OAuthShield: Efficient Security Checking for OAuth Service Provider Implementations

Tamjid Al Rahat  
Dept. of Computer Science  
University of Virginia  
tr9wr@virginia.edu

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

Yuan Tian  
Dept. of Computer Science  
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 libraries that are widely used by service providers. In this paper, we formalize the OAuth specifications and security best practices, and design OAuthShield, an automated static analyzer, to find logical flaws and identify vulnerabilities in the implementation of OAuth authorization server libraries. To efficiently detect OAuth violations in a large codebase, OAuthShield employs a demand-driven algorithm for answering queries about OAuth specifications. To demonstrate the effectiveness of OAuthShield, we evaluate it on ten popular OAuth libraries that have millions of downloads. Among these high-profile libraries, OAuthShield has identified 47 vulnerabilities from ten classes of logical flaws, 24 of which were previously unknown. We got acknowledged by the developers of six libraries, and had three accepted CVEs.

I. INTRODUCTION

OAuth has been widely used for authorizations across different software services. It defines a process for end-users (resource owners) to grant a third-party website/application (relying party [12], also noted as client or client applications) access to their private resources stored on a service provider, without giving the relying party their passwords to the service provider. As a multi-party protocol, the security of OAuth depends on the service providers (also noted as authorization server [12]), the relying parties, and the resource owners (end-users). In particular, vulnerabilities in OAuth service provider implementations could lead to serious consequences because that would impact all the relying parties (along with all the resource owners of the relying parties) that work with the service provider. For example, a vulnerability discovered in 2019 in the redirect URI validation mechanism of Microsoft’s authorization server allowed the attackers to take over Microsoft Azure Accounts [7]. Also, in 2018, a vulnerability in Facebook server-side implementation allowed the attackers to steal access tokens (issued by authorization server) of almost 50 million users [4].

Even well-known service providers such as Microsoft and Facebook still make severe mistakes in their authorization server implementation. This 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, there have been many previous works 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 investigated OAuth at the protocol level and propose formal verification tools [35], [37]. However, these tools only work at the protocol level and do not consider the diversified implementation details of OAuth service providers. Chen et. al. perform the first in-depth study for implementation issues of OAuth, by running manual analysis of mobile apps and the network traffic [29]. Later on, researchers develop semi-automated tools [26], [47] to report various security issues with OAuth relying party implementations. These analyses often need a heavy manual setup and do not scale. Most recently, researchers propose automatic tools for checking the implementation mistakes by relying parties [38], [45], assuming the service providers are securely implemented, which is clearly not the case according to our study in this paper. As a result, it is urgent to help developers check the security of the service provider implementations. As far as we know, there is no systematic security study of service provider (i.e., OAuth server) implementations. Most previous work in OAuth security either focuses on client applications, or reports security issues on an ad hoc basis.

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

To bridge the gap, we propose to systematically check the security of service provider implementations for OAuth authorization. We start by investigating popular open-source OAuth authorization server libraries. The reason for studying these open-source libraries is that we find many developers use these libraries to implement OAuth instead of starting from scratch due to its complexity. For example, Node OAuth2 Server [8] library is used by 2,300 repositories on Github. Since these open-source libraries are widely used by developers to implement OAuth protocols, checking the security of authorization server implementations in these libraries also provides us insights about a large number of OAuth imple-
mentations built on top of these libraries.

Challenges: However, developing an automatic tool for finding logical flaws of OAuth authorization 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.** Server-side implementation of OAuth (i.e., authorization server) protocols are typically complex and large, and often relies on many third-party libraries. Therefore, as we show later in Section VI, a naive whole-program analysis will not scale well.
- **Generality.** Checking OAuth properties in server-side implementations is not straightforward as they depend on various factors such as the client’s platform and types of grants (i.e., OAuth flows) used by the client during the OAuth transaction. Thus, it is non-trivial to devise a framework that unifies all properties enforced by the OAuth specifications.

Our solution: To address the above-mentioned challenges, we design and implement OAuthShield, a scalable and automated static analysis tool for detecting logical flaws in large-scale OAuth servers. 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 2.0 specifications [12] or OAuth security best practices [15]. We identify ten security properties for OAuth, five of which were not studied before (details are in Table II).

Second, given an OAuth property $Q$ encoded in our query language, we represent both the desired property $Q$ and the OAuth server application as system dependence graphs (SDGs). Therefore, OAuthShield converts the problem of checking OAuth properties into a graph query problem, which can be answered by an off-the-shelf Datalog engine [40]. 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.

However, there is a steep trade-off between the precision of an SDG and the cost of constructing it. For example, SDGs that are 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 [36, 32] to answer application-specific queries. To mitigate this challenge and only reason about program fragments that are relevant to the query, we introduce a demand-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 that are relevant to the OAuth property. In particular, we define OAuth request endpoint, which is the precondition of the OAuth property, specifying 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 ten most popular libraries that use OAuth 2.0 for implementing authorization 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). From ten classes of security-critical logical flaws in these libraries, in total, we identify 47 vulnerabilities, 24 of which were previously unknown. Five classes of logical flaws (violating property P4, P5, P6, P7, and P9 in Table II) are novel and studied by us for the first time. We got acknowledged by the developers of six libraries, among which four libraries (e.g., Spring Auth Server [23]). Node OAuth2 Server [8]) immediately took actions to fix the vulnerabilities. Three classes of new vulnerabilities in these libraries also lead us to have new CVE entries (e.g., CVE-2020-26877, CVE-2020-26937 and CVE-2020-26938[4]). These vulnerabilities can lead to account breach and confidential information leakage for both clients and resource owners. Three libraries have immediately fixed the vulnerabilities after we reported them, and the rest of the libraries are currently taking actions to fix the issues.

Contributions. We make the following contributions:

- We identify and formalize the security properties required to implement the OAuth protocol on the authorization server from the specifications and security best practices for OAuth.
- We, for the first time, design and implement an efficient automated tool to find logical flaws in OAuth authorization server implementation based on the standard specifications and security best practices, and we plan to make OAuthShield open-source.
- Our analysis finds that many open-source libraries for OAuth authorization servers make critical security mis-

| Tool          | Auto. | Covg. | Serv. | Ext. |
|---------------|-------|-------|-------|------|
| AuthScan [26] | ☐     | ☐     | ☐     | ☐    |
| SSOScan [47]  | ☐     | ☐     | ☐     | ☐    |
| OAuthTester   | ☐     | ☐     | ☐     | ☐    |
| S3kVetter [45]| ☐     | ☐     | ☐     | ☑    |
| OAuthLint [38]| ☐     | ☐     | ☐     | ☐    |

- Full support ☐ Partial support ☐ No support.

Auto.: Automated, Covg.: Coverage, Serv.: Server-side flaws, Ext.: Extensibility

1The CVEs are currently marked as “Reserved” and will be published after we make the vulnerabilities public
takes due to omitting or incorrectly implementing the security properties.

II. 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 apps (relying parties) to access their information hosted on other web services (identity 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. However, only two grants are widely used in practice—the 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. Figure 1 illustrates the implicit grant flow. First, the user is redirected to the service provider’s authorization server to grant the relying party (client) access to their protected resources. After the user grants permission, the authorization 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 service provider.

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 Figure 2. The authorization code grant augments the implicit grant by adding a step for authenticating the relying party (client). After the user grants permission to the relying party, the service provider redirects the user back to the relying party. Instead of providing the access token directly to the relying party, the service provider sends an authorization code this time. Then the relying party can use the authorization code to exchange for the access token by a server-to-server call. In this call to get the access token, the relying party needs to include its identity. As a result, the service provider 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, where the attackers intercept the authorization code returned from the authorization endpoint (step C in Figure 2) and obtain the access token by exchanging the code at the token endpoint (step D). To mitigate the risk of authorization code interception attack for public clients, OAuth specification introduces an extension of the authorization code grant called Proof Key for Code Exchange (PKCE) [22] and requires all authorization servers to support PKCE for public clients. It is worth mentioning that, although PKCE was originally designed to protect public clients, it is recommended [15] to use PKCE for all kinds of OAuth clients, including web applications.

Proof Key for Code Exchange (PKCE). PKCE is an extension of the authorization code grant designed to mitigate the authorization code injection attacks for public clients. Since public clients cannot maintain confidentiality and cannot securely store the client’s secret, using PKCE allows the authorization server to authenticate clients without the secret key. PKCE flow is illustrated in Figure 3. PKCE utilizes a dynamically created cryptographically 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 authorization server (step A) 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 (Step C) along with the authorization code it received from the previous request. To validate the proof of possession of
the code verifier by the client, the authorization server transforms the code verifier and compares it with the previously received code challenge. For example, for the code challenge method of ‘S256’, the server transforms the code verifier by using Base64UrlEncode(Sha256) and compares if the transformed value is equal to the code challenge, which it received in the previous state of the flow. 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.

III. Overview

This section briefly explains how our tool detects vulnerabilities in the OAuth-server implementation using a motivating example. We show how insufficient security checks in the authorization server allow malicious clients to steal sensitive OAuth credentials and take actions on behalf of users.

**Threat Model:** We aim to detect vulnerabilities in the OAuth authorization server (service provider) at the implementation level. We assume attackers can be a malicious relying party or a malicious resource owner (user) who interacts with the victim service provider. We assume the attackers cannot directly modify the source code or logic of the service provider but can attack the service provider by sending requests to the server. The relying party attackers control their own malicious relying party apps. For example, the relying party attackers might send malicious OAuth requests to the authorization server of 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 login 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, the relying party 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 control the user’s account.

When responding to an authorization request from a client, the authorization 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 allowlist of trusted URIs during the registration process to receive the OAuth tokens. However, while handling the authorization request, many authorization servers do not appropriately validate the value of the redirect_uri parameter, 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 authorization service providers, including the popular ones like Microsoft and Twitter. Similar attacks have been observed in the open-source authorization servers, as they also utilize the OAuth protocol.

For example, ApiFest is a popular open-source implementation of authorization server that uses OAuth 2.0 to provide a secure API management service to its clients. We identify a new vulnerability of incorrect redirect URI validation in this library (CVE-2020-26877). Figure 4 demonstrates a simplified implementation of their authorization request endpoint. ApiFest 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 Figure 4): (1) it does not check whether the redirect URI submitted with the request is registered to the corresponding client, (2) it omits the required sanitization for the redirect URI format (i.e., 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 attackers to craft the redirect_uri parameter of an authorization request with the attackers own redirect URI and thereby, steal the authorization code of a legitimate client when user agent redirects the code to the corresponding redirect URI. Secondly, as the server does not perform any sanitization for the submitted redirect URI, the attacker can also leverage open redirects of the user agent to steal the authorization code. Both of these mistakes also violate the standard OAuth specification as described in RFC 6749.

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 OAuthShield, to analyze the large program that implements an authorization server supported by OAuth. We first identify the security-sensitive properties based on the standard specification and security best practices for OAuth. Then, we meticulously design a query language to express the properties formally, so that they are understandable by the analysis tool and can easily be defined by the developers. OAuthShield then represents the program for the authorization server at the statement level, while maintaining the control- and data-flow relationship between the statements. However, as the implementation of an authorization server can be huge, running analysis on the statement-level representation for the whole program may not scale well. To overcome the scalability challenge, our analysis tool automatically identifies the program component that corresponds to OAuth. Finally, after OAuthShield pinpoints its scope in the OAuth relevant implementation in the program, it executes the queries to identify the violation of OAuth properties that might expose the authorization server to security attacks.

IV. System Design

In this section, we discuss the design and implementation of OAuthShield, our end-to-end static analysis tool for
Fig. 4: Code example for an authorization endpoint that is vulnerable to the redirect URI manipulation during the authorization flow.

Fig. 5: Overview of OAuthShield.

systematically checking the security issues of OAuth server implementation.

Figure 5 shows an overview of the OAuthShield approach for checking OAuth properties for Java or Javascript programs. OAuthShield 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, OAuthShield performs signature matching by checking whether there exists an embedding of the application with respect to property \( Q \).

A. Code Representation

Given a program, OAuthShield 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:

- \( V \) 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 to the entry node of the called procedure. Parameter-in edges connect the actual-in parameter nodes to the formal-in parameter nodes of the called procedure, and parameter-out edges connect the formal-out nodes to the actual-out 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: Figure 5(A) shows a simplified SDG constructed from the implementation (Figure 4) of ApiFest library [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.

B. Facts and Inference Rules for SDG

To enable efficient queries, OAuthShield converts the application’s SDG into its corresponding facts and rules using Datalog. In what follows, we first give some background on Datalog, and then describe the syntax and semantics of OAuthShield’s built-in predicates.

a) Datalog Preliminaries: A Datalog program consists of a set of rules and a set of facts. Facts simply declare predicates that evaluate to true. For example, parent(“Bill”, “Mary”) states that Bill is a parent of Mary. Each Datalog rule is a Horn clause defining a predicate as a conjunction of other predicates. For example, the rule:

\[
\text{ancestor}(x, y) :- \text{parent}(x, z), \text{ancestor}(z, y).
\]

says that ancestor \((x, y)\) is true if both parent \((x, z)\) and ancestor \((z, y)\) are true. In addition to variables, predicates can also contain constants, which are surrounded by double quotes, or “don’t cares”, denoted by underscores.

Datalog predicates naturally represent relations. Specifically, if tuple \((x, y, z)\) is in relation \( A \), this means the predicate \( A(x, y, z) \) is true. In what follows, we write the type of a relation \( R \subseteq X \times Y \times \ldots \) as \((s_1 : X, s_2 : Y, \ldots)\), where \( s_1, s_2, \ldots \) are descriptive texts for the corresponding domains.

b) Base Facts: The base facts of our inference engine describe the instructions in the application’s control-flow graph (CFG). The base facts take the form of \( A(L, y, x_1, \ldots, x_n) \), 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($l_1, r_1, 0$). alloc($l_1, y, x$) means that variable $x$ may point to abstract location $y$. 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 $L_3$. Using the base facts described above, OAuthShield computes two kinds of semantic facts: (i) control-dependency predicates, which capture instruction dependencies according to the application’s CFG, and (ii) data-dependency predicates.

c) SDG Predicates: OAuthShield provides built-in predicates to encode SDG generated from Section IV-A. In particular, the followTo($y, x$) predicate indicates that the value of variable $x$ has data-dependence on $y$. Similarly, the followBy predicate is inferred from the contract’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:

$$\text{flowTo}(x, y) : = \text{alloc}(\_ , y, x)$$
$$\text{flowTo}(x, y) : = \text{assign}(\_ , y, x)$$
$$\text{flowTo}(x, z) : = \text{assign}(\_ , y, x), \text{flowTo}(y, z)$$
$$\text{flowTo}(x, z) : = \text{alias}(y, z), \text{flowTo}(x, y)$$
$$\text{followBy}(x, y) : = \text{follow}(x, y)$$
$$\text{followBy}(x, y) : = \text{followBy}(y, z), \text{flowTo}(x, y)$$

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

d) OAuth-specific Predicates: In addition to basic facts from the SDG, OAuthShield’s query language also defines a list of predicates that are specific to the OAuth domain. The predicate $\text{OAuthTag}(L, T)$ defines that, the value at label $L$ is assigned tag $T$. Here, tag $T$ is associated with program statements that hold OAuth-specific re-

\begin{align*}
\phi & := \text{flowTo}(v_1, v_2) | \text{followBy}(v_1, v_2) | \text{branch}(...) \\
& | \text{OAuthTag}(x, y) | ... \\
& | \neg \phi | \phi \land \phi | \phi \lor \phi
\end{align*}

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
TABLE II: Security properties for the authorization server based on the standard specifications and best practices for OAuth. Among these ten security properties of authorization server, five (P4, P5, P6, P7, and P9) are novel and were not studied in previous works. Due to the space limitation, we provide more details of these properties in Appendix A and describe three attack examples caused for violating the properties in Appendix B.

match the redirect URI against the registered redirection URI. The redirect URI matching approach also involves security implications. An attack example caused by incorrect redirect URI validation is demonstrated in Appendix B.

However, if the redirect URI validation fails, the authorization server should inform the resource owner of the error and must not automatically redirect the user-agent to the invalid redirection URI.

Example 2: Figure 6(B) shows a graphical representation of the query that over-approximates the Redirect URI Property (P1) explained above. It describes 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.

Internally, the signature of the property in Figure 6(B) can be directly expressed using the following query. Here, the control- and data-dependent edges are directly translated into their corresponding predicates:

1. P1 :- OAuthTag(L1, auth_req), OAuthTag(L2, req_URI),
2. OAuthTag(L3, client_URI), OAuthTag(L4, redirect),
3. OAuthTag(L5, error), branch(L6, X, L4, L5),
4. flowTo(L2, X), flowTo(L3, X), followBy(L1, L2),
5. followBy(L1, L3)
This section discusses the design and implementation of OAuthShield, as well as a few key optimizations.

A. **Demand-driven Exploration**

OAuthShield leverages System Dependence Graph (SDG) that captures both control- and data-flow dependencies of a program. However, there is a steep trade-off between the precision of an SDG and the cost of constructing it. For example, SDGs that are 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 [36], [32] to answer application-specific queries. To mitigate this challenge, we introduce a demand-driven approach whose key insight is to only reason about small program fragments that are 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 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.

To compute the sub-callgraph given the OAuth request endpoints (e.g., token endpoint), we implement a lightweight program slicing based on automata theory. Specifically, OAuthShield first constructs the so-called query automaton (QA) 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, 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 $P$, OAuthShield accepts specifications written in the following query language:

$\text{Query } Q := f \in \text{methods}(P) \\
| . | Q_1 \rightarrow Q_2 \\
| Q_1 + Q_2 | Q^* | Q^+ | (Q)$

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 CGI 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 constructed the query and callgraph automata, the next step is to compute the intersection of those two using the JSA [30] tool. The output is a product automaton that encode a relevant program slice with respect to the query.

Example 3: Consider a CGI constructed from the callgraph of an input program and a QA constructed (Figure 7) from the given query $(* \rightarrow A : \text{foo} \rightarrow .* \rightarrow C : \text{bar})$, where the notation $X : f$ denotes the $f$ method in class X. Here, the query is used to represent the entry and exit point of OAuth endpoints. To compute the sub-callgraph relevant to a given query, OAuthShield constructs the product of the CGI and QA. As shown in Figure 8 partial product automaton that leads to an accepting state allows OAuthShield to compute the sub-callgraph relevant to the OAuth properties.

B. **Hybrid Analysis for Dynamic Features**

The system dependence graph may not capture the full semantics of some OAuth specifications. For instance, one CVE entry (CVE-2020-26938), identified by OAuthShield, is caused by checking if the redirect-uri value 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. On the other hand, a fully dynamic
analysis will be prohibitive since we have to deal with libraries with large codebase.

To mitigate the above-mentioned challenge, OAuthShield 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 OAuthShield. We iterate this process until OAuthShield confirms a violation or all false alarms are eliminated. For dynamic features such as reflective calls in Java, our current implementation can handle cases where reflective calls take arguments with string constants. In that case, we leverage the current data-flow analysis to keep track of the strings that may be used as class or function names.

C. The OAuthShield Tool

We implemented our core static analysis on top of the WALA framework [24], which provides compilation and analysis infrastructure for both Java and Javascript. OAuthShield’s implementation consists of approximately 9,700 lines of Java code. We use an Andersen-style pointer analysis and the CHA (Class Hierarchy Analysis) callgraph algorithm provided by WALA. We convert user-provided regular expressions to query automata using JSA [30], which is a general tool for performing string analysis of Java programs. OAuthShield also leverages the Soufflé [40] Datalog solver for checking conformance between the system dependenc graph and the OAuth properties as discussed in Section V-B.

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. 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. To resolve this, we use the pointer analysis to first 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.

VI. Evaluation

To determine the effectiveness of OAuthShield, we perform evaluation on popular OAuth-server libraries to answer the following research questions:

- RQ1: Can OAuthShield identify real-world vulnerabilities?
- RQ2: Is our demand-driven approach efficient and effective?

Our evaluation shows that OAuthShield can effectively discover the OAuth vulnerabilities and substantially improve the query answering time compared to the eager analysis approach.

A. OAuth Libraries

For our evaluation, we obtain ten high-profile open-source libraries that follow OAuth 2.0 for implementing the authorization server. To select the libraries, we consider their popularity among the web-developers community based on the number of downloads and stars. 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. Key statistics for the selected libraries are shown in Table III. Since all the libraries follow the standard OAuth specification [12], they represent generic OAuth service providers’ implementation. All of the libraries support the widely used authorization code grant, and three libraries also support the implicit grant. Four of these libraries are implemented in Javascript, and six of them are implemented in Java. Based on the statistics retrieved from Node Package Manager (npm) [10], the javascript library node-oauth2-server is downloaded approximately 200k times each month. Similarly, oauth2orize and node-oidc-provider are downloaded approximately 179k and 65k times each month. However, these numbers indicate the number of times the package is installed via the npm and do not count if the library is installed directly from the source code in Github (e.g., git clone). Table III also shows the number of stars retrieved from Github platform, which indicates the popularity of the libraries where a higher number of stars implies higher popularity. Regarding the size of the libraries, as indicated by the number of lines in Table III, some libraries (e.g., oxAuth, node-oidc-provider) are considerably larger than others. This is because these large libraries also support various other endpoints (e.g., token introspection endpoint) and platform-specific custom grants for their authorization server.

B. Experimental Setup and Results

In our experiments, we analyze ten high-profile OAuth-server libraries, among which four libraries are implemented in Javascript, and six are implemented in Java. All experiments are conducted on a Quad-Core Intel Core i5 computer and 16G of memory running on the macOS 10.15 operating system.

1) Discovered Vulnerabilities: To answer RQ1, we use OAuthShield to analyze ten open-sourced OAuth-server libraries to find vulnerabilities in their implementation of the authorization server. As presented in Table IV, we discovered in total 47 confirmed vulnerabilities in the ten popular
libraries of the OAuth authorization server, 24 of which were previously unknown. Security impacts of these vulnerabilities can cause various OAuth attacks (Appendix B) on the OAuth authorization server. We discuss these vulnerabilities and their impacts as follows:

**Redirect URI manipulation (violates P1 or P2).** OAuthShield successfully identified two libraries that violate the property that the redirect URI received during the authorization request must match the redirect URI registered by the client (P1). We found nine libraries that violate the property that the redirect URI must be an absolute URI and must not contain any fragment (P2). Violation of these properties regarding redirect URI allows the attackers to manipulate (Appendix B) the redirect URI while initiating the authorization request and obtain the authorization code or access token of a legitimate resource owner directly from the authorization server. Our finding indicates that developers often do not follow the check requirement for redirect URI, possibly because it may not seem to be security-critical from reading the specification.

**Authorization code injection (violates one of the following properties: P3, P4, P5, P6, P7).** OAuthShield successfully identified six libraries that violate at least one property which is required for the authorization server to protect against authorization code injection. We found six libraries that omit or incorrectly implement the property that the authorization code must be single-use (P3). This vulnerability allows the attackers to re-use the authorization code obtained by intercepting the authorization request made by a legitimate client and exchange the code at the token endpoint access token. We found two libraries that do not bind the authorization code to any particular client (P4), which allows the attackers to exchange the code (obtained from a legitimate client) by using a malicious client application. OAuthShield also successfully identified four libraries that do not bind the authorization code to the redirect URI (P5) used during the authorization request. As a result of this vulnerability, at the token endpoint, the authorization server can not verify that the redirect URI submitted during the token request is the same one where the authorization code was issued to. Each of these vulnerabilities leaves an open door for the attackers to perform an authorization code injection attack that allows the attackers to obtain a victim resource owner’s access token. We also found seven libraries that do not provide any support for Proof Key for Code Exchange (PKCE)—a recommended mechanism designed for the authorization server to mitigate against authorization code injection attack at the token endpoint (P6 & P7).

**Access token injection (violates P9 or P10).** OAuthShield successfully identified five libraries that violate at least one property required to protect against the access token injection attack. We found five libraries that do not issue client-constrained access tokens at the token endpoint (P9). A client-constrained access token limits the applicability of the token to a certain client and prevents the attackers from using an access token, which was previously obtained through a malicious client application. OAuthShield also successfully identified four libraries that stores the access token in clear

---

**TABLE III: Statistics for the open-source OAuth authorization server libraries used for evaluation.**

| OAuth-server libraries          | Version | Language | #stars (github) | Supported grants | #lines of code |
|--------------------------------|---------|----------|-----------------|------------------|-----------------|
| 1) node.oauth2-server          | 3.1.1   | JS       | 3,180           | Auth code        | 3,936           |
| 2) oauth2orize                 | 1.11.0  | JS       | 3,119           | Auth code, Implicit | 2,603          |
| 3) node-oidc-provider          | 6.29.5  | JS       | 1,291           | Auth code, Implicit | 15,493         |
| 4) oauth2-server-node          | 1.0     | JS       | 105             | Auth code        | 1,126           |
| 5) spring-authorization-server  | 2.3     | Java     | 1,124           | Auth code        | 9,858           |
| 6) clowway-oauth2-server       | 1.0.6   | Java     | 37              | Auth code        | 5,359           |
| 7) jobmission-oauth2-server    | 1.0     | Java     | 321             | Auth code        | 3,883           |
| 8) apitest-oauth20             | 0.3.1   | Java     | 67              | Auth code        | 14,371          |
| 9) yoichiro-oauth2-server      | 1.0     | Java     | 96              | Auth code        | 6,764           |
| 10) oxAuth                     | 3.0.2   | Java     | 276             | Auth code, Implicit | 62,857         |

**TABLE IV: Evaluation results for ten popular open-source OAuth-server libraries in Javascript and Java (✓ - library satisfies the property, × - library violates the property). Here, ×* indicates a false positive (FP). In other words, a library satisfies the desired property, but OAuthShield 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.**

| OAuth-server libraries | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 | P10 | #Violation | #FP | #FN |
|------------------------|----|----|----|----|----|----|----|----|----|----|------------|-----|-----|
| 1) node.oauth2-server  | ✓  | ×  | ✓  | ✓  | ✓  | ×  | ✓  | ✓  | ×  | ×  | 6          | 1   | 0   |
| 2) oauth2orize         | ✓  | ×  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ×  | ×  | 1          | 0   | 2   |
| 3) node-oidc-provider  | ✓  | ×  | ✓  | ✓  | ✓  | ×  | ×  | ×  | ✓  | ✓  | 5          | 1   | 0   |
| 4) oauth2-server-node  | ×* | ×  | ×  | ×  | ×* | ×  | ×  | ×  | ×  | ×  | 1          | 0   | 0   |
| 5) spring-authorization-server | ✓  | ×  | ✓  | ✓  | ✓  | ✓* | ×  | ×  | ✓  | ✓  | 4          | 0   | 0   |
| 6) clowway-oauth2-server | ✓  | ×  | ×  | ×  | ×  | ×  | ×  | ×  | ✓  | ✓  | 7          | 2   | 0   |
| 7) jobmission-oauth2-server | ×* | ×  | ×  | ×  | ×  | ×  | ×  | ×  | ×  | ×  | 8          | 1   | 0   |
| 8) apitest-oauth20     | ×* | ✓  | ✓  | ✓  | ×  | ×  | ×  | ×  | ✓  | ✓  | 8          | 0   | 0   |
| 9) yoichiro-oauth2-server | ×  | ×  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | 2          | 0   | 0   |
| 10) oxAuth             | ✓* | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | ✓  | 1          | 0   | 0   |

**Total vulnerabilities:** 2 9 6 2 3 7 7 2 5 4 47 7 3
text format in the storage (P10). Storing access tokens as clear-text is vulnerable as the attackers might be able to obtain access tokens from the authorization server by either gaining access to the database or launching a SQL injection attack.

**Vulnerable to CSRF (violates P8).** OAuthShield allowed us to successfully identify two libraries that do not provide any protection against CSRF-attack (P8) to its client applications. It is worth mentioning that the libraries supporting the PKCE feature are protected from CSRF protection by design. Also, libraries using the "nonce" parameter in OpenID Connect flow are also protected from CSRF. Libraries that neither support PKCE or OpenID Connect must handle the "state" parameter during the OAuth authorization request to protect its clients against CSRF. The libraries reported as vulnerable by OAuthShield do not provide any of the CSRF protection mechanisms mentioned above.

**Manual Assessment.** To verify the violations reported by OAuthShield, we manually deploy each library on a local server by following the documentation and implement the corresponding client-side program to simulate the OAuth flow, so that we can verify the property violations by observing the transactions during the flow. For example, to verify the redirect URI property P1 for oauth2-server-node library, 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 we replace the "redirect_uri" parameter of the request with the redirect URI from the malicious client. We observe the traces generated by the server, and if the server redirects the authorization response (with authorization code) to the malicious URI, we successfully verify that the library violates the redirect URI property P1.

**Analysis Accuracy.** As shown in Table IV, OAuthShield successfully identified 47 property violations and reports 7 false-positive (FP) cases and 3 false-negative (FN) cases for the ten libraries. We found that 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 few cases during Javascript library analysis. As our manual investigation found, the reasons for the FP and FN cases are 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., eval) are not currently modeled.**
- **Some Javascript libraries use complex frameworks such as Node.js**[11], which causes WALA to result in broken and imprecise callgraph that leads to both false positives and false negatives.

**Acknowledgements from developers.** We reported all the vulnerabilities to the developers of the libraries. So far, we have received confirmation from the developers of 6 libraries, who have acknowledged our findings and mentioned they were working on fixing the issues. By the time of writing this paper, two libraries have fixed the PKCE issues as we reported them, and mentioned the other remaining issues are undergoing a review process. Developers of the four libraries responded that they are taking actions regarding the issues we reported, and the issues will be addressed in their next release. The vulnerabilities that were unknown before in these libraries lead us to have three new CVE entries (e.g., CVE-2020-26877, CVE-2020-26937, and CVE-2020-26938). However, we have not yet received feedback from 4 libraries. We are making our best effort to reach the developers through different channels and help them to fix the vulnerabilities.

| Result for RQ1: |
|-----------------|
| We identified 47 vulnerabilities from ten popular libraries, including acknowledgement in six libraries and three accepted CVE entries. |

**2) Performance Evaluation:**

To answer RQ2 and evaluate the effectiveness of our demand-driven algorithm discussed in Section IV-A, we compare OAuthShield with its variant that constructs the callgraph for the entire program eagerly (notated as OAuthShield′). Unlike OAuthShield, which constructs callgraph on-demand and prunes away any program components that are not relevant to the OAuth query, OAuthShield′ does not consider any relevance with OAuth and constructs the callgraph eagerly for the whole program. We execute our queries corresponding to the OAuth properties with both OAuthShield′ and OAuthShield, and compare their performances.

Figure 9 shows the average query execution time for the two different design choices of callgraph construction by our tool: OAuthShield and OAuthShield′. Y-axis shows the average
time in seconds to answer queries for each library (X-axis) selected for the evaluation. Our analysis shows that our tool that executes queries over an on-demand callgraph, takes 65-90% less execution time than the eagerly constructed callgraph by OAuthShield'. For executing the queries, we limit the execution time to 10 minutes (600 seconds). In other words, the query execution automatically terminates if it cannot find any result by 10 minutes. However, as shown in Figure 9 when executing the queries with OAuthShield', 3 libraries – node-oicd-provider, apifest-oauth20 and oauth could not return any result within the limit of 10 minutes. To further investigate their performance, we increase the maximum time limit to 1 hour. With this extended setting, one library (apifest-oauth20) returned the result after running for 22 minutes, and two other libraries ran into memory explosion after running for 27 and 32 minutes, respectively. The reason for this significantly large query execution time with OAuthShield' 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). However, when we used OAuthShield for these three libraries, they successfully returned the results by 121, 123, and 178 seconds, respectively, which shows significant performance improvement by OAuthShield when compared to the OAuthShield'.

| Result for RQ2: Our on-demand analysis reduces the query execution time by 65-90%. |

C. Case Studies

In this section, we present our findings, with the help of OAuthShield, in two popular OAuth libraries. We further discuss the developer’s reactions and impacts upon our findings in these libraries.

Node OAuth2 Server This is one of the most popular authorization server libraries among the Node.js [11] community. As retrieved from npm platform, the library is downloaded on average 200k times each month. In addition, it is listed as a dependant of 2,300 other repositories in Github, meaning the authorization server APIs provided by this library is used by those repositories.

Even though the documentation claims the library is fully compliant with standard OAuth specifications, we run OAuthShield on this library and discovered five vulnerabilities, among which one vulnerability leads to a new CVE entry (CVE-2020-26938). When handling the redirect_uri parameter in an authorization request, the library does not properly sanitize the redirect URI before sending a redirection response that carries OAuth credentials. For example, to check if the redirect_uri value contains an absolute URI, the library matches it with an incorrect URI pattern “/[a-zA-Z][a-zA-Z0-9+.-]*:”. This incorrect check allows the attackers to have a malicious redirect URI to get passed through the authorization request and allows them to pass an XSS payload with the redirect_uri parameter. We reported this vulnerability to the developers, who acknowledge, “I agree. This is a severe issue and needs to be fixed”. However, along with incorrect handling of redirect URI, OAuthShield also identified that the library does not provide any PKCE support for the clients. In addition, the access token issued by the token endpoint of this library is not constrained to a certain client, which leads to the vulnerability of impersonating a legitimate user by exploiting the token. After our report, the developers immediately took actions, as they responded that the PKCE implementation is currently under review and will be merged soon, and the next release will fix the rest of the issues. Our investigation also found that OAuthShield reported one false negative on the check for state parameter. This false negative case was caused by the imprecision of WALA’s points-to analysis when analyzing the program’s dynamic initialization.

Spring Authorization Server. This library implements OAuth 2.0 to provide authorization server support for Spring – a popular and widely used application framework for the Java platform. OAuthShield successfully identified four violations of security-sensitive properties, which leaves the authorization server vulnerable to severe OAuth attacks like authorization code injection. We manually verified each of these violations. We found that when using the authorization code grant, the library does not revoke an authorization code once it is used by the client to obtain an access token. Violation of this property allows an attacker to re-use an intercepted authorization code at the token endpoint of the server and steal an access token of a legitimate resource owner. OAuthShield also reported that the library does not check whether the redirect URI submitted at the authorization or token endpoint contains any fragment. In addition, we discovered that this library does not provide any PKCE support, which leaves the public clients that use this authorization server vulnerable to OAuth attacks. As the library is widely used and very popular among the Java Spring community, we immediately reported these vulnerabilities to the developers who acknowledged our findings. Considering the severity of the reported vulnerabilities, the developers swiftly agreed to fix the issues. By the time of writing this paper, the developers added PKCE support for OAuth and mentioned the code revocation issue is under development and will be merged by the next release.

VII. RELATED WORK

Since OAuth is a widely used and security-critical multi-party protocol, many researchers have studied OAuth both at the protocol and implementation level for various platforms such as mobile [43], web [44], [41] and IoT [59], [31]. Yang et al. [44] study the attacker models to perform the common OAuth attacks (e.g., impersonation attacks, CSRF attacks) in web applications. Chen et al. [29] present the first in-depth study on OAuth attack vectors caused by the implementation mistakes or misunderstanding by developers in mobile applications. However, this study is based on manual investigations; thus, it is not scalable and might miss vulnerabilities. In addition, this study focuses on analyzing mobile
apps, and mostly identifying implementation issues of reply parties. In comparison, OAuthShield is the first automated and scalable tool to systematically check the OAuth service provides (servers) implementation vulnerabilities. In addition, OAuthShield defines new security properties and detects novel vulnerabilities in real-world implementations of OAuth service providers. Even if the security analysts can spend a large amount of manual efforts to check following the method in the above-mentioned paper \cite{29}, they won’t be able to check six of out ten properties we proposed (P3,4,5,6,7,10). This is because Chen et al.’s work focuses on analyzing mobile apps. Thus, their manual analysis techniques of the mobile apps cannot be used to check the logic flow between the relying party servers and service provider servers. Emerson et al. \cite{33} propose an OAuth-based central access management system to provide a secure authentication scheme for IoT devices. Calzavara et al. \cite{28} study the browser-side (i.e., client-side) security for using OAuth 2.0 while considering the service providers as black-box. Veronese et al. \cite{42} propose a network-traffic-based security monitoring system for different entities of OAuth. However, these studies are focused on the security implications on the client-side OAuth 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 server-side implementation.

Researchers also study automated analysis to find security issues in client-side OAuth flow. Yang et al. \cite{45} develop a symbolic execution-based testing tool to check the correctness of 10 popular OAuth SDKs and identified 7 vulnerabilities. Rahat et al. \cite{38} build a static taint analysis tool to detect the vulnerable OAuth-specific data flow in Android applications. These tools are designed to identify the client-side logical mistakes in OAuth authorization flow for a certain client type (i.e., public or confidential) and cannot be applied to detect the complex logical implementation on the server-side in a large-scale. Additionally, instead of analyzing the entire application, our tool is designed to automatically extract the OAuth relevant program component and explore the control and data flow to identify the complex logical mistakes in large programs for the OAuth server.

In recent years, researchers have been studying the challenges of detecting incorrect API usage in different security domains. He et al. implemented SSLint \cite{40} that uses static analysis and graph signature matching to identify incorrect API usage for SSL. Egele et al. \cite{32} 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 and do not scale well for large programs, whereas our tool is designed to identify fine-grained complex properties in a scalable fashion.

VIII. Discussion

While designing OAuthShield, 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 Section \cite{V}, 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 OAuthShield. Those features can cause OAuthShield to have false negatives (although such cases are very 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 \cite{27} and ACG \cite{34} tools for systematically reasoning about dynamic features into the OAuthShield toolchain.

Our study in this paper focuses on OAuth 2.0, a prevalent multi-party protocol for authorization. OAuthShield is designed based on the standard OAuth specification \cite{12}. Therefore, any implementation that follows the standard specification can be analyzed using our proposed approach, which exhibits the generality of our work. Additionally, OAuthShield can be extended to other relevant protocols such as OpenID Connect \cite{21}. Since OpenID Connect uses similar grants and flows as OAuth 2.0, the vulnerabilities we address in this paper are applicable for any OpenID Connect supported server. On top of that, OAuthShield can be used to check any OpenID Connect specific properties (e.g., authentication rules) using our query language (Section \cite{IV}). 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 OAuthShield 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.

Our analysis in this paper is focused on the attacks that utilize incorrect or logical implementation mistakes made during the OAuth flow. 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.

IX. Conclusion

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

X. ACKNOWLEDGEMENTS

This work is partially supported by NSF Awards (1850479 and 19431100).

REFERENCES

[1] ApiFest api security. http://www.apiFest.org
[2] ApiFest-oauth20. https://github.com/ApiFest/apiFest-oauth20
[3] Clouway: OAuth2-server. https://github.com/clouway/oauth2-server
[4] Facebook security update. https://about.fb.com/news/2018/09/security-update
[5] Gluufederation: oxauth. https://github.com/Gluufederation/oxAuth
[6] A method for signing http requests for oauth daff-oauth-signed-http-request-03. https://tools.ietf.org/html/draft-daff-oauth-signed-http-request-03
[7] Microsoft azure account takeover. https://black.direct
[8] node-oauth2-server. https://github.com/oauthjs/node-oauth2-server
[9] oauth2-server. https://github.com/yoichiro/oauth2-server
[10] Oauth2-server. https://github.com/jobmission/oauth2-server
[11] Oauth 2.0 security best current practice. https://tools.ietf.org/html/rfc6819
[12] The oauth 2.0 authorization framework. https://tools.ietf.org/html/rfc6750
[13] The oauth 2.0 authorization framework: Iwt pop token usage draft-sakimura-oauth-jpop-05. https://tools.ietf.org/html/draft-sakimura-oauth-jpop-05
[14] Oauth 2.0 mutual-tls client authentication and certificate-bound access tokens. https://tools.ietf.org/html/rfc8705
[15] Oauth 2.0 security best current practice. https://tools.ietf.org/html/draft-ietf-oauth-security-topics-15
[16] Oauth 2.0 threat model and security considerations. https://tools.ietf.org/html/rfc6819
[17] OAuth2-server. https://github.com/jobmission/oauth2-server
[18] oauth2-server. https://github.com/yoschiro/oauth2-server
[19] oauth2orize. https://github.com/jaredhanson/oauth2orize
[20] oauth2_server_node. https://github.com/af83/oauth2_server_node
[21] OpenID connect core 1.0. https://openid.net/specs/openid-connect-core-1_0.html
[22] Proof key for code exchange by oauth public clients. https://tools.ietf.org/html/rfc7636
[23] spring-authorization-server. https://github.com/spring-projects-experimental/spring-authorization-server
[24] T.j. watson libraries for analysis (wala). http://wala.sourceforge.net/wiki/index.php/Main_Page
[25] Twitter redirect uri attack. https://hackerone.com/reports/110293
[26] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayana Venkata-raman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. Authtscan: Automatic extraction of web authentication protocols from implementations. 2013.
[27] Eric Bodden, Andreas Sewe, Jan Sinscheck, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In Richard N. Taylor, Harold C. Gall, and Nenad Medvidovic, editors, Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011, pages 241–250. ACM, 2011.
[28] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. {WPSE}: fortifying web protocols via browser-side security monitoring. In 27th {USENIX} Security Symposium ({USENIX} Security 18), pages 1493–1510, 2018.
[29] Eric Y Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. OAuth demystified for mobile application developers. In Proceedings of the 2014 ACM SIGSAC conference on computer and communications security, pages 892–903, 2014.
[30] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings, pages 1–18, 2003.
[31] S. Cirani, M. Picone, P. Gonizzi, L. Veltri, and G. Ferrari. Iot-oas: An oauth-based authorization service architecture for secure services in iot scenarios. IEEE Sensors Journal, 15(2):1224–1234, 2015.
[32] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pages 73–84, 2013.
[33] S. Emerson, Y. Choi, D. Hwang, K. Kim, and K. Kim. An oauth based authentication mechanism for iot networks. In 2015 International Conference on Information and Communication Technology Convergence (ICTC), pages 1072–1074, 2015.
[34] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript IDE services. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, 35th International Conference on Software Engineering, ICSE ’13, San Francisco, CA, USA, May 18-26, 2013, pages 752–761. IEEE Computer Society, 2013.
[35] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of oauth 2.0. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 1204–1215, 2016.
[36] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrisnan, Runqing Yang, and Zhenrui Zhang. Vetting ssl usage in applications with sslint. In 2015 IEEE Symposium on Security and Privacy, pages 519–534. IEEE, 2015.
[37] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh. Formal verification of oauth 2.0 using alloy framework. In 2011 International Conference on Communication Systems and Network Technologies, pages 655–659, 2011.
[38] Tamjil Al Rahat, Yu Feng, and Yuan Tian. 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, pages 293–304, 2019.
[39] Savio Sciancalepore, Giuseppe Piro, Daniele Caldorala, Gennaro Boglia, and Giuseppe Bianchi. OAuth-iot: An access control framework for the internet of things based on open standards. In 2017 IEEE Symposium on Computers and Communications (ISCC), pages 676–681. IEEE, 2017.
[40] Soufflé Developers. Soufflé - datalog. https://souffle-lang.github.io/
datalog, 2020. [Online; accessed 13-Mar-2020].
[41] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of oauth sso systems. In Proceedings of the 2012 ACM conference on Computer and communications security, pages 378–390, 2012.
[42] Lorenzo Veronese, Stefano Calzavara, and Luca Compagna. Bulwark: Holistic and verified security monitoring of web protocols. In European Symposium on Research in Computer Security, pages 23–41. Springer, 2013.
[43] Hui Wang, Yuanuyan Zhang, Juannu Li, Hui Liu, Wenbo Yang, Bodong Li, and Dawu Gu. Vulnerability assessment of oauth implementations via symbolic reasoning. In Static Analysis, 18th International Symposium, SAS 2011, San Diego, CA, USA, September 14-17, 2011. Proceedings, pages 495–510, 2011.
[44] F. Yang and S. Manoharan. A security analysis of the oauth protocol. In 2015 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), pages 276–279, 2013.
[45] Ronghai Yang, Wing Cheong Lau, Jiongyi Chen, and Kehuan Zhang. Vetting single-sign-on (SSO) implementations via symbolic reasoning. In 27th {USENIX} Security Symposium ({USENIX} Security 18), pages 1497–1474, 2018.
[46] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. 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, pages 651–662, 2016.
[47] Yuchen Zhou and David Evans. Soscan: automated testing of web applications for single-sign-on vulnerabilities. In 23rd {USENIX} Security Symposium ({USENIX} Security 14), pages 495–510, 2014.
APPENDIX A
SECURITY PROPERTIES FOR OAUTH AUTHORIZATION SERVER

In this section, we describe more details about the properties in Table I. We identify these properties by analyzing standard OAuth specification [12] and Security best practices [13], [16]. As P1 is described in the main paper, we describe the other properties as following:

Redirect URI Property (P2): Redirect URI must be an absolute URI and must not contain any fragment. OAuth specification [12] also requires that, “the redirection endpoint URI must be an absolute URI as defined by [RFC3986] Section 4.3.” It also specifies that “the endpoint URI must not include a fragment component”. The fragment component in URI is typically followed by a hash mark (“#”). 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 [13].

Authorization Code Property (P3): Authorization code must be single-use. According to OAuth specification [12], “the client must not use authorization code more than once. If an authorization code is used more than once, the authorization server must deny the request.” In addition to this check, the specification also recommends that “if the authorization server observes multiple attempts to exchange an authorization code for an access token, the authorization server should attempt to revoke all access tokens already granted based on the compromised authorization code.” This allows the authorization server to protect the resources of a potentially compromised resource owner. Typically, the authorization 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. Using the query language and predicates defined in Section IV, the above property can be formally expressed using the following query:

1. P3 := OAuthTag(L1, token_req),
2. OAuthTag(L2, gen_token), OAuthTag(L3, db_read),
3. OAuthTag(L4, db_delete), OAuthTag(L5, error),
4. OAuthTag(L6, code), branch(L6, X, L4, L5),
5. FlowTo(L6, L3), FlowTo(L3, X), FlowTo(L6, L4),
6. followBy(L1, L2), followBy(L2, L4)

Authorization Code Property (P4): Authorization code must be bound to client. OAuth specification [12] states that “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 discussed above, this check by the authorization 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 authorization server must check that the client identifier that initiated the token request with an authorization code is identical to the client identifier to which the code was issued to.

Authorization Code Property (P5): Authorization code must be bound to the redirect URI. OAuth specification requires the authorization server to “… ensure that the redirect_uri parameter is present if the redirect_uri parameter was included in the initial authorization request as described in Section 4.1.1, and if included ensure that their values are identical.”. 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 above authorization code injection attack scenario, 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 authorization server would fail, and the token request would be rejected. To satisfy this property, the authorization server should store the complete redirect URI used in the authorization endpoint and compare it with the parameter value of redirect_uri at the token endpoint.

An attack example (authorization code injection) for violating the above-mentioned properties of authorization code is demonstrated in Appendix [13].

Satisfying the above properties is not enough to prevent the authorization code injection attack 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 authorization 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 [15] 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. As PKCE involves implementation at authorization and token request endpoint, we describe the PKCE properties for both endpoints as following:

PKCE Property (P6): Validate and securely store PKCE parameters at the authorization endpoint. PKCE utilizes a
PKCE Property (P7): Verify PKCE parameters at the token endpoint. At the token request endpoint, along with the authorization code, the authorization server also received a code_verifier—the secret generated by the client during the authorization request. The authorization server transforms the code_verifier and compares it with the code_challenge which it received from the client during the authorization request. The server denies the request 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. However, according to PKCE specification [22], the value of code_challenge method can either be "S256" or "plain". The "S256" method protects against eavesdropping or intercepting the code_challenge, because 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 when the transformation method is "plain." 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.

Cross-Site Request Forgery (CSRF). 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 [12], “a CSRF attack against the authorization 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 authorization server must implement CSRF protection for its authorization endpoint...". To protect clients from CSRF attack, the authorization server should satisfy the following property: CSRF Protection using “state” parameter (P8): Provide CSRF protection for authorization request. Authorization 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 [16]. However, using PKCE or OpenID Connect’s “nonce” value provides default protection against CSRF. Therefore, if the authorization server does not support PKCE or OpenID Connect, it should enforce clients to use the “state” parameter during the authorization request. To satisfy this property, the authorization server should either support PKCE or OpenID Connect. Otherwise, it should check if the parameter value for state presents in the authorization request.

Token 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 authorization server encrypts the token with a commonly distributed key between the client and the authorization 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 [14], signed HTTP requests [6] and JWT Pop Tokens [13]. However, since the mutual TLS 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 [15]. In the mutual TLS approach, the authorization 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 cannot inject a stolen access token to the legitimate client as the attack will be detected when the token is exchanged for the resources.

Using the query language and predicates defined in Section IV, the mutual TLS approach for the above property can be formally expressed using the following query:

1. P9 :- OAuthTag(L1, access_token),
2. OAuthTag(L2, client_cert), OAuthTag(L3, b64_decode), OAuthTag(L4, b64_encode),
3. OAuthTag(L5, sha256), OAuthTag(L6, add_cert),
4. FlowTo(L2, L3), FlowTo(L3, L5), FlowTo(L5, L4),
5. FlowTo(L1, L6), FlowTo(L4, L6),

Token Property (P10): Access tokens should not be
stored as clear-text. The authorization server must prevent the leakage of access tokens from its database. If the attacker gets access to the authorization server’s database, it can expose the access tokens of all the resource owners hosted by the corresponding authorization server. The attacker might also steal the access token of a victim resource owner by performing an SQL injection attack. However, preventing an SQL injection attack is out of the scope of this paper. If the authorization 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 [16] 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.

APPENDIX B
ATTACKS ON OAuth AUTHORIZATION SERVER

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

Redirect URI Manipulation: Lack of redirect URI validation (P1 & P2) by the authorization 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 authorization 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 authorization 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:

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 authorization server. The authorization server, however, might interpret the wildcard syntax ‘*’ as a match for any character, and thereby, might 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 Figure 10. 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 authorization 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 authorization 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 authorization 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.

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 authorization 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 authorization 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:

**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 authorization server to the legitimate client. The legitimate client then sends the code with the client’s credentials to the token endpoint of the authorization 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, the authorization 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.

**Access Token Injection:** Violation of token properties (P9 & P10) can result in access token injection attack. The authorization 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:

**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 authorization server issues an access token to the client application by using URI redirection through the user agent. The attacker modifies the response from the authorization 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 [29], [38] 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.