X-axis) selected for the evaluation. Our analysis shows that our tool, *Cerberus*, which executes queries over an on-demand callgraph, is on average 25× faster than *Cerberus*$_{eager}$, which eagerly constructs callgraph. For example, *Spring Authorization Server* and *Node Oidc Provider* library take more than one hour to answer an OAuth query using *Cerberus*$_{eager}$. On the other hand, the same query is answered by 147 and 121 seconds, respectively, using the on-demand approach by *Cerberus*. This significant query execution time with *Cerberus*$_{eager}$ is because, by design, the analysis also visits the program components that are not directly relevant to the OAuth protocol (e.g., database models and operations). On the other hand, *Cerberus* only visits the program components relevant to the submitted OAuth query and therefore requires significantly less time to answer a given query.

However, we limit the execution time to 3 hours for executing the queries. In other words, the query execution automatically terminates if it can not find any result by 3 hours. We find that one of the large libraries, *OxAuth* ran into a memory explosion after running for more than 2 hours using *Cerberus*$_{eager}$. However, it takes 178 seconds to answer the same queries using *Cerberus*, which shows the effectiveness of our tool for analyzing large-scale implementations.

Comparison with existing tools. We further extend our effort to compare *Cerberus* against *OAuthlint*, which only supports OAuth properties with data-flow predicates. We managed to encode only three of our properties (out of ten) using their predicates with our best effort. As shown in Table 6, for the three properties supported by both tools, *Cerberus* is 22× and 21× faster than *OAuthlint* on $\texttt{DataSet}_K$ and $\texttt{DataSet}_U$, respectively.

Result for RQ2: Our query-driven analysis is 25× faster in checking OAuth security properties than the eager analysis.

6.2.5 Extensibility. To show the extensibility of *Cerberus*, we use a recent property from DPoP (Demonstrating Proof-of-Possession) [31], which is a new draft describing a mechanism for sender-constrained OAuth token. When sending a token request, the client sends an additional DPoP token in JWT format [1] that allows the server to verify the proof of possession of the token. The server first verifies the token using the key embedded within the token. If verified, it adds the thumbprint of the key with the access token. Otherwise, the token request is rejected with an error response. One can use *Cerberus*'s predicates to express this query as follows:

```
1. P :- invoke(L1, token_req), OAuthTag(L2, dpop),
2.      OAuthTag(L3, key), invoke(L4, jwt_verify),
3.      OAuthTag(L5, access_token), invoke(L6, add_thumbprint),
4.      error(L7, _), branch(L8,X,L6,L7), followBy(L1,L2),
5.      followBy(L1,L5), depOn(L2,L4),depOn(L3,L4),
6.      depOn(L4,X), depOn(L5,L6), depOn(L3,L6).
```

As shown from the above query, it is straightforward for a domain expert to use *Cerberus*'s built-in predicates to define new properties. Specifically, given a new property, a domain expert only needs to encode the new domain-specific information using predicates `OAuthTag` and `invoke`. E.g., for the DPoP property, the extra taint sources will be `dpop`, `key` and `jwt_verify`.

7 RELATED WORK

Since OAuth is a widely used and security-critical multi-party protocol, many researchers have studied OAuth at the protocol and implementation level for various platforms such as mobile [73], web [41, 69], and IoT [45, 65]. Yang et al. [79] study the attacker models to perform the common OAuth attacks (e.g., impersonation attacks, CSRF attacks) in web applications. Chen et al. [42] present the first in-depth study on OAuth attack vectors caused by the mistakes made by developers in client applications. However, manual analysis on client applications cannot be used to check logic flaws in OAuth server. Although, the manual approach can be effective for checking particular flaws, it is not scalable and might miss vulnerabilities. In comparison, *Cerberus* is the first automated and scalable tool to systematically check the logical flaws on OAuth server implementation. In addition, *Cerberus* defines new security properties and detects novel vulnerabilities in real-world OAuth server. Emerson et al. [47] propose an OAuth-based central access management system to provide a secure authentication scheme for IoT devices. Calzavara et al. [39] study the browser-side (i.e., client-side) security for using OAuth while considering the OAuth server as black-box. Veronese et al. [72] propose a network-traffic-based security monitoring system for different entities of OAuth. However, these studies are focused on the security implications with respect to the client-side flow and depend on manual analysis by security experts such as monitoring the network traffic or inferring the protocol flows and cannot be applied or extended to detect missing or incorrect security checks on the OAuth server implementation.

Researchers also study automated analysis to find security issues in client-side OAuth flow. Yang et al. [80] develop a symbolic execution-based testing tool to check the correctness of OAuth SDKs that provides APIs for client applications. Rahat et al. [64] build a static taint analysis tool to detect five categories of OAuth-specific data flow in Android applications. While these tools are effective for checking flaws on client applications, as we show in our evaluation, they are not deisgned to analyze the large scale codebase on OAuth server. Instead of analyzing the entire application, our tool is designed to automatically extract the OAuth relevant programs and explore the control and data flow to identify security critical logical mistakes.

Graph-based query approaches [55, 60] have been applied to solve a wide variety of security issues such as API misuse detection [32, 59, 63, 68, 70, 83], tainted data-flow analysis [43, 49, 57, 77, 82] and threat detection [54, 62]. He et al. implemented SSLint [56] that uses static analysis and graph signature matching to identify incorrect API usage for SSL. Egele et al. [46] studied cryptographic API misuse and used static program slicing to detect incorrect cryptographic operations in Android applications. These works focus on high-level API usage patterns, whereas *Cerberus* is designed to identify fine-grained properties (e.g., expressions). Graph traversal using different graph representations[33, 52, 76] of source code has also been used for checking code properties in various applications [40, 48, 66]. In addition to these classic representations, combined graphs [58] have also been used for vulnerability detection. For example, Yamaguchi et al. [75] use a combined graph representation called Code Property Graph to discover vulnerabilities. However, though these implementations are effective for checking common vulnerabilities (e.g., buffer overflows) in specific languages (e.g., C/C++), we cannot evaluate or compare against them as they do not support the languages (Java and Javascript) we target in this paper. In addition, these works eagerly construct the graphs for the entire program, which, as our evaluation demonstrates (Fig. 6), does not scale for large-scale programs such as OAuth server implementation.

8 DISCUSSION

While designing *Cerberus*, we strived to achieve a good trade-off between expressiveness and scalability. For complex semantics (e.g., secure computation, storage modeling, etc.) that are difficult to model precisely, our current static analysis leverages data- and control-dependencies to over-approximate the actual semantics, which, in theory, can lead to spurious execution paths. However, as discussed in Sec. 6, our tool achieves a low false positive rate despite our modeling. Secondly, dynamic features of Java and Javascript, such as reflective calls, dynamic class loading, and exceptional handling pose challenges for *Cerberus*. Those features can cause *Cerberus* to have false negatives (although such cases are rare based on our evaluation). Our current implementation of hybrid analysis can already handle some cases of dynamic features (e.g., reflective calls with string constants). We plan to further mitigate this issue by integrating Tamiflex [38] and ACG [50] tools for systematically reasoning about dynamic features into the *Cerberus* toolchain.

Our study in this paper focuses on OAuth 2.0, a prevalent multi-party protocol for authorization. *Cerberus* is designed based on the standard OAuth specification [21]. Therefore, any implementation that follows the standard specification can be analyzed using our proposed approach, which exhibits the generality of our work. Additionally, *Cerberus* can be extended to other relevant protocols like OpenID Connect [18]. Since OpenID Connect uses similar grants and flows as OAuth 2.0, the vulnerabilities we address in this paper are also applicable for OpenID Connect supported servers. In particular, *Cerberus*'s predicates can be extended to check OpenID properties using our query-based method. However, as our current analysis does not model the cryptographic APIs (e.g., RSA verification), some OpenID Connect flows (e.g., hybrid flow) that involve cryptographic operations cannot be checked by our tool. Additionally, it would be an interesting future work to apply *Cerberus* on a large scale of OAuth service provider implementations to check the prevalence of the issues we identified. The results might be similar because we study very popular libraries for OAuth service provider implementations, and developers usually just call these libraries' APIs instead of building their own implementations. Even worse, if developers start from scratch to build their service providers, they might make more security mistakes.

Finally, our analysis in this paper is focused on the attacks that utilize incorrect or logical implementation mistakes made during the OAuth flow. Our tool is designed to find violations in the implementation with respect to the queries representing the violation pattern. While our properties cover the OAuth-specific attacks commonly observed on the OAuth servers, attackers might exploit other vulnerabilities to attack the server. For example, Attackers might leverage generic web/mobile vulnerabilities and perform

web-based/mobile attacks (e.g., SQL-injection) to steal OAuth credentials or protected resources of the resource owner. Detecting those generic vulnerabilities is beyond the scope of this paper.

9 CONCLUSION

In this paper, we have presented *Cerberus*, an automated analyzer that can discover logic vulnerabilities in OAuth server libraries that service providers widely use. To efficiently detect OAuth violations in large codebases, *Cerberus* employs a query-driven algorithm for answering queries about security-critical OAuth properties. To demonstrate the effectiveness of *Cerberus*, we evaluate it on datasets of popular OAuth server libraries with millions of downloads. Among these high-profile libraries, *Cerberus* has discovered 47 vulnerabilities from ten classes of logic flaws, 24 of which were previously unknown and led to new CVEs.

10 ACKNOWLEDGEMENTS

We are grateful to the anonymous reviewers for their insightful and constructive feedback and suggestions. This work is supported in part by National Science Foundation under the agreement number of 1850479 and 1943100.

REFERENCES

[1] 2015. "Json Web Token". https://datatracker.ietf.org/doc/html/rfc7519.
[2] 2018. "Facebook Security Update". https://about.fb.com/news/2018/09/security-update.
[3] 2020. "Apifest Oauth2". https://github.com/apifest/apifest-oauth20.
[4] 2020. "Microsoft Azure Account Takeover". https://www.cyberark.com/resources/threat-research-blog/blackdirect-microsoft-azure-account-takeover.
[5] 2021. "A Method for Signing HTTP Requests for OAuth". https://tools.ietf.org/html/draft-ietf-oauth-signed-http-request-03.
[6] 2021. "ApiFest API Security". http://www.apifest.org.
[7] 2021. "Clouway: Oauth2 Server". https://github.com/clouway/oauth2-server.
[8] 2021. "GluuFederation: oxAuth". https://github.com/GluuFederation/oxAuth.
[9] 2021. "Node Oauth2 Server". https://github.com/oauthjs/node-oauth2-server.
[10] 2021. "Node Oidc Provider". https://github.com/panva/node-oidc-provider.
[11] 2021. "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens". https://tools.ietf.org/html/rfc8705.
[12] 2021. "OAuth 2.0 Security Best Current Practice". https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics.
[13] 2021. "OAuth 2.0 Threat Model and Security Considerations". https://tools.ietf.org/html/rfc6819.
[14] 2021. "Oauth2 Server". https://github.com/jobmission/oauth2-server.
[15] 2021. "Oauth2 Server". https://github.com/yoichiro/oauth2-server.
[16] 2021. "Oauth2 Server Node". https://github.com/af83/oauth2_server_node.
[17] 2021. "Oauth2orize". https://github.com/jaredhanson/oauth2orize.
[18] 2021. "OpenID Connect Core 1.0". https://openid.net/specs/openid-connect-core-1_0.html.
[19] 2021. "Proof Key for Code Exchange by OAuth Public Clients". https://tools.ietf.org/html/rfc7636.
[20] 2021. "Spring authorization server". https://github.com/spring-projects-experimental/spring-authorization-server.
[21] 2021. "The OAuth 2.0 Authorization Framework". https://tools.ietf.org/html/rfc6750.
[22] 2021. "T.J. Watson Libraries for Analysis (WALA)". http://wala.sourceforge.net/wiki/index.php/Main_Page.
[23] 2021. "Twitter Redirect URI Attack". https://hackerone.com/reports/110293.
[24] 2022. "Authlete Java OAuth". https://github.com/authlete/java-oauth-server.
[25] 2022. "Connect OAuth2". https://github.com/makesites/connect-oauth2.
[26] 2022. "Datalog". https://en.wikipedia.org/wiki/Datalog.
[27] 2022. "Egg OAuth2 Server". https://github.com/Azard/egg-oauth2-server.
[28] 2022. "Java Spring Server". https://github.com/mitreid-connect/OpenID-Connect-Java-Spring-Server.
[29] 2022. "Loopback OAuth Component". https://github.com/strongloop/loopback-component-oauth2.
[30] 2022. "Node OAuth20 Provider". https://github.com/t1msh/node-oauth20-provider.
[31] 2022. "OAuth 2.0 Security Best Current Practice". https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop.
[32] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. 2018. Precise Android API protection mapping derivation and reasoning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1151–1164.
[33] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: Principles, Techniques, & Tools*. Pearson Education India.
[34] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley. https://www.worldcat.org/oclc/12285707
[35] Lars Ole Andersen. 1994. Program analysis and specialization for the C programming language. In *PhD thesis*. University of Copenhagen, DIKU.
[36] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. 2013. Authscan: Automatic extraction of web authentication protocols from implementations. (2013).
[37] Gerard Berry and Ravi Sethi. 1986. From regular expressions to deterministic automata. *Theoretical computer science* 48 (1986), 117–126.
[38] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 241–250.
[39] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. 2018. WPSE: fortifying web protocols via browser-side security monitoring. In *27th USENIX Security Symposium*. 1493–1510.
[40] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework.. In *NDSS*.
[41] Yinzhi Cao, Yan Shoshitaishvili, Kevin Borgolte, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2014. Protecting Web Single Sign-on against Relying Party Impersonation Attacks through a Bi-directional Secure Channel with Authentication. (2014).
[42] 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*. 892–903.
[43] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. 2018. DTaint: detecting the taint-style vulnerability in embedded device firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 430–441.
[44] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*. 1–18.
[45] S. Cirani, M. Picone, P. Gonizzi, L. Veltri, and G. Ferrari. 2015. IoT-OAS: An OAuth-Based Authorization Service Architecture for Secure Services in IoT Scenarios. *IEEE Sensors Journal* 15, 2 (2015), 1224–1234.
[46] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 73–84.
[47] S. Emerson, Y. Choi, D. Hwang, K. Kim, and K. Kim. 2015. An OAuth based authentication mechanism for IoT networks. In *2015 International Conference on Information and Communication Technology Convergence (ICTC)*. 1072–1074.
[48] Aurore Fass, Michael Backes, and Ben Stock. 2019. Hidenoseek: Camouflaging malicious javascript in benign asts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1899–1913.
[49] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. 2021. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1789–1804.
[50] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 752–761.
[51] Yu Feng, Xinyu Wang, Isil Dillig, and Calvin Lin. 2015. EXPLORER : query- and demand-driven exploration of interprocedural control flow properties. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 520–534.
[52] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.
[53] 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*. 1204–1215.

[54] Peng Gao, Fei Shao, Xiaoyuan Liu, Xusheng Xiao, Haoyuan Liu, Zheng Qin, Fengyuan Xu, Prateek Mittal, Sanjeev R Kulkarni, and Dawn Song. 2021. A system for efficiently hunting for cyber threats in computer systems using threat intelligence. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2705–2708.

[55] Simon F Goldsmith, Robert O'Callahan, and Alex Aiken. 2005. Relational queries over program traces. *ACM SIGPLAN Notices* 40, 10 (2005), 385–402.

[56] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. 2015. Vetting SSL usage in applications with SSLint. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 519–534.

[57] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2014. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. *arXiv preprint arXiv:1404.7431* (2014).

[58] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node. js Vulnerabilities via Object Dependence Graph and Query. In *USENIX Security Symposium*.

[59] Tao Lv, Ruishi Li, Yi Yang, Kai Chen, Xiaojing Liao, XiaoFeng Wang, Peiwei Hu, and Luyi Xing. 2020. Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1837–1852.

[60] Michael Martin, Benjamin Livshits, and Monica S Lam. 2005. Finding application errors and security flaws using PQL: a program query language. *Acm Sigplan Notices* 40, 10 (2005), 365–383.

[61] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh. 2011. Formal Verification of OAuth 2.0 Using Alloy Framework. In *2011 International Conference on Communication Systems and Network Technologies*. 655–659.

[62] Kyuhong Park, Burak Sahin, Yongheng Chen, Jisheng Zhao, Evan Downing, Hong Hu, and Wenke Lee. 2021. Identifying Behavior Dispatchers for Malware Analysis. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 759–773.

[63] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng Yao. 2019. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2455–2472.

[64] Tamjid Al Rahat, Yu Feng, and Yuan Tian. 2019. OAUTHLINT: An Empirical Study on OAuth Bugs in Android Applications. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. 293–304.

[65] Savio Sciancalepore, Giuseppe Piro, Daniele Caldarola, Gennaro Boggia, and Giuseppe Bianchi. 2017. OAuth-IoT: An access control framework for the Internet of Things based on open standards. In *2017 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 676–681.

[66] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.. In *NDSS*, Vol. 1. 1–1.

[67] Soufflé Developers. 2020. Soufflé - Datalog. https://souffle-lang.github.io/index.html.

[68] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE. JS.. In *NDSS*.

[69] 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*. 378–390.

[70] Amann Sven, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2019. Investigating next steps in static API-misuse detection. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 265–275.

[71] Maarten H Van Emden and Robert A Kowalski. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)* 23, 4, 733–742.

[72] Lorenzo Veronese, Stefano Calzavara, and Luca Compagna. 2020. Bulwark: Holistic and Verified Security Monitoring of Web Protocols. In *European Symposium on Research in Computer Security*. Springer, 23–41.

[73] 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*. 61–70.

[74] Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 359–368.

[75] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.

[76] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 359–368.

[77] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 797–812.

[78] Feng Yang and Sathiamoorthy Manoharan. 2013. A security analysis of the OAuth protocol. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. IEEE, 271–276.

[79] F. Yang and S. Manoharan. 2013. A security analysis of the OAuth protocol. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. 271–276.

[80] Ronghai Yang, Wing Cheong Lau, Jiongyi Chen, and Kehuan Zhang. 2018. Vetting Single Sign-On SDK Implementations via Symbolic Reasoning. In *27th USENIX Security Symposium*. 1459–1474.

[81] 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*. 651–662.

[82] Zhemin Yang and Min Yang. 2012. Leakminer: Detect information leakage on android with static taint analysis. In *2012 Third World Congress on Software Engineering*. IEEE, 101–104.

[83] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 1105–1116.

[84] Yuchen Zhou and David Evans. 2014. SSOScan: automated testing of web applications for single sign-on vulnerabilities. In *23rd USENIX Security Symposium*. 495–510.

A APPENDIX: SECURITY PROPERTIES FOR OAUTH SERVER

In this section, we describe more details about the properties in Table 2. We identify these properties by analyzing standard OAuth specification [13, 21] and Security best practices [12].

Property P1: Redirect URI must match client's registered uri. With the redirect URI parameter provided by the client at the authorization request endpoint, the OAuth server must check if the URI is the same as the one registered to the client. As mentioned by the specification [21], the OAuth server must compare and match the redirect URI against the registered redirection URI. Failure to validate the redirect URI properly involves severe security implications. We demonstrate an attack example caused by incorrect redirect URI validation in Appendix B. Additionally, OAuth security best practices [12] states the complexity of implementing and managing to match URI correctly obviously causes security issues. The server must compare the two URIs using simple string comparison as defined in RFC-3986. However, if the redirect URI validation fails, the OAuth server should inform the resource owner and avoid making an automatic redirection to the redirect URI, as the URI is invalid and potentially be manipulated by the attackers. According to the specification [21], "if the request fails due to a missing, invalid, or mismatching redirection URI, ... the server should inform the resource owner of the error and must not automatically redirect the user-agent to the invalid redirection URI."

Property P2: Redirect URI must be an absolute URI. OAuth specification [21] requires that, "the redirection endpoint URI must be an absolute URI as defined by RFC-3986 Section 4.3." The URI value should not include a fragment component, since allowing the fragment in the redirect URI enables the attacker to utilize the open redirector of the user agent and intercept the authorization code or token attached with the redirect URI. These validation checks for redirect URI add an extra layer of security against the redirect URI manipulation by the attacker during the authorization request. An attack example for violating the redirect URI properties is demonstrated in Appendix B.

Property P3: Authorization code must be single-use. The specification states, "the client must not use authorization code more than once. If an authorization code is used more than once, the server must deny the request.". In addition to this check, the specification also recommends that if the server observes multiple attempts to exchange an authorization code for an access token, the server should reject the request with an error message and should revoke all access tokens already granted based on the compromised authorization code." This allows the server to protect the resources of a potentially compromised resource owner. Typically, the server stores the authorization code and the associated authorization information in its storage (e.g., database) during the authorization request and removes the code from the storage once it is used at the token endpoint by the client. Thus, if the submitted code at the token endpoint does not return any associated information from the storage, the server determines a potential malicious attempt of the authorization code being used for multiple times.

Property P4: Authorization code must be bound to client. As the specification states, "authorization code is bound to the client identifier". Precisely, authorization code submitted at token endpoint should be issued to the same client during the authorization request. As discussed in the authorization code injection attack scenario in Appendix B, this check by the OAuth server prevents the attacker from injecting an authorization code that was obtained from a different client application under the control of the attacker. To satisfy this property, the server must check that the client that initiated the token request with an authorization code is identical to the client to which the code was issued to.

Property (P5): Authorization code must be bound to the redirect URI. OAuth specification requires the OAuth server to ensure that the authorization code presented at the token endpoint is associated with the redirect URI submitted during the authorization request. Particularly, this validation ensures the authorization code used by the client was initially issued to the redirect URI that belongs to the client. This check is important to prevent the attacker from injecting an authorization code obtained by manipulating the redirect URI during the authorization request. As discussed in the authorization code injection attack scenario in Appendix B, the legitimate client would send the same redirect URI is used for the authorization request, but it would not match the manipulated redirect URI used by the attacker to obtain the code. Therefore, the checking at the token endpoint of the OAuth server would fail, and the token request would be rejected.

PKCE.. Satisfying the above properties is not enough to prevent the authorization code injection attack (Appendix B) for public clients (e.g., native desktop apps) as they may not have a server-side counterpart and cannot securely store the client credentials. Such clients often store the credentials as a hard-coded variable in the source code, which allows the attacker extracts the credentials and effectively break the client authentication during the token request. PKCE allows the OAuth server to authenticate the client at the token request without the client credentials and prevents attackers from performing authorization code injection attacks. OAuth security current best practices [12] states that "authorization servers must support PKCE". Although it was originally designed solely for native applications, it is now recommended to use PKCE for regular authorization code grant type. PKCE involves implementation at authorization and token endpoint and we describe the PKCE properties for both endpoints as following:

Property P6: Validate and store PKCE parameters at the authorization endpoint. PKCE utilizes a dynamically created cryptographic random key called $code\_verifier$. For every authorization request, a unique $code\_verifier$ is created by, and its transformed value, called *code_challenge* is sent to the server to obtain the authorization code. Clients often include the transformation method, called *code_challenge_method* with the authorization request. These parameters must be stored so that they can be accessed to verify the subsequent request received at the token endpoint.

Property P7: Verify PKCE parameters at the token endpoint. At the token endpoint, along with the authorization code, the OAuth server also receive a $code\_verifier$–the secret generated by the

client before initiating the authorization request. The server transforms the *code_verifier* and compares it with the *code_challenge* which it received from the client in previous request (i.e., authorization). However, the server denies the request (with an error response) if the values do not match, as it implies the *code* was not issued to the same client that made the request for the token. This approach eliminates the need to send the client secret at the token request endpoint to authenticate the client. The requirement to send client secret makes it difficult to store the secret securely, especially for public clients (e.g., mobile or native desktop apps) who cannot maintain client-side confidentiality. According to PKCE specification [19], the value of *code_challenge_method* that is used to transform the *code_verifier*, can either be 'S256' or 'plain'. The 'S256' is secure and recommended method since it protects against eavesdropping or intercepting the *code_challenge* and the challenge cannot be used without the *code_verifier*. On the other hand, as the term suggests, 'plain' does not provide any transformation of the *code_verifier*, and therefore, the value of *code_challenge* is exactly the same as the value of *code_verifier*. According to the PKCE specification for OAuth, "*plain* should not be used and exists only for compatibility with deployed implementations where the request path is already protected." Therefore, for this property, we only consider 'S256' as the value of the *code_challenge_method*.

Property P8: Use 'state' parameter to provide CSRF protection. CSRF allows an attacker to cause the user agent of a victim user to follow a malicious URI (e.g., redirection) to a legitimate server. In the OAuth context, a CSRF attack allows the attacker to inject a request to the legitimate client's redirection URI with its own authorization code or access token. It causes the client to use an access token associated with the attacker's resources instead of the victim's resources. As mentioned in the OAuth specification, "a CSRF attack against the server's authorization endpoint can result in an attacker obtaining end-user authorization for a malicious client without involving or alerting the end-user. The server must implement CSRF protection for its authorization endpoint. "OAuth servers often do not require clients to provide '*state*' parameter during the authorization request. This string value parameter helps to maintain the state between the client's authorization request and the server's authorization response. Thus, it provides clients a protection mechanism from Cross-site request forgery (CSRF) attack [13]. The satisfy this property, the server checks if the '*state*' parameter is present at the authorization endpoint and it associates the parameter value with the response, so that the client can validate the authenticity of the response received from the server.

Property P9: Access tokens should be constrained to a certain client. A client-constrained access token limits the applicability of an access token to a particular client. In other words, instead of a portable and plain access token, the OAuth server encrypts the token with a commonly distributed key between the client and the OAuth server. This binding mechanism also allows the client to demonstrate the proof of possession when exchanging the access token for resources. There have been several proposed approaches to demonstrate the proof of possession for access token such as mutual-TLS [11] and signed HTTP requests [5]. However, since the mutual TLS (also known as mTLS) is the most widely used and the only standardized client-constrained mechanism, in this paper, we consider the mutual TLS approach for implementing client-constrained approach, which is also recommended by the OAuth security current best practices [12]. In the mutual TLS approach, the OAuth server obtains the client's public key from the TLS stack during the token request at the token endpoint and associates the key with the access token before issuing the token to the client. This mechanism allows the server to verify the proof of possession when the token is submitted for requesting the resources. Therefore, an attacker can not inject a stolen access token to the legitimate client as the attack will be detected when the token is exchanged for the resources.

Property P10: Access tokens should not be stored as clear-text. The OAuth server must prevent the leakage of access tokens from its database. If the attacker gets access to the server's database, it can expose the access tokens of all the resource owners hosted by the corresponding server. The attacker might also steal the access token of a victim resource owner by performing an SQL injection attack. If the OAuth server stores the access token in clear text, the attacker can use the token for performing an access token injection attack as described above. The standard security considerations for OAuth [13] recommends to "store access token hashes only." However, prevention the common attacks such as SQL injection is out of the scope of this paper. Therefore, we focus on the mitigation of the OAuth attacks, such as access token injection.

B APPENDIX: ATTACKS ON OAUTH SERVER

This section describes the common attacks for the OAuth server. We describe an attack scenario and security impacts for each attack on different endpoints of the server.

B.1 Redirect URI Manipulation:

Lack of redirect URI validation (P1 & P2) by the OAuth server effectively breaks the client identification (authorization code grant) or authentication (implicit grant) and allows attackers to steal the authorization code or access token. Missing or incorrect validation of redirect URI allows the attacker to obtain the OAuth credentials by either (1) directly redirecting the user agent to a URI under the attacker's control or (2) exposing the credentials to an attacker by utilizing an open redirector at the client application by leveraging the way user agents handle URI fragments. For example, instead of complete redirect URIs, some servers allow clients to register to redirect URI patterns to support dynamic redirect URI for different sub-domains. Therefore, for the request at the authorization endpoint, the server matches the redirect URI parameter value against the registered patterns. Although this approach allows the clients to register one pattern for all sub-domains or encode transaction state into redirect URI parameter, it opens a gateway for the attacker to obtain the authorization code or access token sent from the authorization endpoint. For example, for a client using authorization code grant type, an attack may work as following:

B.1.1 Attack example. Assume an honest client intends to use any sub-domain of *honestclient.com* as redirect URI for their application and registers the redirect URI pattern *https://*.honestclient.com/** with the OAuth server. The server, however, might interpret the wildcard syntax '*' as a match for any character, and thereby, might

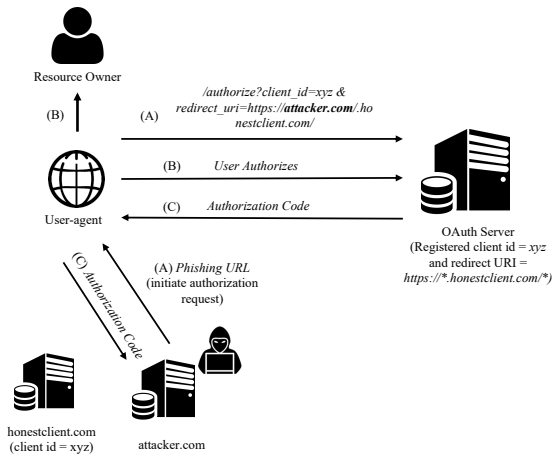

Figure 7: An attack scenario where attacker initiates an authorization request with a manipulated redirect URI parameter and obtains the authorization code of a victim resource owner.

recognize *https://attacker.com/.honestclient.com/* as a valid redirect URI, even though *attacker.com* is a different domain that the attacker could control. The attack scenario is illustrated in Fig. 7. First, the attacker takes a phishing approach (step A) to trick the resource owner to land on a malicious page that initiates an authorization request with the value of the *redirect_uri* parameter as *https://attacker.com/.honestclient.com/*. The resource owner thinks of the phishing web page as from the legitimate client and authorizes the page (step B). The server compares the received redirect URI with the redirect URI pattern registered by the client and processes the authorization request. The resource owner may not notice the malicious redirect URI as for some user agents (e.g., Android WebView), the URI is not visible to the users. Therefore, the authorization code is issued (step C) by the server and directly sent to the attacker's domain. Now the attacker has the authorization code, which it may use to impersonate the client and exchange the authorization code for the access token at the token endpoint of the same server. If the client uses the implicit grant type, this attack can even be worse as the attacker can directly obtain the access token using this approach.

B.2 Authorization Code Injection:

Violation of authorization code properties (P3, P4& P5) can result in an authorization code injection attack. In this attack, the attacker attempts to inject a stolen authorization code it obtained from the authorization endpoint of the server. There can be two attack scenarios for authorization code injection. First, an OAuth server with minimal security may not associate an authorization code with a particular client. Therefore, the attacker can easily obtain the access token from the token endpoint as it does not require authentication for any client. In the second scenario, the server binds the code with a particular client so that the code cannot be exchanged for an access token without an authenticated client. Therefore, in this attack scenario, the goal is to associate the attacker's session at the client with the victim resource owner, as the attacker cannot exchange the code for an access token by himself. This kind of attack is also useful when the attacker wants to impersonate his victim of a certain client application or a website. An authorization code injection attack may work as follows:

B.2.1 Attack example. To conduct an authorization code injection, the attacker first obtains an authorization code by performing a code intercept attack by manipulating the redirect URI as discussed above. Then he performs a regular authorization process with a legitimate client on his device. Since the authorization response passes through the attacker's device, the attacker can use any tool to intercept and manipulate the response to this end. The attacker injects the stolen authorization code in the response of the server to the legitimate client. The legitimate client then sends the code with the client's credentials to the token endpoint of the OAuth server. The server checks the authenticity of the client's credentials and if the code is issued to the particular client. If all checks succeed, server issues the access token to the client. Thus, the attacker has now successfully associated his session with the legitimate client with the victim resource owner.

B.3 Access Token Injection:

Violation of token properties (P9 & P10) can result in access token injection attack. The server needs to implement measures to protect its clients from this attack. In access token injection attack, the attacker attempts to impersonate a resource owner by injecting a stolen access token into a legitimate client. An attack scenario of access token injection is discussed as follows:

B.3.1 Attack example. To conduct an access token injection attack, the attacker first initiates the OAuth flow from a client application that uses the implicit grant for the authorization request. After successful authorization is obtained from the resource owner, the server issues an access token to the client application by using URI redirection through the user agent. The attacker modifies the response from the server and replaces the access token value with a stolen or leaked access token. As the attacker keeps all other parameters (e.g., *state* parameter) the same as the original request, the client does not recognize the response as a CSRF attack and uses the access token injected by the attacker.

Although clients using the implicit grant are most vulnerable to the access token injection attack, such attacks are also common for the clients that use the authorization code grant in an incorrect way. A significant number of mobile applications have been observed [42, 64] to use an incorrect authorization code grant where authorization request and token request are both made from the client applications. Since the attacker can modify any request or response at the client application, they can conduct the access token injection attack when the client uses an implicit grant or an incorrect authorization code grant. The authorization should implement the following properties to mitigate the risk of access token injection attacks.